Unicode Utilities Reference

Carbon > Text & Fonts

2006-01-10

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

Simultaneously published in the United States and Canada.

Contents

Unicode Utilities Reference 7

Overview 7
Functions by Task 7
- Inputting Unicode Text 7
- Comparing Unicode Strings 7
- Identifying Unicode Text Boundaries 8

Functions 8
- UCCompareCollationKeys 8
- UCCompareText 9
- UCCompareTextDefault 11
- UCCompareTextNoLocale 12
- UCCreateCollator 13
- UCCreateTextBreakLocator 14
- UCDisposeCollator 16
- UCDisposeTextBreakLocator 16
- UCFindTextBreak 17
- UCGetCollationKey 18
- UCKeyTranslate 20

Data Types 22
- CollatorRef 22
- TextBreakLocatorRef 22
- UCCollationValue 23
- UCKeyboardLayout 23
- UCKeyboardTypeHeader 24
- UCKeyCharSeq 25
- UCKeyLayoutFeatureInfo 26
- UCKeyModifiersToTableNum 27
- UCKeyOutput 27
- UCKeySequenceDataIndex 28
- UCKeyStateEntryRange 29
- UCKeyStateEntryTerminal 30
- UCKeyStateRecord 31
- UCKeyStateRecordsIndex 32
- UCKeyStateTerminators 33
- UCKeyToCharTableIndex 34

Constants 35
- Fixed Ordering Scheme 35
- Fixed Ordering Masks 1 35
- Fixed Ordering Masks 2 36
- Key Actions 36
- Key Format Codes 37

Key Output Index Masks 38
Key State Entry Formats 39
Key Translation Options Flag 39
Key Translation Options Mask 40
Operation Class 40
Standard Options Mask 41
String Comparison Options 41
Text Break Options 42
Text Break Types 43
Text Boundary Operation Class 44

Appendix A **Specification for 'uchr' 45**

Unicode Keyboard-Layout Resource Format 45

Document Revision History 51

Index 53

Figures

Appendix A **Specification for 'uchr' 45**

Figure A-1 Overview of a 'uchr' resource 45
Figure A-2 'uchr' resource header 46
Figure A-3 'uchr' modifier combination to key-code-to-character table number map 47
Figure A-4 'uchr' key-code-to-character tables 47
Figure A-5 'uchr' dead-key state records 48
Figure A-6 'uchr' default dead-key state terminators 49
Figure A-7 'uchr' character key sequences 49

Unicode Utilities Reference

Framework: CoreServices/CoreServices.h

Declared in UnicodeUtilities.h

Overview

Unicode Utilities allow applications and text service components (such as input methods) to perform various operations on Unicode text; for example, Unicode key translation. Resources defined for use with Unicode Utilities permit control of Unicode-related text behavior, such as the specification of Unicode keyboard layouts.

Carbon fully supports the Unicode Utilities.

Functions by Task

Inputting Unicode Text

`UCKeyTranslate` (page 20)
Converts a combination of a virtual key code, a modifier key state, and a dead-key state into a string of one or more Unicode characters.

Comparing Unicode Strings

`UCCreateCollator` (page 13)
Creates an object encapsulating locale and collation information, for the purpose of performing Unicode string comparison.

`UCCompareText` (page 9)
Uses locale-specific collation information to compare Unicode strings.

`UCGetCollationKey` (page 18)
Uses locale-specific collation information to generate a collation key for a Unicode string.

`UCCompareCollationKeys` (page 8)
Uses collation keys to compare Unicode strings.

`UCDisposeCollator` (page 16)
Disposes a collator object.

`UCCompareTextDefault` (page 11)
Uses the default system locale to compare Unicode strings.

`UCCompareTextNoLocale` (page 12)
Uses a fixed, locale-insensitive order to compare Unicode strings.

Identifying Unicode Text Boundaries

`UCCreateTextBreakLocator` (page 14)
Creates an object encapsulating locale and text-break information, for the purpose of finding boundaries in Unicode text.

`UCFindTextBreak` (page 17)
Uses locale-specific text-break information to find boundaries in Unicode text.

`UCDisposeTextBreakLocator` (page 16)
Disposes a text-break locator object.

Functions

UCCompareCollationKeys

Uses collation keys to compare Unicode strings.

```
OSStatus UCCompareCollationKeys (
   const UCCollationValue *key1Ptr,
   ItemCount key1Length,
   const UCCollationValue *key2Ptr,
   ItemCount key2Length,
   Boolean *equivalent,
   SInt32 *order
);
```

Parameters

key1Ptr
A pointer to the collation key (a `UCCollationValue` array) for the first string to compare. You can obtain a collation key with the function `UCGetCollationKey` (page 18). The collation key supplied in `key1Ptr` for the first string must be generated with the same collator object as that used to generate the collation key supplied in `key2Ptr` for the second string.

key1Length
An `ItemCount` value specifying the actual length of the collation key supplied in the `key1Ptr` parameter. You can obtain this value from the function `UCGetCollationKey` (page 18) when you obtain the new collation key.

key2Ptr
A pointer to the collation key (a `UCCollationValue` array) for the second string to compare. You can obtain a collation key with the function `UCGetCollationKey` (page 18). The collation key supplied in `key2Ptr` for the second string must be generated with the same collator object as that used to generate the collation key supplied in `key1Ptr` for the first string.

key2Length
An `ItemCount` value specifying the actual length of the collation key supplied in the `key2Ptr` parameter. You can obtain this value from the function `UCGetCollationKey` (page 18) when you obtain the new collation key.

equivalent

A pointer to a `Boolean` value or pass `NULL`. On return, `UCCompareCollationKeys` produces a value of `true` if the strings represented by the collation keys are equivalent for the options you have specified in the collator object. If you wish simply to sort a list of strings in order, using your specified options, you can pass `NULL` for the `equivalent` parameter and only use the `order` parameter's result. In this case, all available comparison criteria are used to put the strings in a deterministic order, even if they are considered "equivalent" for the options you have specified. Note that you can set either the `equivalent` or the `order` parameters to `NULL`, but not both.

order

A pointer to a signed, 32-bit integer value, or pass `NULL`. If you wish simply to test the strings represented by the collation keys for equivalence, using your specified options (which can be much faster than determining ordering), you can pass `NULL` for the `order` parameter and only use the `equivalent` parameter's result. (Note that either the `equivalent` or the `order` parameters may be `NULL`, but not both.

Return Value
A result code. This function can return `paramErr`, for example, if `key1Ptr` or `key2Ptr` are `NULL`.

Discussion
If you wish to compare the same strings several times, as when sorting a list of strings, it may be most efficient for you to derive a collation key for each string and then compare the collation keys. A collation key is a transformation of the string that depends on the collator object (that is, it depends on the locale, the collation variant if any, and the collation options).

Collation keys that are generated using the same collator object—but for different strings—can quickly be compared with each other, without further reference to the collator object or collation tables. The disadvantage is that the collation keys may be rather large. After you use the function `UCGetCollationKey` (page 18) to create a collation key from a given string and collator object, you can call the `UCCompareCollationKeys` function to compare two collation keys that were generated with the same collator object.

If you are comparing different strings, it may be more efficient for you to call the function `UCCompareText` (page 9) multiple times using the same collator object.

Note that collation keys should be used only in a runtime context. They should not be stored in a persistent state (such as to disk) because the format of a collation key could change in the future.

Availability
Available in CarbonLib 1.0 and later when running Mac OS 8.6 or later.
Available in Mac OS X 10.0 and later.

Declared In
`UnicodeUtilities.h`

UCCompareText

Uses locale-specific collation information to compare Unicode strings.

```
OSStatus UCCompareText (
   CollatorRef collatorRef,
   const UniChar *text1Ptr,
   UniCharCount text1Length,
   const UniChar *text2Ptr,
   UniCharCount text2Length,
   Boolean *equivalent,
   SInt32 *order
);
```

Parameters

collatorRef

A valid reference to a collator object; `NULL` is not allowed. You can use the function `UCCreateCollator` (page 13) to obtain a collator reference.

text1Ptr

A pointer to the first Unicode string (a `UniChar` array) to compare.

text1Length

The total count of Unicode characters in the first string being compared.

text2Ptr

A pointer to the second Unicode string to compare.

text2Length

The total count of Unicode characters in the second string being compared.

equivalent

A pointer to a `Boolean` value or `NULL`. On return, `UCCompareText` produces a value of `true` if the strings are equivalent for the options you have specified in the collator object. If you wish simply to sort a list of strings in order, using your specified options, you can pass `NULL` for the `equivalent` parameter and only use the `order` parameter's result. In this case, all available comparison criteria are used to put the strings in a deterministic order, even if they are considered "equivalent" for the options you have specified. Note that you can set either the `equivalent` or the `order` parameters to `NULL`, but not both.

order

A pointer to a signed, 32-bit integer value, or pass `NULL`. If you wish simply to test strings for equivalence, using your specified options (which can be much faster than determining ordering), you can pass `NULL` for the `order` parameter and only use the `equivalent` parameter's result. (Note that either the `equivalent` or the `order` parameters may be `NULL`, but not both.

Return Value

A result code. The function can return `paramErr` (for example, if `collatorRef`, `text1Ptr`, or `text2Ptr` are `NULL`.

Discussion

You can use the `UCCompareText` function to perform various types of string comparison for a given set of locale and collation specifications. You can

- simply test whether two strings are equivalent
- determine the relative ordering of two strings
- check whether a given string is equivalent to any string in an ordered list

You can also call the `UCCompareText` function multiple times to compare different strings using the same collator object. If you wish to compare the same strings several times, as when sorting a list of strings, it may be more efficient for you to derive a collation key for each string and then compare the collation keys. For more on comparison using collation keys, see the functions `UCGetCollationKey` (page 18) and `UCCompareCollationKeys` (page 8).

Availability
Available in CarbonLib 1.0 and later when running Mac OS 8.6 or later.
Available in Mac OS X 10.0 and later.

Declared In
`UnicodeUtilities.h`

UCCompareTextDefault

Uses the default system locale to compare Unicode strings.

```
OSStatus UCCompareTextDefault (
   UCCollateOptions options,
   const UniChar *text1Ptr,
   UniCharCount text1Length,
   const UniChar *text2Ptr,
   UniCharCount text2Length,
   Boolean *equivalent,
   SInt32 *order
);
```

Parameters

options
A `UCCollateOptions` value specifying any collation options for the string comparison.

text1Ptr
A pointer to the first Unicode string (a `UniChar` array) to compare.

text1Length
The total count of Unicode characters in the first string being compared.

text2Ptr
A pointer to the second Unicode string to compare.

text2Length
The total count of Unicode characters in the second string being compared.

equivalent
A pointer to a `Boolean` value or pass `NULL`. On return, `UCCompareTextDefault` produces a value of `true` if the strings are equivalent for the options you have specified. If you wish simply to sort a list of strings in order, using your specified options, you can pass `NULL` for the `equivalent` parameter and only use the `order` parameter's result. In this case, all available comparison criteria are used to put the strings in a deterministic order, even if they are considered "equivalent" for the options you have specified. Note that you can set either the `equivalent` or the `order` parameters to `NULL`, but not both.

order

A pointer to a signed, 32-bit integer value, or pass `NULL`. If you wish simply to test the strings for equivalence, using your specified options (which can be much faster than determining ordering), you can pass `NULL` for the `order` parameter and only use the `equivalent` parameter's result. (Note that either the `equivalent` or the `order` parameters may be `NULL`, but not both.

Return Value
A result code.

Discussion
You can call the `UCCompareTextDefault` function when you want to use a simple collation function that requires minimum setup. This function uses the system default collation order (that is, the collation order for a `LocaleRef` of `NULL` and a variant of 0), and it does not require a collator object or collation keys.

Availability
Available in CarbonLib 1.0 and later when running Mac OS 8.6 or later.
Available in Mac OS X 10.0 and later.

Declared In
`UnicodeUtilities.h`

UCCompareTextNoLocale

Uses a fixed, locale-insensitive order to compare Unicode strings.

```
OSStatus UCCompareTextNoLocale (
   UCCollateOptions options,
   const UniChar *text1Ptr,
   UniCharCount text1Length,
   const UniChar *text2Ptr,
   UniCharCount text2Length,
   Boolean *equivalent,
   SInt32 *order
);
```

Parameters

options

A `UCCollateOptions` value specifying the fixed ordering scheme to use for the string comparison. This value must be nonzero. Bits 24-31 of `UCCollateOptionsValue` specify which fixed ordering scheme to use. Currently there is only scheme—`kUCCollateTypeHFSExtended`. See "Fixed Ordering Scheme" (page 35) for additional details.

text1Ptr

A pointer to the first Unicode string (a `UniChar` array) to compare.

text1Length

The total count of Unicode characters in the first string being compared.

text2Ptr

A pointer to the second Unicode string to compare.

text2Length

The total count of Unicode characters in the second string being compared.

equivalent

A pointer to a `Boolean` value or pass `NULL`. On return, `UCCompareTextNoLocale` produces a value of `true` if the strings are equivalent for the ordering scheme you have specified. If you wish simply to sort a list of strings in order, using the specified ordering scheme, you can pass `NULL` for the `equivalent` parameter and only use the `order` parameter's result. In this case, all available comparison criteria are used to put the strings in a deterministic order, even if they are considered "equivalent" for the specified ordering scheme. Note that you can set either the `equivalent` or the `order` parameters to `NULL`, but not both.

order

A pointer to a signed, 32-bit integer value, or pass `NULL`. If you wish simply to test the strings for equivalence, using the specified ordering scheme (which can be much faster than determining ordering), you can pass `NULL` for the `order` parameter and only use the `equivalent` parameter's result. (Note that either the `equivalent` or the `order` parameters may be `NULL`, but not both.

Return Value
A result code. This function can return `paramErr` if you pass an invalid value for one of the parameters. For example, if you pass `0` for the `options` paramter, the function returns `paramErr`.

Discussion
You can call the `UCCompareTextNoLocale` function when you want to perform a fixed, locale-insensitive comparison that is guaranteed not to change from one system release to the next. This type of comparison could be used for sorting a Unicode key string in a database, for example. The `UCCompareTextNoLocale` function can provide comparison according to various fixed ordering schemes (only one is supported for Mac OS 8.6 and 9.0). This type of comparison is not usually used for a user-visible ordering, so the ordering schemes need not match any user's expectation of a sensible collation order.

The `UCCompareTextNoLocale` function does not require a collator object or collation keys. Another advantage of `UCCompareTextNoLocale` on Mac OS 9 is that it is exported from the `UnicodeUtilitiesCoreLib` library, which does not depend on other libraries (the other comparison functions exported from `UnicodeUtilitiesLib`, which depends on `LocalesLib` and `TextCommon`).

Availability
Available in CarbonLib 1.0 and later when running Mac OS 8.6 or later.
Available in Mac OS X 10.0 and later.

Declared In
`UnicodeUtilities.h`

UCCreateCollator

Creates an object encapsulating locale and collation information, for the purpose of performing Unicode string comparison.

```
OSStatus UCCreateCollator (
   LocaleRef locale,
   LocaleOperationVariant opVariant,
   UCCollateOptions options,
   CollatorRef *collatorRef
);
```

Parameters

locale

A valid `LocaleRef` representing a specific locale, or pass `NULL` to request the default system locale. You can supply the value `kUnicodeCollationClass` in the `opClass` parameter of the Locales Utilities functions `LocaleOperationCountLocales` and `LocaleOperationGetLocales` to obtain the locales available for collation on the current system.

opVariant

A `LocaleOperationVariant` value identifying a collation variant within the locale specified in the `locale` parameter. You can also pass 0 to request the default collation variant for any locale. To obtain the varieties of locale-specific collation that are currently available, you can supply the value `kUnicodeCollationClass` in the `opClass` parameter of the Locales Utilities functions `LocaleOperationCountLocales` and `LocaleOperationGetLocales`.

options

A `UCCollateOptions` value specifying any collation options that you want to use for the string comparison.

collatorRef

A pointer to a value of type `CollatorRef`. On return, the `CollatorRef` value contains a valid reference to a new collator object.

Return Value

A result code. The function can return memory errors and `paramErr`, for example, if the `collatorRef` parameter is `NULL`. It can also return resource errors in Mac OS 9 and CarbonLib.

Discussion

To perform Unicode string comparison, you must supply locale and collation specifications to a collation function such as `UCCompareText` (page 9). You provide this information by means of a collator object, created via the `UCCreateCollator` function. When finished with the collator object, you dispose of it using the function `UCDisposeCollator` (page 16).

Special Considerations

The collator object is allocated in the current heap. This function can move memory.

Availability

Available in CarbonLib 1.0 and later when running Mac OS 8.6 or later.

Available in Mac OS X 10.0 and later.

Declared In

`UnicodeUtilities.h`

UCCreateTextBreakLocator

Creates an object encapsulating locale and text-break information, for the purpose of finding boundaries in Unicode text.

```
OSStatus UCCreateTextBreakLocator (
   LocaleRef locale,
   LocaleOperationVariant opVariant,
   UCTextBreakType breakTypes,
   TextBreakLocatorRef *breakRef
);
```

Parameters

locale

A valid `LocaleRef` representing a specific locale, or pass `NULL` to request the default system locale. You can supply the value `kUnicodeTextBreakClass` in the `opClass` parameter of the Locales Utilities functions `LocaleOperationCountLocales` and `LocaleOperationGetLocales` to obtain the locales available for finding text boundaries on the current system.

opVariant

A `LocaleOperationVariant` value identifying a text-break operation variant within the locale specified in the `locale` parameter. You can also pass 0 to request the default text-break variant for any locale. To obtain the varieties of locale-specific text-break variants that are currently available, you can supply the value `kUnicodeTextBreakClass` in the `opClass` parameter of the Locales Utilities functions `LocaleOperationCountLocales` and `LocaleOperationGetLocales`.

breakTypes

A `UCTextBreakType` value specifying each type of text boundary that the text-break locator should support. You do not need to create a text-break locator solely for the `BreakChar` type; it is locale-independent and automatically supported by the function `UCFindTextBreak` (page 17). If `BreakChar` is the only type for which you call the `UCCreateTextBreakLocator` function, on return the `breakRef` parameter returns a `NULL` value (with no error).

breakRef

A pointer to a value of type `TextBreakLocatorRef`. On return, the `TextBreakLocatorRef` value contains a valid reference to a new text-break locator object.

Return Value

A result code. The function can return memory errors and `paramErr` (for example, if the `breakRef` parameter is `NULL` or if invalid bits are set in the `breakTypes` parameter). It can also return resource errors in Mac OS 9 and CarbonLib.

Discussion

To find boundaries in Unicode text, you must supply locale and text-break specifications to the function `UCFindTextBreak` (page 17). You provide this information by means of a text-break locator object, created via the `UCCreateTextBreakLocator` function. When finished with the text-break locator object, you should dispose of it using the function `UCDisposeTextBreakLocator` (page 16).

The `UCCreateTextBreakLocator` function creates a text-break locator object for a specified locale, a specified text-break variant within that locale, and a specified set of break types. The different types of breaks or boundaries in a line of Unicode text can include

- Boundaries of characters (treating surrogate pairs as a single character).
- Boundaries of character clusters. A cluster is a group of characters that should be treated as single text element for editing operations such as cursor movement. Typically this includes groups such as a base character followed by a sequence of combining characters, for example, a Hangul syllable represented as a sequence of conjoining jamo characters or an Indic consonant cluster.
- Boundaries of words. This can be used to determine what to highlight as the result of a double-click.
- Potential line break locations.

Special Considerations

This function can move memory.

Availability

Available in CarbonLib 1.0 and later when running Mac OS 9 or later.

Available in Mac OS X 10.0 and later.

Declared In

`UnicodeUtilities.h`

UCDisposeCollator

Disposes a collator object.

```
OSStatus UCDisposeCollator (
   CollatorRef *collatorRef
);
```

Parameters

`collatorRef`

A reference to a valid collator object. The `UCDisposeCollator` function sets `*collatorRef` to `NULL`.

Return Value

A result code.

Discussion

To perform Unicode string comparison, you must supply locale and collation specifications to a collation function such as `UCCompareText` (page 9). You provide this information by means of a collator object, created via the function `UCCreateCollator` (page 13). When finished with the collator object, you should dispose of it using the function `UCDisposeCollator`.

Availability

Available in CarbonLib 1.0 and later when running Mac OS 8.6 or later.

Available in Mac OS X 10.0 and later.

Declared In

`UnicodeUtilities.h`

UCDisposeTextBreakLocator

Disposes a text-break locator object.

```
OSStatus UCDisposeTextBreakLocator (
   TextBreakLocatorRef *breakRef
);
```

Parameters

`breakRef`

A reference to a valid text-break locator object. The `UCDisposeTextBreakLocator` function sets `*breakRef` to `NULL`.

Return Value

A result code. This function can return `paramErr`, for example, if the `breakRef` parameter is `NULL`.

Discussion
To find boundaries in Unicode text, you must supply locale and text-break specifications to the function `UCFindTextBreak` (page 17). You provide this information by means of a text-break locator object, created via the function `UCCreateTextBreakLocator` (page 14). When finished with the text-break locator object, you should dispose of it using the function `UCDisposeTextBreakLocator`.

Availability
Available in CarbonLib 1.0 and later when running Mac OS 9 or later.
Available in Mac OS X 10.0 and later.

Declared In
`UnicodeUtilities.h`

UCFindTextBreak

Uses locale-specific text-break information to find boundaries in Unicode text.

```
OSStatus UCFindTextBreak (
   TextBreakLocatorRef breakRef,
   UCTextBreakType breakType,
   UCTextBreakOptions options,
   const UniChar *textPtr,
   UniCharCount textLength,
   UniCharArrayOffset startOffset,
   UniCharArrayOffset *breakOffset
);
```

Parameters

breakRef
A valid reference to a text-break locator object. If the type of boundary specified by the `breakType` parameter is `BreakChar`, you can pass `NULL`. You use the function `UCCreateTextBreakLocator` (page 14) to obtain a text-break locator object reference. If non-`NULL`, the text-break locator object must support the type of boundary specified in the `breakType` parameter.

breakType
A value of type `UCTextBreakType`, with exactly one bit set to specify a single type of boundary to be located. Since support for finding character boundaries is locale-independent and built into the `UCFindTextBreak` function, if you specify `BreakChar` as the type of boundary, then the `breakRef` parameter is ignored and may be `NULL`.

options
A `UCTextBreakOptions` value to specify the operation of the `UCFindTextBreak` function. You can use text-break locator options to control some location-independent aspects of a text-boundary search. Note that if you do not specify any `UCTextBreakOptions` values, `UCFindTextBreak` searches forward, but assumes that the `startOffset` value refers to the character preceding the offset rather than the one at the offset. This can result in `UCFindTextBreak` returning an offset that is equal to the start offset.

textPtr
A pointer to the initial character of the Unicode string to search.

textLength
The total count of Unicode characters in the string to search.

startOffset

A UniCharArrayOffset value specifying the offset from which UCFindTextBreak is to begin searching for the next text boundary of the type specified in the breakType parameter. If startOffset == 0 then kUCTextBreakLeadingEdgeMask must be set in the options parameter; if startOffset == textLength then kUCTextBreakLeadingEdgeMask must not be set.

breakOffset

A pointer to a UniCharArrayOffset value. On return, the value pointed to by the breakOffset parameter is set to the offset of the text boundary located by UCFindTextBreak. In normal usage (when exactly one of kUCTextBreakLeadingEdgeMask and kUCTextBreakGoBackwardsMask are set), the result returned in breakOffset is not equal to that supplied in the startOffset parameter unless an error occurs (and the function result is other than noErr). However, when kUCTextBreakLeadingEdgeMask and kUCTextBreakGoBackwardsMask are both set or both clear, the result produced in breakOffset can be equal to the value of startOffset.

Return Value

A result code. The text-break locator referenced by the breakRef parameter must support the type of boundary specified in the breakType parameter; otherwise, the function returns kUCTextBreakLocatorMissingType.

Discussion

The UCFindTextBreak function starts from a specified offset in a text buffer, and then proceeds forward or backward (as requested) until it finds the next text boundary of a particular locale-specific type, using a given set of options. The different types of breaks or boundaries in a line of Unicode text can include

- Boundaries of characters (treating surrogate pairs as a single character).
- Boundaries of character clusters. A cluster is a group of characters that should be treated as single text element for editing operations such as cursor movement. Typically this includes groups such as a base character followed by a sequence of combining characters, for example, a Hangul syllable represented as a sequence of conjoining jamo characters or an Indic consonant cluster.
- Boundaries of words. This can be used to determine what to highlight as the result of a double-click.
- Potential line break locations.

Finding boundaries of characters is a locale-independent operation, and support for it is built directly into the UCFindTextBreak function. If that is the only type of text boundary that you wish to locate, it is not necessary to call UCCreateTextBreakLocator and create a text-break locator object.

When finished with the text-break locator object, dispose it using the function UCDisposeTextBreakLocator (page 16).

Availability

Available in CarbonLib 1.0 and later when running Mac OS 9 or later.

Available in Mac OS X 10.0 and later.

Declared In

UnicodeUtilities.h

UCGetCollationKey

Uses locale-specific collation information to generate a collation key for a Unicode string.

```
OSStatus UCGetCollationKey (
   CollatorRef collatorRef,
   const UniChar *textPtr,
   UniCharCount textLength,
   ItemCount maxKeySize,
   ItemCount *actualKeySize,
   UCCollationValue collationKey[]
);
```

Parameters

collatorRef

A valid reference to a collator object; `NULL` is not allowed. You can use the function `UCCreateCollator` (page 13) to obtain a collator reference.

textPtr

A pointer to the Unicode string (a `UniChar` array) for which to generate a collation key.

textLength

The total count of Unicode characters in the string referenced by the `textPtr` parameter.

maxKeySize

An `ItemCount` value specifying the length of the `UCCollationValue` array passed in the `collationKey` parameter. This dimension should typically be at least `5*textLength`, as the byte length of a collation key is typically more than 16 times the number of Unicode characters in the string.

actualKeySize

On return, the actual length of the `UCCollationValue` array returned in the `collationKey` parameter.

collationKey

An array of `UCCollationValue` values. On return, the array contains the new collation key. The collation key consists of a sequence of primary weights for all of the collation text elements in the string, followed by a separator and a sequence of secondary weights for all of the text elements in the string, and so on for several levels of significance. The separator is usually 0; however, 1 is used as the separator at the boundary between levels that are significant and levels that are insignificant for the options you supply in the collator object.

Return Value

A result code. The function can return `paramErr`, for example, if the parameters `collatorRef`, `textPtr`, `actualKeySize`, or `collationKey` are `NULL`. It can also return memory errors. If `maxKeySize` is too small for the collation key, the function returns `kUCOutputBufferTooSmall`.

Discussion

If you want to compare the same strings several times, as when sorting a list of strings, it may be most efficient for you to derive a collation key for each string and then compare the collation keys. A collation key is a transformation of the string that depends on the collator object (that is, it depends on the locale, the collation variant if any, and the collation options).

Collation keys that are generated using the same collator object—but for different strings—can quickly be compared with each other, without further reference to the collator object or collation tables. The disadvantage is that the collation keys may be rather large. After you use the `UCGetCollationKey` function to create a collation key from a given string and collator object, you can call the function `UCCompareCollationKeys` (page 8) to compare two collation keys that were generated with the same collator object.

If you are comparing different strings, it may be more efficient for you to call the function `UCCompareText` (page 9) multiple times using the same collator object.

Note that collation keys should be used only in a runtime context. They should not be stored in a persistent state (such as to disk) because the format of a collation key could change in the future.

Special Considerations

This function can move memory.

Availability

Available in CarbonLib 1.0 and later when running Mac OS 8.6 or later.

Available in Mac OS X 10.0 and later.

Declared In

`UnicodeUtilities.h`

UCKeyTranslate

Converts a combination of a virtual key code, a modifier key state, and a dead-key state into a string of one or more Unicode characters.

```
OSStatus UCKeyTranslate (
   const UCKeyboardLayout *keyLayoutPtr,
   UInt16 virtualKeyCode,
   UInt16 keyAction,
   UInt32 modifierKeyState,
   UInt32 keyboardType,
   OptionBits keyTranslateOptions,
   UInt32 *deadKeyState,
   UniCharCount maxStringLength,
   UniCharCount *actualStringLength,
   UniChar unicodeString[]
);
```

Parameters

keyLayoutPtr

A pointer to the first element in a resource of type `'uchr'`. Pass a pointer to the `'uchr'` resource that you wish the `UCKeyTranslate` function to use when converting the virtual key code to a Unicode character. The resource handle associated with this pointer need not be locked, since the `UCKeyTranslate` function does not move memory.

virtualKeyCode

An unsigned 16-bit integer. Pass a value specifying the virtual key code that is to be translated. For ADB keyboards, virtual key codes are in the range from 0 to 127.

keyAction

An unsigned 16-bit integer. Pass a value specifying the current key action. See "Key Actions" (page 36) for descriptions of possible values.

modifierKeyState

An unsigned 32-bit integer. Pass a bit mask indicating the current state of various modifier keys. You can obtain this value from the modifiers field of the event record as follows:

```
modifierKeyState = ((EventRecord.modifiers) >> 8) & 0xFF;
```

keyboardType

An unsigned 32-bit integer. Pass a value specifying the physical keyboard type (that is, the keyboard shape shown by Key Caps). You can call the function `LMGetKbdType` for this value.

keyTranslateOptions

A bit mask of options for controlling the UCKeyTranslate function. See "Key Translation Options Flag" (page 39) and "Key Translation Options Mask" (page 40) for descriptions of possible values.

deadKeyState

A pointer to an unsigned 32-bit value, initialized to zero. The UCKeyTranslate function uses this value to store private information about the current dead key state.

maxStringLength

A value of type UniCharCount. Pass the number of 16-bit Unicode characters that are contained in the buffer passed in the unicodeString parameter. This may be a value of up to 255, although it would be rare to get more than 4 characters.

actualStringLength

A pointer to a value of type UniCharCount. On return this value contains the actual number of Unicode characters placed into the buffer passed in the unicodeString parameter.

unicodeString

An array of values of type UniChar. Pass a pointer to the buffer whose sized is specified in the maxStringLength parameter. On return, the buffer contains a string of Unicode characters resulting from the virtual key code being handled. The number of characters in this string is less than or equal to the value specified in the maxStringLength parameter.

Return Value

A result code. If you pass NULL in the keyLayoutPtr parameter, UCKeyTranslate returns paramErr. The UCKeyTranslate function also returns paramErr for an invalid 'uchr' resource format or for invalid virtualKeyCode or keyAction values, as well as for NULL pointers to output values.The result kUCOutputBufferTooSmall (-25340) is returned for an output string length greater than maxStringLength.

Discussion

The UCKeyTranslate function uses the data in a Unicode keyboard-layout ('uchr') resource to map a combination of virtual key code and modifier key state to a sequence of up to 255 Unicode characters. This mapping process depends on, and may update, a dead key state; the UCKeyTranslate function and the 'uchr' resource support multiple dead keys. The mapping may also depend on the specific type of key action and the type of physical keyboard being used. The UCKeyTranslate function supports non-ADB keyboards, an extensible set of modifier keys, and other possible extensions.

In most cases, your application does not need to call the UCKeyTranslate function, since the Text Services Manager automatically calls it on your behalf to handle input from a Unicode keyboard layout. However, there may be some circumstances in which your application should call UCKeyTranslate. For example, your application may need to determine what character(s) would have been generated for the virtual key code in the current key-down event if a different modifier-and-key combination had been used.

The basic process by which UCKeyTranslate uses the 'uchr' resource to translate virtual key codes into Unicode characters is detailed in the following steps:

1. The bit pattern specifying the modifier key state is mapped by the UCKeyModifiersToTableNum structure to a table number.

2. The table number maps to an offset within a UCKeyToCharTableIndex structure that refers to the actual key-code-to-character tables.

3. The key-code-to-character tables map the virtual key code to UCKeyOutput values, for which there are two possibilities:

- If bits 15 and 14 of the `UCKeyOutput` value are 01, the `UCKeyOutput` value is an index into the offsets contained in a `UCKeyStateRecordsIndex` structure. If this occurs, the mapping process for the virtual key code continues on to the next step
- Otherwise, the `UCKeyOutput` value produces one or more Unicode characters, either directly or via reference to a `UCKeySequenceDataIndex` structure. This ends the mapping process for a given virtual key code.

4. The offsets in a `UCKeyStateRecordsIndex` structure refer to `UCKeyStateRecord` dead-key state records.

5. The dead-key state records map from the current dead-key state to one or more Unicode characters to be output or the following dead-key state (if any). The mapping process for a given virtual key code may end with the dead-key state record or, if there is no dead-key state record entry for the key code, with a default state terminator, as specified in the resource's `UCKeyStateTerminators` table.

Availability
Available in CarbonLib 1.0 and later when running Mac OS 8.5 or later.
Available in Mac OS X 10.0 and later.

Declared In
`UnicodeUtilities.h`

Data Types

CollatorRef

Refers to an opaque object that encapsulates locale and collation information for the purpose of performing Unicode string comparison.

```
typedef struct OpaqueCollatorRef * CollatorRef;
```

Discussion
You can obtain a `CollatorRef` value from the function `UCCreateCollator` (page 13).

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

Declared In
`UnicodeUtilities.h`

TextBreakLocatorRef

Refers to an opaque object that encapsulates locale and text-break information for the purpose of finding boundaries in Unicode text.

```
typedef struct OpaqueTextBreakLocatorRef * TextBreakLocatorRef;
```

Discussion
You can obtain a `TextBreakLocatorRef` value from the function `UCCreateTextBreakLocator` (page 14).

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

Declared In
`UnicodeUtilities.h`

UCCollationValue

Specifies a Unicode collation key.

```
typedef UInt32 UCCollationValue;
```

Discussion
Collation keys consist of an array of `UCCollationValue` values. The collation key consists of a sequence of primary weights for all of the collation text elements in the string, followed by a separator and a sequence of secondary weights for all of the text elements in the string, and so on for several levels of significance. The separator is usually 0; however, 1 is used as the separator at the boundary between levels that are significant and levels that are insignificant for the options you supply in the collator object. You can obtain a collation key with the function `UCGetCollationKey` (page 18).

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

Declared In
`UnicodeUtilities.h`

UCKeyboardLayout

Provides header data for a `'uchr'` resource.

```
struct UCKeyboardLayout {
    UInt16 keyLayoutHeaderFormat;
    UInt16 keyLayoutDataVersion;
    ByteOffset keyLayoutFeatureInfoOffset;
    ItemCount keyboardTypeCount;
    UCKeyboardTypeHeader keyboardTypeList[1];
};
typedef struct UCKeyboardLayout UCKeyboardLayout;
```

Fields

`keyLayoutHeaderFormat`
An unsigned 16-bit integer identifying the format of the structure. Set to `kUCLayoutHeaderFormat`.

`keyLayoutDataVersion`
An unsigned 16-bit integer identifying the version of the data in the resource, in binary code decimal format. For example, 0x0100 would equal version 1.0.

`keyLayoutFeatureInfoOffset`

An unsigned 32-bit integer providing an offset to a structure of type `UCKeyLayoutFeatureInfo` (page 26), if such is used in the resource. May be 0 if no `UCKeyLayoutFeatureInfo` table is included in the resource.

`keyboardTypeCount`

An unsigned 32-bit integer specifying the number of `UCKeyboardTypeHeader` structures in the `keyboardTypeList[]` field's array.

`keyboardTypeList`

A variable-length array containing structures of type `UCKeyboardTypeHeader`. Each `UCKeyboardTypeHeader` entry specifies a range of physical keyboard types and contains offsets to each of the key mapping sections to be used for that range of keyboard types.

Discussion

The Unicode keyboard-layout (`'uchr'`) resource contains the data necessary to map virtual key codes to Unicode character codes for a given keyboard layout. The `'uchr'` format consists of a header information section and five key mapping data sections. The `UCKeyboardLayout` type is used in the `'uchr'` resource header. It specifies version and format information, offsets to the various subtables, and an array of `UCKeyboardTypeHeader` entries.

You should use low-ASCII (0 - 0x7F) only for the KCHR/uchr resource names and you should use Unicode in the Info.plist file when you specify strings for the user-interface (UI).

Availability

Available in Mac OS X v10.0 and later.

Declared In

`UnicodeUtilities.h`

UCKeyboardTypeHeader

Specifies a range of physical keyboard types in a `'uchr'` resource.

```
struct UCKeyboardTypeHeader {
    UInt32 keyboardTypeFirst;
    UInt32 keyboardTypeLast;
    ByteOffset keyModifiersToTableNumOffset;
    ByteOffset keyToCharTableIndexOffset;
    ByteOffset keyStateRecordsIndexOffset;
    ByteOffset keyStateTerminatorsOffset;
    ByteOffset keySequenceDataIndexOffset;
};
typedef struct UCKeyboardTypeHeader UCKeyboardTypeHeader;
```

Fields

`keyboardTypeFirst`

An unsigned 32-bit integer specifying the first keyboard type in this entry. For the initial entry (that is, the default entry) in an array of `UCKeyboardTypeHeader` structures, you should set this value to 0. The initial `UCKeyboardTypeHeader` entry is used if the keyboard type passed to the function `UCKeyTranslate` (page 20) does not match any other entry, that is, if it is not within the range of values specified by `keyboardTypeFirst` and `keyboardTypeLast` for any entry.

`keyboardTypeLast`
: An unsigned 32-bit integer specifying the last keyboard type in this entry. For the initial entry (that is, the default entry) in an array of `UCKeyboardTypeHeader` structures, you should set this value to 0.

`keyModifiersToTableNumOffset`
: An unsigned 32-bit integer providing an offset to a structure of type `UCKeyModifiersToTableNum` (page 27). The `'uchr'` resource requires a `UCKeyModifiersToTableNum` structure, therefore this field must contain a non-zero value.

`keyToCharTableIndexOffset`
: An unsigned 32-bit integer providing an offset to a structure of type `UCKeyToCharTableIndex` (page 34). The `'uchr'` resource requires a `UCKeyToCharTableIndex` structure, therefore this field must contain a non-zero value.

`keyStateRecordsIndexOffset`
: An unsigned 32-bit integer providing an offset to a structure of type `UCKeyStateRecordsIndex` (page 32), if such is used in the resource. This value may be 0 if no dead-key state records are included in the resource.

`keyStateTerminatorsOffset`
: An unsigned 32-bit integer providing an offset to a structure of type `UCKeyStateTerminators` (page 33), if such is used in the resource. This value may be 0 if no dead-key state terminators are included in the resource.

`keySequenceDataIndexOffset`
: An unsigned 32-bit integer providing an offset to a structure of type `UCKeySequenceDataIndex` (page 28), if such is used in the resource. This value may be 0 if no character key sequences are included in the resource.

Discussion
The `UCKeyboardTypeHeader` type is used in a structure of type `UCKeyboardLayout` (page 23) to specify a range of physical keyboard types and contains offsets to each of the key mapping sections to be used for that range of keyboard types. Typically, you use an array of `UCKeyboardTypeHeader` structures, of which the first entry in the array is the default and will be used if the keyboard type does not fall within the range for any other entry. See `UCKeyboardLayout` (page 23) for a further discussion of the context for use of the `UCKeyboardTypeHeader` type.

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

Declared In
`UnicodeUtilities.h`

UCKeyCharSeq

Specifies the output of a dead-key state in a `'uchr'` resource.

```
typedef UInt16 UCKeyCharSeq;
```

Discussion
The Unicode keyboard-layout (`'uchr'`) resource contains the data necessary to map virtual key codes to Unicode character codes for a given keyboard layout. The `'uchr'` format consists of a header information section and five key mapping data sections. The `UCKeyCharSeq` type is a 16-bit value used in the third key mapping section of the `'uchr'` resource to specify the output of a dead-key state.

Specifically, the dead-key state record—a structure of type `UCKeyStateRecord` (page 31) —uses a `UCKeyCharSeq` value to contain the character output that results from the resolution of a given dead-key state. You can use a `UCKeyCharSeq` value in a dead-key state record to represent either an index to a Unicode character sequence or a single Unicode character. The `UCKeyCharSeq` type is similar to the type `UCKeyOutput` (page 27) , but does not itself support indices into dead-key state records.

The interpretation of `UCKeyCharSeq` depends on bits 15 and 14.

If they are 10 (that is, for values in the range of 0x8000–0xBFFF), then bits 0–13 are an index into the `charSequenceOffsets`[field of a structure of type `UCKeySequenceDataIndex` (page 28) , which contains offsets to a separate resource-wide list of Unicode character sequences. If a `UCKeySequenceDataIndex` structure is not present in the resource or the index is beyond the end of the list, then the entire value (that is, bits 0–15) is a single Unicode character to emit. Otherwise (for values in the range of 0x0000–0x7FFF and 0xC000–0xFFFD), bits 0–15 are a single Unicode character, with the exception that a value of 0xFFFE–0xFFFF means no character output (these are invalid Unicode codes).

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

Declared In
`UnicodeUtilities.h`

UCKeyLayoutFeatureInfo

Specifies the longest possible output string to be produced by the current `'uchr'` resource.

```
struct UCKeyLayoutFeatureInfo {
    UInt16 keyLayoutFeatureInfoFormat;
    UInt16 reserved;
    UniCharCount maxOutputStringLength;
};
typedef struct UCKeyLayoutFeatureInfo UCKeyLayoutFeatureInfo;
```

Fields

`keyLayoutFeatureInfoFormat`
: An unsigned 16-bit integer identifying the format of the `UCKeyLayoutFeatureInfo` structure. Set to `kUCKeyLayoutFeatureInfoFormat`.

`reserved`
: Reserved. Set to 0.

`maxOutputStringLength`
: An unsigned 32-bit integer specifying the longest possible output string of Unicode characters to be produced by this `'uchr'` resource.

Discussion
The Unicode keyboard-layout (`'uchr'`) resource contains the data necessary to map virtual key codes to Unicode character codes for a given keyboard layout. The `'uchr'` format consists of a header information section and five key mapping data sections. The `UCKeyLayoutFeatureInfo` type is used in the header section of the `'uchr'` resource.

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

Declared In
`UnicodeUtilities.h`

UCKeyModifiersToTableNum

Maps a modifier key combination to a particular key-code-to-character table number in a 'uchr' resource.

```
struct UCKeyModifiersToTableNum {
    UInt16 keyModifiersToTableNumFormat;
    UInt16 defaultTableNum;
    ItemCount modifiersCount;
    UInt8 tableNum[1];
};
typedef struct UCKeyModifiersToTableNum UCKeyModifiersToTableNum;
```

Fields

keyModifiersToTableNumFormat
: An unsigned 16-bit integer identifying the format of the UCKeyModifiersToTableNum structure. Set to kUCKeyModifiersToTableNumFormat.

defaultTableNum
: An unsigned 16-bit integer identifying the table number to use for modifier combinations that are outside of the range included in the tableNum field.

modifiersCount
: An unsigned 32-bit integer specifying the range of modifier bit combinations for which there are entries in the tableNum[] field.

tableNum
: An array of unsigned 8-bit integers that map modifier bit combinations to table numbers. These values are indexes into the keyToCharTableOffsets array in a UCKeyToCharTableIndex (page 34)structure; these, in turn, are offsets to the actual key-code-to character tables, which follow the UCKeyToCharTableIndex structure in the 'uchr' resource.

Discussion

The Unicode keyboard-layout ('uchr') resource contains the data necessary to map virtual key codes to Unicode character codes for a given keyboard layout. The 'uchr' format consists of a header information section and five key mapping data sections. The UCKeyModifiersToTableNum type is used in the first key mapping section of the 'uchr' resource. It maps a modifier key combination to a particular key-code-to-character table number.

Availability

Available in Mac OS X v10.0 and later.

Declared In

UnicodeUtilities.h

UCKeyOutput

Specifies values in key-code-to-character tables in a 'uchr' resource.

```
typedef UInt16 UCKeyOutput;
```

Discussion

The Unicode keyboard-layout ('uchr') resource contains the data necessary to map virtual key codes to Unicode character codes for a given keyboard layout. The 'uchr' format consists of a header information section and five key mapping data sections. The UCKeyOutput type is a 16-bit value used in the second key mapping section of a 'uchr' resource to specify values in key-code-to-character tables.

You use a `UCKeyOutput` value in a key-code-to-character table to represent one of the following: an index to a dead-key state record, an index to a Unicode character sequence, or a single Unicode character.

The interpretation of a `UCKeyOutput` value depends on bits 15 and 14.

If they are 01 (that is, for values in the range of 0x4000-0x7FFF), then bits 0-13 are an index into the `keyStateRecordOffsets` field of a structure of type `UCKeyStateRecordsIndex` (page 32) , which contains offsets to a separate resource-wide list of dead-key state records.

If they are 10 (that is, for values in the range of 0x8000-0xBFFF), then bits 0-13 are an index into the `charSequenceOffsets` field of a structure of type `UCKeySequenceDataIndex` (page 28) , which contains offsets to a separate resource-wide list of Unicode character sequences. If a `UCKeySequenceDataIndex` structure is not present in the resource or the index is beyond the end of the list, then the entire value (that is, bits 0-15) is a single Unicode character to emit.

Otherwise (for values in the range of 0x0000-0x3FFF and 0xC000-0xFFFD), bits 0-15 are a single Unicode character, with the exception that a value of 0xFFFE-0xFFFF means no character output (these are invalid Unicode codes).

Most single Unicode characters that are likely to be generated by direct keyboard input are in the range 0x0000-0x33FF or 0xE000-0xFFFD, and so are covered by the single-character cases above. Characters outside this range can still be generated by direct keyboard input, in which case they must be represented as 1-character sequences. The fifth key mapping section of the `'uchr'` resource, introduced by the `UCKeySequenceDataIndex` type, provides for this option.

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

Declared In
`UnicodeUtilities.h`

UCKeySequenceDataIndex

Contains offsets to a list of character sequences for a `'uchr'` resource.

```
struct UCKeySequenceDataIndex {
    UInt16 keySequenceDataIndexFormat;
    UInt16 charSequenceCount;
    UInt16 charSequenceOffsets[1];
};
typedef struct UCKeySequenceDataIndex UCKeySequenceDataIndex;
```

Fields

`keySequenceDataIndexFormat`
: An unsigned 16-bit integer identifying the format of the `UCKeySequenceDataIndex` structure. Set to `kUCKeySequenceDataIndexFormat`.

`charSequenceCount`
: An unsigned 16-bit integer specifying the number of Unicode character sequences that follow the end of the `UCKeySequenceDataIndex` structure.

`charSequenceOffsets`

An array of offsets from the beginning of the `UCKeySequenceDataIndex` structure to the Unicode character sequences that follow it. Because a given offset indicates both the beginning of a new character sequence and the end of the sequence that precedes it, the length of each sequence is determined by the difference between the offset to that sequence and the value of the next offset in the array. The array contains one more entry than the number of character sequences; the final entry is the offset to the end of the final character sequence.

Discussion

The Unicode keyboard-layout (`'uchr'`) resource contains the data necessary to map virtual key codes to Unicode character codes for a given keyboard layout. The `'uchr'` format consists of a header information section and five key mapping data sections. The `UCKeySequenceDataIndex` type is used in the fifth key mapping section of the `'uchr'` resource.

The `UCKeySequenceDataIndex` structure contains offsets to a list of character sequences for the `'uchr'` resource. This permits a single keypress to generate a sequence of characters, or to generate a single character outside the range that can be represented directly by values of type `UCKeyOutput` (page 27) or `UCKeyCharSeq` (page 25).

Availability

Available in Mac OS X v10.0 and later.

Declared In

`UnicodeUtilities.h`

UCKeyStateEntryRange

Maps from a dead-key state to either the resultant Unicode character(s) or the new dead key state produced when the current state is terminated by a given character key for a `'uchr'` resource.

```
struct UCKeyStateEntryRange {
    UInt16 curStateStart;
    UInt8 curStateRange;
    UInt8 deltaMultiplier;
    UCKeyCharSeq charData;
    UInt16 nextState;
};
typedef struct UCKeyStateEntryRange UCKeyStateEntryRange;
```

Fields

`curStateStart`

An unsigned 16-bit integer specifying the beginning of a given dead-key state range.

`curStateRange`

An unsigned 8-bit integer specifying the number of entries in a given dead-key state range.

`deltaMultiplier`

An unsigned 8-bit integer.

`charData`

A value of type `UCKeyCharSeq`. This base character value is used to determine the actual Unicode character(s) produced when a given dead-key state terminates.

`nextState`

An unsigned 16-bit integer. This base dead-key state value is used to determine the following dead-key state, if any.

Discussion
The `UCKeyStateEntryRange` type is used in the `stateEntryData[]` field of a structure of type `UCKeyStateRecord` (page 31). You should use the `UCKeyStateEntryRange` format for complex (multiple) dead-key states.

For each virtual key code, an entry in its dead-key state record maps from the current dead-key state to the Unicode character(s) produced or to the next dead-key state, as follows.

If the current dead-key state is within a valid dead-key state range for the given input character—that is, if its value is greater than or equal to `curStateStart` and less than or equal to `curStateStart + curStateRange`—then

- If the base `charData` value for the given dead-key state range is in the range of valid Unicode characters, a character is produced and the dead-key state may be terminated.

and/or

- If the base `nextState` value is not 0, a new dead-key state is produced.

In the first case, the output character is determined as follows: The base `charData` value is incremented by the resulting product of (the difference between the current state and the start of that state's range) and (a multiplier). That is:

```
charData += (curState - curStateStart) * deltaMultiplier
```

Similarly, in the second case, the resulting dead-key state, which is the new `curState` value, is determined as follows: The base `nextState` value is incremented by the resulting product of (the difference between the current state and the start of that state's range) and (a multiplier). That is:

```
nextState += (curState - curStateStart) * deltaMultiplier
```

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

Declared In
`UnicodeUtilities.h`

UCKeyStateEntryTerminal

Maps from a dead-key state to the Unicode character(s) produced when that state is terminated by a given character key for a `'uchr'` resource.

```
struct UCKeyStateEntryTerminal {
    UInt16 curState;
    UCKeyCharSeq charData;
};
typedef struct UCKeyStateEntryTerminal UCKeyStateEntryTerminal;
```

Fields
`curState`
An unsigned 16-bit integer specifying the current dead-key state.

`charData`

A value of type `UCKeyCharSeq` specifying the Unicode character(s) produced when a given character key is pressed.

Discussion

The `UCKeyStateEntryTerminal` type is used in the `stateEntryData[]` field of a structure of type `UCKeyStateRecord` (page 31). You should use the `UCKeyStateEntryTerminal` format for simple dead-key states that are terminated by a single keystroke, as in the U.S. keyboard layout. Each entry maps from the current dead-key state to the Unicode character(s) produced when a given character key is pressed that terminates the dead-key state.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`UnicodeUtilities.h`

UCKeyStateRecord

Determines dead-key state transitions in a `'uchr'` resource.

```
struct UCKeyStateRecord {
    UCKeyCharSeq stateZeroCharData;
    UInt16 stateZeroNextState;
    UInt16 stateEntryCount;
    UInt16 stateEntryFormat;
    UInt32 stateEntryData[1];
};
typedef struct UCKeyStateRecord UCKeyStateRecord;
```

Fields

`stateZeroCharData`

A value of type `UCKeyCharSeq` specifying the Unicode character(s) produced from a given key code while no dead-key state is in effect.

`stateZeroNextState`

An unsigned 16-bit integer specifying the dead-key state produced from a given key code when no previous dead-key state is in effect. If the `UCKeyStateRecord` structure does not initiate a dead-key state (but only provides terminators for other dead-key states), this will be 0. A non-zero value specifies the resulting new dead-key state and refers to the current state entry within the `stateEntryData[]` field for the following dead-key state record that is applied.

`stateEntryCount`

An unsigned 16-bit integer specifying the number of elements in the `stateEntryData` field's array for a given dead-key state record.

`stateEntryFormat`

An unsigned 16-bit integer specifying the format of the data in the `stateEntryData` field's array. This should be 0 if the `stateEntryCount` field is set to 0. Currently available values are `kUCKeyStateEntryTerminalFormat` and `kUCKeyStateEntryRangeFormat`; see "Key State Entry Formats" (page 39) for descriptions of these values.

`stateEntryData`

An array of dead-key state entries, whose size depends on their format, but which will always be a multiple of 4 bytes. Each entry maps from the current dead-key state to the Unicode character(s) that result when a given character key is pressed or to the next dead-key state, if any. The format of the entry is specified by the `stateEntryFormat` field to be either that of type `UCKeyStateEntryTerminal` (page 30) or `UCKeyStateEntryRange` (page 29).

Discussion

The Unicode keyboard-layout (`'uchr'`) resource contains the data necessary to map virtual key codes to Unicode character codes for a given keyboard layout. The `'uchr'` format consists of a header information section and five key mapping data sections. The `UCKeyStateRecord` type is used in the third key mapping section of the `'uchr'` resource to determine dead-key state transitions. The `UCKeyStateRecord` structure permits complex dead-key state processing, such as a series of transitions from one dead-key state directly into another, in which each transition can emit a sequence of one or more Unicode characters.

Any modifier key combination which initiates a dead-key state or which is a valid terminator of a dead-key state refers to one of these records via the `UCKeyOutput` (page 27) values in key-code-to-character tables. A `UCKeyOutput` value may index the offsets contained in a `UCKeyStateRecordsIndex` (page 32) structure, which in turn refers to the actual dead-key state records.

Each `UCKeyStateRecord` structure maps from the current dead-key state to the character data to be output or the following dead-key state (if any), as follows:

- If the current dead-key state is zero (that is, there are no dead keys in effect) the value in `stateZeroCharData` is output and the state is set to the value in `stateZeroNextState` (this can be used to initiate a dead-key state).
- If the current dead-key state is non-zero and there is an entry for the state in `stateEntryData`, then the corresponding value in `stateEntryData.charData` is output. The next state is then set to either a `kUCKeyStateEntryTerminalFormat` or a `kUCKeyStateEntryRangeFormat` value; in either case, if the next dead-key state is 0, this implements a valid dead-key state terminator.
- If the current dead-key state is non-zero, and there is no entry for the state in `stateEntryData`, the default state terminator is output from the `'uchr'` resource's `UCKeyStateTerminators` (page 33) table for the current state (or nothing may be output, if there is no `UCKeyStateTerminators` table or it has no entry for the current state). Then the value in `stateZeroCharData` is output, and the state is set to the value in `stateZeroNextState`.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`UnicodeUtilities.h`

UCKeyStateRecordsIndex

Provides a count of, and offsets to, dead-key state records in a `'uchr'` resource.

```
struct UCKeyStateRecordsIndex {
    UInt16 keyStateRecordsIndexFormat;
    UInt16 keyStateRecordCount;
    ByteOffset keyStateRecordOffsets[1];
};
typedef struct UCKeyStateRecordsIndex UCKeyStateRecordsIndex;
```

Fields

`keyStateRecordsIndexFormat`

An unsigned 16-bit integer identifying the format of the `UCKeyStateRecordsIndex` structure. Set to `kUCKeyStateRecordsIndexFormat`.

`keyStateRecordCount`

An unsigned 16-bit integer specifying the number of dead-key state records that are included in the resource.

`keyStateRecordOffsets`

An array of offsets from the beginning of the resource to each of the `UCKeyStateRecord` values that follow this structure in the `'uchr'` resource.

Discussion

The Unicode keyboard-layout (`'uchr'`) resource contains the data necessary to map virtual key codes to Unicode character codes for a given keyboard layout. The `'uchr'` format consists of a header information section and five key mapping data sections. The `UCKeyStateRecordsIndex` type is used in the third key mapping section of the `'uchr'` resource.

The `UCKeyStateRecordsIndex` structure is an index to dead-key state records of type `UCKeyStateRecord` (page 31). Any keycode-modifier combination which initiates a dead-key state or which is a valid terminator of a dead-key state refers to one of these records, via the `UCKeyStateRecordsIndex` structure.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`UnicodeUtilities.h`

UCKeyStateTerminators

Lists the default terminators for each dead-key state handled by a `'uchr'` resource.

```
struct UCKeyStateTerminators {
    UInt16 keyStateTerminatorsFormat;
    UInt16 keyStateTerminatorCount;
    UCKeyCharSeq keyStateTerminators[1];
};
typedef struct UCKeyStateTerminators UCKeyStateTerminators;
```

Fields

`keyStateTerminatorsFormat`

An unsigned 16-bit integer identifying the format of the `UCKeyStateTerminators` structure. Set to `kUCKeyStateTerminatorsFormat`.

`keyStateTerminatorCount`

An unsigned 16-bit integer specifying the number of default dead-key state terminators contained in the `keyStateTerminators[]` array.

`keyStateTerminators`
: An array of default dead-key state terminators, described as values of type `UCKeyCharSeq` (page 25); the value `keyStateTerminators[0]` is the terminator for state 1, and so on.

Discussion
The Unicode keyboard-layout (`'uchr'`) resource contains the data necessary to map virtual key codes to Unicode character codes for a given keyboard layout. The `'uchr'` format consists of a header information section and five key mapping data sections. The `UCKeyStateTerminators` type is used in the fourth key mapping section of the `'uchr '` resource.

The `UCKeyStateTerminators` structure contains the list of default terminators (characters or sequences) for each dead-key state that is handled by a `'uchr'` resource. When a dead-key state is in effect but a modifier-and-key combination is typed which has no special handling for that state, the default terminator for the state is output before the modifier-and-key combination is processed. If this table is not present or does not extend far enough to have a terminator for the state, nothing is output when the state terminates.

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

Declared In
`UnicodeUtilities.h`

UCKeyToCharTableIndex

Provides a count of, and offsets to, key-code-to-character tables in a `'uchr'` resource.

```
struct UCKeyToCharTableIndex {
    UInt16 keyToCharTableIndexFormat;
    UInt16 keyToCharTableSize;
    ItemCount keyToCharTableCount;
    ByteOffset keyToCharTableOffsets[1];
};
typedef struct UCKeyToCharTableIndex UCKeyToCharTableIndex;
```

Fields
`keyToCharTableIndexFormat`
: An unsigned 16-bit integer identifying the format of the `UCKeyToCharTableIndex` structure. Set to `kUCKeyToCharTableIndexFormat`.

`keyToCharTableSize`
: An unsigned 16-bit integer specifying the number of virtual key codes supported by this resource; for ADB keyboards this is 128 (with virtual key codes ranging from 0 to 127).

`keyToCharTableCount`
: An unsigned 32-bit integer specifying the number of key-code-to-character tables, typically 6 to 12.

`keyToCharTableOffsets`
: An array of offsets from the beginning of the `'uchr'` resource to each of the `UCKeyOutput` key-code-to-character tables in the `keyToCharData[]` array that follows this structure in the resource.

Discussion
The Unicode keyboard-layout (`'uchr'`) resource contains the data necessary to map virtual key codes to Unicode character codes for a given keyboard layout. The `'uchr'` format consists of a header information section and five key mapping data sections. The `UCKeyToCharTableIndex` type is used in the second key mapping section of the `'uchr'` resource. The `UCKeyToCharTableIndex` structure precedes the list of key-code-to-character tables, each of which maps a key code to a 16-bit value of type `UCKeyOutput` (page 27).

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

Declared In
`UnicodeUtilities.h`

Constants

Fixed Ordering Scheme

Specifies to use the fixed ordering scheme.

```
enum {
    kUCCollateTypeHFSExtended = 1
};
```

Constants
`kUCCollateTypeHFSExtended`
The `kUCCollateTypeHFSExtended` ordering scheme sorts maximally decomposed Unicode according to the rules used by the HFS Extended volume format for its catalog. When this order is used, other collation options are ignored; this order is always case-insensitive (for decomposed characters) and ignores the Unicode characters 200C-200F, 202A-202E, 206A-206F, FEFF.

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

Discussion
`UCCollateOptions` is a 32-bit value. Bits 0-23 are described in "String Comparison Options" (page 41). The field consisting of bits 24-31 is used for values that specify which fixed ordering scheme to use with the function `UCCompareTextNoLocale` (page 12). Currently only one such scheme is provided.

Constants are provided for setting and testing the `UCCollateOptions` field that specifies the ordering scheme. These values are described in "Fixed Ordering Masks 1" (page 35) and "Fixed Ordering Masks 2" (page 36).

Fixed Ordering Masks 1

Set and test the `UCCollateOptions` field that specifies a fixed ordering scheme.

```
enum {
    kUCCollateTypeSourceMask = 0x000000FF,
    kUCCollateTypeShiftBits = 24
};
```

Constants
`kUCCollateTypeSourceMask`
You can use this mask, in conjunction with the `kUCCollateTypeShiftBits` constant, to obtain a value identifying a fixed ordering scheme.

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

`kUCCollateTypeShiftBits`

You can use this value, along with one of the constants described in "Fixed Ordering Scheme" (page 35), to specify a fixed ordering scheme. You can also use this value, in conjunction with the `kUCCollateTypeSourceMask` constant, to obtain a value identifying a fixed ordering scheme.

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

Discussion
You can use these constants to set or obtain a value that specifies a fixed ordering scheme. For a description of the available types of fixed ordering schemes, see "Fixed Ordering Scheme" (page 35).

For example, to specify `kUCCollateTypeHFSExtended` in the `options` parameter of the function `UCCompareTextNoLocale` (page 12) , the `kUCCollateTypeHFSExtended` value must be shifted by `kUCCollateTypeShiftBits` :

```
options = kUCCollateTypeHFSExtended kUCCollateTypeShiftBits;
```

You would obtain the ordering scheme value from the `options` parameter as follows:

```
fixedOrderType = ((options > > kUCCollateTypeShiftBits) &
kUCCollateTypeSourceMask);
```

See also "Fixed Ordering Masks 2" (page 36).

Fixed Ordering Masks 2

Test the `UCCollateOptions` field that specifies a fixed ordering scheme.

```
enum {
    kUCCollateTypeMask = kUCCollateTypeSourceMask << kUCCollateTypeShiftBits
};
```

Constants
`kUCCollateTypeMask`

You can use this mask to directly test bits 24-31 of a `UCCollateOptions` value.

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

Discussion
See also "Fixed Ordering Scheme" (page 35).

See also "Fixed Ordering Masks 1" (page 35).

Key Actions

Indicate the current key action.

```
enum {
    kUCKeyActionDown = 0,
    kUCKeyActionUp = 1,
    kUCKeyActionAutoKey = 2,
    kUCKeyActionDisplay = 3
};
```

Constants

`kUCKeyActionDown`

The user is pressing the key.

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

`kUCKeyActionUp`

The user is releasing the key.

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

`kUCKeyActionAutoKey`

The user has the key in an "auto-key" pressed state that is, the user is holding down the key for an extended period of time and is thereby generating multiple key strokes from the single key.

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

`kUCKeyActionDisplay`

The user is requesting information for key display, as in the Key Caps application.

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

Discussion

You can supply the following constants for the `keyAction` parameter of the function `UCKeyTranslate` (page 20) to indicate the current key action.

Key Format Codes

Indicate a structure format used in a `'uchr'` resource.

```
enum {
    kUCKeyLayoutHeaderFormat = 0x1002,
    kUCKeyLayoutFeatureInfoFormat = 0x2001,
    kUCKeyModifiersToTableNumFormat = 0x3001,
    kUCKeyToCharTableIndexFormat = 0x4001,
    kUCKeyStateRecordsIndexFormat = 0x5001,
    kUCKeyStateTerminatorsFormat = 0x6001,
    kUCKeySequenceDataIndexFormat = 0x7001
};
```

Constants

`kUCKeyLayoutHeaderFormat`

The format of a structure of type `UCKeyboardLayout` (page 23).

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

`kUCKeyLayoutFeatureInfoFormat`
The format of a structure of type `UCKeyLayoutFeatureInfo` (page 26).

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

`kUCKeyModifiersToTableNumFormat`
The format of a structure of type `UCKeyModifiersToTableNum` (page 27).

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

`kUCKeyToCharTableIndexFormat`
The format of a structure of type `UCKeyToCharTableIndex` (page 34).

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

`kUCKeyStateRecordsIndexFormat`
The format of a structure of type `UCKeyStateRecordsIndex` (page 32).

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

`kUCKeyStateTerminatorsFormat`
The format of a structure of type `UCKeyStateTerminators` (page 33).

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

`kUCKeySequenceDataIndexFormat`
The format of a structure of type `UCKeySequenceDataIndex` (page 28).

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

Discussion
These constants are those currently defined to be used within the various structures in a `'uchr'` resource to indicate each structure's format.

Key Output Index Masks

Test the bits in `UCKeyOutput` values.

```
enum {
    kUCKeyOutputStateIndexMask = 0x4000,
    kUCKeyOutputSequenceIndexMask = 0x8000,
    kUCKeyOutputTestForIndexMask = 0xC000,
    kUCKeyOutputGetIndexMask = 0x3FFF
};
```

Constants
`kUCKeyOutputStateIndexMask`
If the bit specified by this mask is set, the `UCKeyStateRecordsIndex` (page 32) `UCKeyOutput` value contains an index into a structure of type `UCKeyStateRecordsIndex` (page 32).

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

`kUCKeyOutputSequenceIndexMask`
: If the bit specified by this mask is set, the `UCKeyOutput` value contains an index into a structure of type `UCKeySequenceDataIndex` (page 28).

  Available in Mac OS X v10.0 and later.

  Declared in `UnicodeUtilities.h`.

`kUCKeyOutputTestForIndexMask`
: You can use this mask to test the bits (14–15) in the `UCKeyOutput` value that determine whether the value contains an index to any other structure. If both bits specified by this mask are clear, the `UCKeyOutput` value does not contain an index to any other structure.

  Available in Mac OS X v10.0 and later.

  Declared in `UnicodeUtilities.h`.

`kUCKeyOutputGetIndexMask`
: You can use this mask to test the bits (0–13) in a `UCKeyOutput` value that provide the actual index to another structure.

  Available in Mac OS X v10.0 and later.

  Declared in `UnicodeUtilities.h`.

Discussion
You can use these masks to test the bits in `UCKeyOutput` values.

Key State Entry Formats

Indicate the format for dead-key state records.

```
enum {
    kUCKeyStateEntryTerminalFormat = 0x0001,
    kUCKeyStateEntryRangeFormat = 0x0002
};
```

Constants

`kUCKeyStateEntryTerminalFormat`
: Specifies that the entry format is that of a structure of type `UCKeyStateEntryTerminal` (page 30). Use this format for simple (single) dead-key states, as in the U.S. keyboard layout.

  Available in Mac OS X v10.0 and later.

  Declared in `UnicodeUtilities.h`.

`kUCKeyStateEntryRangeFormat`
: Specifies that the entry format is that of a structure of type `UCKeyStateEntryRange` (page 29). Use this format for complex (multiple) dead-key states, as in the hex input and Hangul input keyboard layouts.

  Available in Mac OS X v10.0 and later.

  Declared in `UnicodeUtilities.h`.

Discussion
These constants are used in `UCKeyStateRecord` structures to indicate the format for dead-key state records.

Key Translation Options Flag

Indicates the dead-key processing state.

```
enum {
    kUCKeyTranslateNoDeadKeysBit = 0
};
```

Constants

`kUCKeyTranslateNoDeadKeysBit`

The bit number of the bit that turns off dead-key processing. This prevents setting any new dead-key states, but allows completion of any dead-key states currently in effect.

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

Discussion

Theis constant is the currently defined bit assignment for the `keyTranslateOptions` parameter of the function `UCKeyTranslate` (page 20).

Key Translation Options Mask

Specifies the mask for the bit that controls dead-key processing state.

```
enum {
    kUCKeyTranslateNoDeadKeysMask = 1L << kUCKeyTranslateNoDeadKeysBit
};
```

Constants

`kUCKeyTranslateNoDeadKeysMask`

The mask for the bit that turns off dead-key processing. This prevents setting any new dead-key states, but allows completion of any dead-key states currently in effect.

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

Discussion

This constant is the currently defined mask for the `keyTranslateOptions` parameter of the function `UCKeyTranslate` (page 20).

Operation Class

Identifies collation as a class of Unicode utility operations.

```
enum {
    kUnicodeCollationClass = 'ucol'
};
```

Constants

`kUnicodeCollationClass`

Identifies collation as a class of operations.

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

Discussion

The locales and collation variants available for collation operations can be determined by calling the Locales Utilities functions `LocaleOperationCountLocales` and `LocaleOperationGetLocales` with the `opClass` parameter set to the `kUnicodeCollationClass` constant.

Standard Options Mask

Specifies standard options for Unicode string comparison.

```
enum {
    kUCCollateStandardOptions = kUCCollateComposeInsensitiveMask
| kUCCollateWidthInsensitiveMask
};
```

Constants

`kUCCollateStandardOptions`

If the `kUCCollateComposeInsensitiveMask` and `kUCCollateWidthInsensitiveMask` bits are set, then (1) precomposed and decomposed representations of the same text element will be treated as equivalent, and (2) fullwidth and halfwidth compatibility forms will be treated as equivalent to the corresponding non-compatibility characters.

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

Discussion

For descriptions of other collation options, see "String Comparison Options" (page 41).

String Comparison Options

Specifies options for Unicode string comparison.

```
typedef UInt32 UCCollateOptions;
enum {
    kUCCollateComposeInsensitiveMask = 1L << 1,
    kUCCollateWidthInsensitiveMask = 1L << 2,
    kUCCollateCaseInsensitiveMask = 1L << 3,
    kUCCollateDiacritInsensitiveMask = 1L << 4,
    kUCCollatePunctuationSignificantMask = 1L << 15,
    kUCCollateDigitsOverrideMask = 1L << 16,
    kUCCollateDigitsAsNumberMask = 1L << 17
};
```

Constants

`kUCCollateComposeInsensitiveMask`

If the corresponding bit is set, then precomposed and decomposed representations of the same text element are treated as equivalent. This option is among those set by the `kUCCollateStandardOptions` constant, as described in "Standard Options Mask" (page 41).

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

`kUCCollateWidthInsensitiveMask`

If the corresponding bit is set, then fullwidth and halfwidth compatibility forms are treated as equivalent to the corresponding non-compatibility characters. This option is among those set by the `kUCCollateStandardOptions` constant, as described in "Standard Options Mask" (page 41).

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

`kUCCollateCaseInsensitiveMask`
: If the corresponding bit is set, then uppercase and titlecase characters are treated as equivalent to the corresponding lowercase characters.

  Available in Mac OS X v10.0 and later.

  Declared in `UnicodeUtilities.h`.

`kUCCollateDiacritInsensitiveMask`
: If the corresponding bit is set, then characters with diacritics are treated as equivalent to the corresponding characters without diacritics.

  Available in Mac OS X v10.0 and later.

  Declared in `UnicodeUtilities.h`.

`kUCCollatePunctuationSignificantMask`
: If the corresponding bit is set, then punctuation and symbols are treated as significant instead of ignorable. This will produce results closer to the behavior of the older non-Unicode Mac OS collation functions. This option is available with Mac OS 9 and later.

  Available in Mac OS X v10.0 and later.

  Declared in `UnicodeUtilities.h`.

`kUCCollateDigitsOverrideMask`
: If the corresponding bit is set, then the number-handling behavior is specified by the remaining number-handling option bits, instead of by the collation information for the locale. If the bit is clear, the locale controls how numbers are handled and the remaining number-handling option bits are ignored. This option is available with Mac OS 9 and later.

  Available in Mac OS X v10.0 and later.

  Declared in `UnicodeUtilities.h`.

`kUCCollateDigitsAsNumberMask`
: If the corresponding bit is set (and if the bit corresponding to `kUCCollateDigitsOverrideMask` is also set), then numeric substrings up to six digits long are collated by their numeric value—that is, they are treated as a single text element whose primary weight depends on the numeric value of the digit string. This primary weight will be greater than the weight of any valid Unicode character, but less than the primary weight of any unassigned Unicode character. For example, this will result in "Chapter 9" sorting before "Chapter 10." Currently, these digit strings can include digits with numeric value 0-9 in any script (excluding the ideographic characters for 1-9). If the bit is clear, digits are treated like other characters for sorting. Numeric substrings longer than 6 digits are always treated as normal characters. This option is available with Mac OS 9 and later.

  Available in Mac OS X v10.0 and later.

  Declared in `UnicodeUtilities.h`.

Discussion
For a description of the `UCCollateOptions` values, see "Standard Options Mask" (page 41).

Text Break Options

Specifies options for locating boundaries in Unicode text.

```
typedef UInt32 UCTextBreakOptions;
enum {
    kUCTextBreakLeadingEdgeMask = 1L << 0,
    kUCTextBreakGoBackwardsMask = 1L << 1,
    kUCTextBreakIterateMask = 1L << 2
};
```

Constants

`kUCTextBreakLeadingEdgeMask`

If the corresponding bit is set, then the starting offset for the `UCFindTextBreak` function is assumed to be in the word containing the character following the offset; this is the normal case when searching forward. If the corresponding bit is clear, then the starting offset for `UCFindTextBreak` is assumed to be in the word containing the character preceding the offset; this is the normal case when searching backward.

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

`kUCTextBreakGoBackwardsMask`

If the corresponding bit is set, then `UCFindTextBreak` searches backward from the value provided in its `startOffset` parameter to find the next text break. If the corresponding bit is clear, then `UCFindTextBreak` searches forward from the `startOffset` value to find the next text break.

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

`kUCTextBreakIterateMask`

The corresponding bit may be set to indicate to the `UCFindTextBreak` function that the specified starting offset is a known break of the type specified in the `breakType` parameter. This permits `UCFindTextBreak` to optimize its search for the subsequent break of the same type. When iterating through all the breaks of a particular type in a particular buffer, this bit should be set for all calls except the first (since the initial `startOffset` value may not be a known break of the specified type).

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

Text Break Types

Specifies kinds of text boundaries.

```
typedef UInt32 UCTextBreakType;
enum {
    kUCTextBreakCharMask = 1L << 0,
    kUCTextBreakClusterMask = 1L << 2,
    kUCTextBreakWordMask = 1L << 4,
    kUCTextBreakLineMask = 1L << 6
};
```

Constants

`kUCTextBreakCharMask`

If the bit specified by this mask is set, boundaries of characters may be located (with surrogate pairs treated as a single character).

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

`kUCTextBreakClusterMask`

If the bit specified by this mask is set, boundaries of character clusters may be located. A cluster is a group of characters that should be treated as single text element for editing operations such as cursor movement. Typically this includes groups such as a base character followed by a sequence of combining characters, for example, a Hangul syllable represented as a sequence of conjoining jamo characters or an Indic consonant cluster.

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

`kUCTextBreakWordMask`

If the bit specified by this mask is set, boundaries of words may be located. This can be used to determine what to highlight as the result of a double-click.

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

`kUCTextBreakLineMask`

If the bit specified by this mask is set, potential line breaks may be located.

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

Text Boundary Operation Class

Identifies the class of Unicode utility operations that find text boundaries.

```
enum {
    kUnicodeTextBreakClass = 'ubrk'
};
```

Constants

`kUnicodeTextBreakClass`

Identifies the class of Unicode utility operations that find text boundaries.

Available in Mac OS X v10.0 and later.

Declared in `UnicodeUtilities.h`.

Discussion

The locales and text-break variants available for finding boundaries in Unicode text can be determined by calling the Locales Utilities functions `LocaleOperationCountLocales` and `LocaleOperationGetLocales` with the `opClass` parameter set to the `kUnicodeTextBreakClass` constant.

Specification for 'uchr'

The Unicode keyboard-layout (`'uchr'`) resource contains the data necessary to map virtual key codes to Unicode character codes for a given keyboard layout. Each installed script system has one or more keyboard-layout resources, which may be of type `'uchr'` or `'KCHR'` (an older, non-Unicode keyboard-layout format). There may be one or more keyboard-layout resources for each language or region, depending upon the preference of the user.

The `'uchr'` resource ID is determined as for the `'KCHR'` resource, with one exception. That is, typically the resource ID for each Unicode keyboard-layout resource is within the range of resource ID numbers for its script system. The ID number of the default keyboard-layout resource for a script system is specified in the `itlbKeys` field of the script's international bundle (`'itlb'`) resource. The exception to this is that if a given `'uchr'` resource specifies any Unicode characters that are not within the range of a single Mac OS encoding (or are not in any Mac OS encoding), then you must use a negative number for the resource.

For a given resource ID, the system may contain a `'KCHR'` resource, a `'uchr'` resource, or both. If both a `'KCHR'` resource and a `'uchr'` resource are present, they must have the same ID, and the `'uchr'` resource should match the behavior of the `'KCHR'` resource. The keyboard menu shows each keyboard layout as a single entry, regardless of whether it is specified by a `'KCHR'`, a `'uchr'`, or both.

Note: The `'uchr'` resource contains offsets to tables that may be in any order. A `'uchr'` resource may be created in any data-description language that allows the specification of arbitrary binary data.

Unicode Keyboard-Layout Resource Format

The `'uchr'` format consists of a header information section and five key mapping data sections, as shown in Figure A-1.

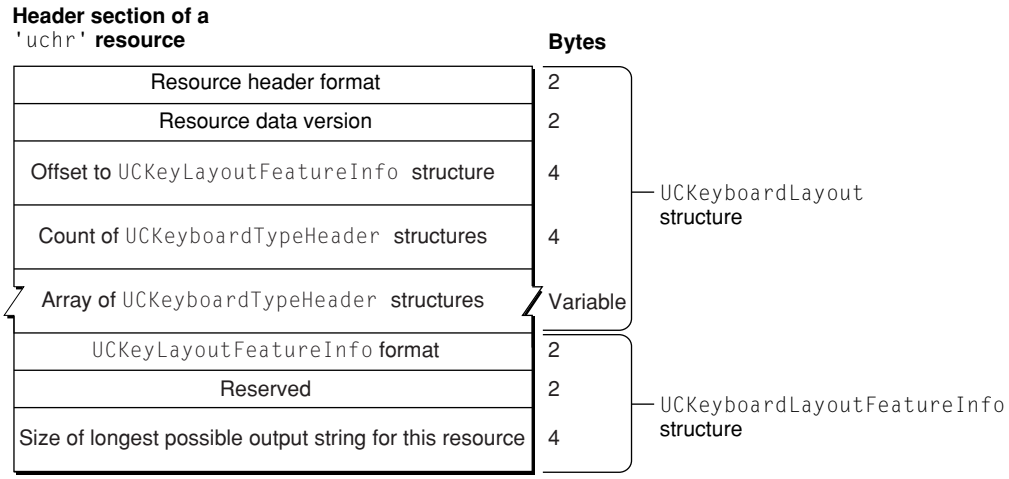
Figure A-1 Overview of a 'uchr' resource

Overview of a `'uchr'` resource

Resource header
Modifier-key-to-character table numbers
Key-code-to-character tables
Dead-key state records
Default dead-key state terminators
Character key sequences

The header section of a compiled `'uchr'` resource contains a structure of type `UCKeyboardLayout` and an optional structure of type `UCKeyLayoutFeatureInfo`. See for an illustration of this section.

Figure A-2 'uchr' resource header



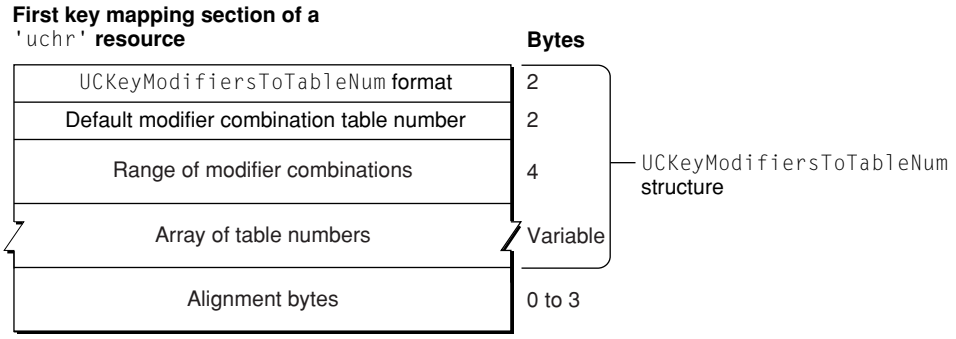
The elements in the header section of a `'uchr'` resource are

- the resource header format
- the version of the data in this resource
- an offset to a `UCKeyLayoutFeatureInfo` structure, if any
- a count of the `UCKeyboardTypeHeader` structures that follow
- an array of structures of type `UCKeyboardTypeHeader`; each `UCKeyboardTypeHeader` entry specifies a range of physical keyboard types and contains offsets to each of the key mapping sections to be used for that range of keyboard types
  - first keyboard type in this entry
  - last keyboard type in this entry
  - offset to the `UCKeyModifiersToTableNum` structure (required)
  - offset to the `UCKeyToCharTableIndex` structure (required)
  - offset to the `UCKeyStateRecordsIndex` structure (optional, may be 0 if there is no table)
  - offset to the `UCKeyStateTerminators` structure (optional, may be 0 if there is no table)
  - offset to the `UCKeySequenceDataIndex` structure (optional, may be 0 if there is no table)
- the format of the `UCKeyLayoutFeatureInfo` structure
- a reserved field
- a value of type `UniCharCount`, specifying the longest possible output string to be produced by this `'uchr'` resource

There may be a variable number of each of the following `'uchr'` key mapping sections.

The first key mapping section contains a structure of type `UCKeyModifiersToTableNum`, which maps a modifier key combination to a particular key-code-to-character table number; and alignment bytes. There may be multiple instances of this entire key mapping section. See Figure A-3 for an illustration of this section.

Figure A-3 'uchr' modifier combination to key-code-to-character table number map

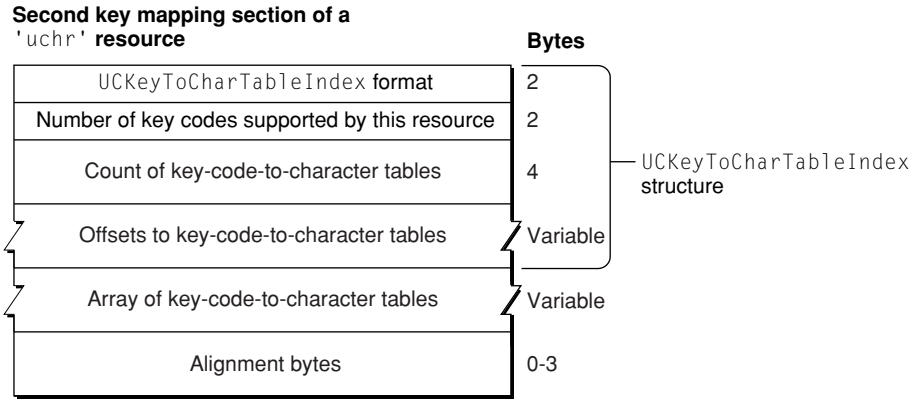


The elements in the first key mapping section of a `'uchr'` resource are

- the format of the `UCKeyModifiersToTableNum` structure
- the table number for modifier combinations that are outside of the range of the `tableNum` field's array; that is, the default (fallback) table number
- the range of modifier bit combinations for which there are entries in the `tableNum` field's array
- n an array of indexes into the key-code-to-character table offsets contained in the `UCKeyToCharTableIndex` structure in the next section
- alignment bytes (to a 4-byte boundary)

The second key mapping section contains a structure of type `UCKeyToCharTableIndex`; the list of key-code-to-character tables, each of which maps a virtual key code to a 16-bit `UCKeyOutput` value; and alignment bytes. There may be multiple instances of this entire key mapping section. See Figure A-4 for an illustration of this section.

Figure A-4 'uchr' key-code-to-character tables

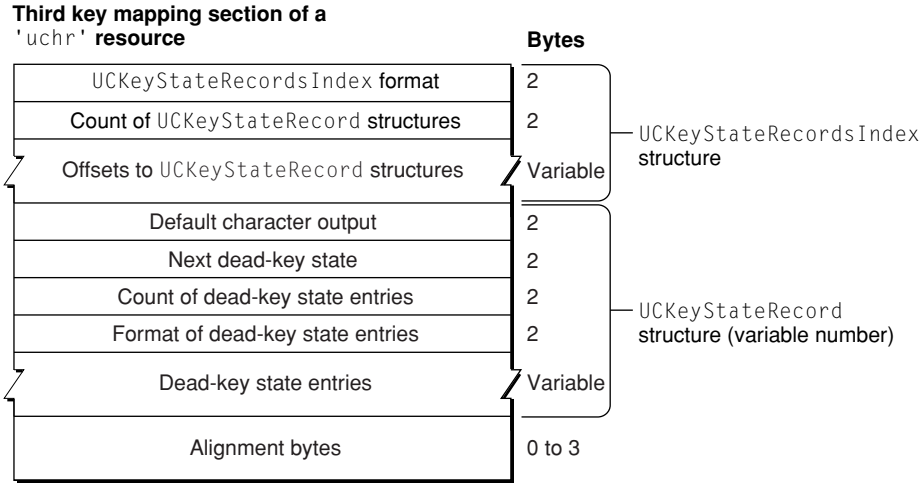


The elements in the second key mapping section of a `'uchr'` resource are

- the format of the `UCKeyToCharTableIndex` structure
- the number of virtual key codes supported by this resource
- a count of the key-code-to-character tables
- an array of offsets from the beginning of the resource to each of the key-code-to-character tables
- an array of key-code-to-character tables containing values of type `UCKeyOutput`
- alignment bytes (to a 4-byte boundary)

The third key mapping section is a map to dead-key state records. It contains a structure of type `UCKeyStateRecordsIndex`, which is an index to `UCKeyStateRecord` structures; a variable number of dead-key state records of type `UCKeyStateRecord`; and alignment bytes. There may be multiple instances of this entire key mapping section (or 0; this section need not be present if no `UCKeyOutput` value refers to a dead-key state record). See Figure A-5 for an illustration of this section.

Figure A-5 'uchr' dead-key state records



The elements in the third key mapping section of a `'uchr'` resource are

- the format of the `UCKeyStateRecordsIndex` structure
- a count of the dead-key state records to follow
- an array of offsets from the beginning of the resource to each of the `UCKeyStateRecord` values following

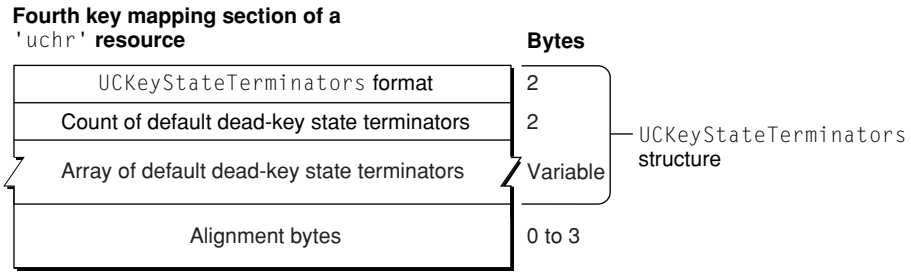
Immediately following the `UCKeyStateRecordsIndex` structure are a variable number of values of type `UCKeyStateRecord`. Any keycode-modifier combination which initiates a dead-key state or which is a valid terminator of a dead-key state refers to one of these records. However, these records also permit more complex dead-key state processing, such as a series of transitions from one dead-key state directly into another in which each transition can emit a sequence of one or more Unicode characters. Each record contains

- a value of type `UCKeyCharSeq` specifying the character(s) produced by the input keycode when no dead-key state is currently in effect
- a value specifying the dead-key state produced by the input keycode when no dead-key state is currently in effect

- a count of the elements in the `stateEntryData` field's array
- the format of the data in the `stateEntryData` field's array
- an array of dead-key state entry data; each entry maps from the current dead-key state to the character(s) that are produced or to the following dead-key state, if any
- alignment bytes (to a 4-byte boundary)

The fourth key mapping section contains a structure of type `UCKeyStateTerminators` and alignment bytes. This is an optional list of default terminators (characters or sequences) for each state; if this table is not present or does not extend far enough to have a terminator for the state, nothing is output when the state terminates. There may be multiple (or 0) instances of this entire key mapping section. See Figure A-6 for an illustration of this section.

Figure A-6 'uchr' default dead-key state terminators

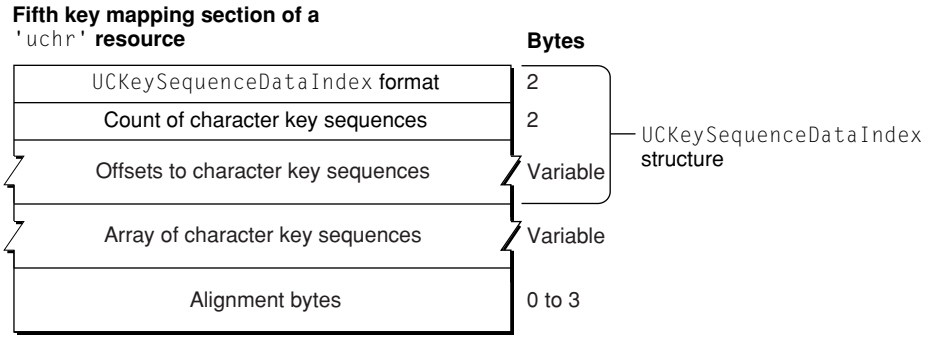


The elements in the fourth key mapping section of a `'uchr'` resource are

- the format of the `UCKeyStateTerminators` structure
- a count of default dead-key state terminators contained in the `keyStateTerminators` field's array
- an array of default dead-key state terminators, described as values of type `UCKeyCharSeq`
- alignment bytes (to a 4-byte boundary)

The fifth key mapping section of the resource is an optional list of character sequences; it contains a structure of type `UCKeySequenceDataIndex` and Unicode character sequences. This permits a single keypress to generate a sequence of characters, or to generate a single character outside the range that can be represented directly by a `UCKeyOutput` or `UCKeyCharSeq` value. There may be multiple (or 0) instances of this entire key mapping section. See Figure A-7 for an illustration of this section.

Figure A-7 'uchr' character key sequences



The elements in the fifth key mapping section of a `'uchr'` resource are

- the format of the `UCKeySequenceDataIndex` structure
- a count of the Unicode character sequences that follow the `UCKeySequenceDataIndex` structure
- an array of offsets from the beginning of the `UCKeySequenceDataIndex` structure to the Unicode character sequences that follow it
- an array of Unicode character sequences
- alignment bytes (to a 4-byte boundary)

Document Revision History

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

Date	Notes
2006-01-10	Updated the description of the options parameter for the UCCompareTextNoLocale function.
2005-07-07	Fixed typographical errors.
2003-05-01	Added information about collation keys to the functions `UCGetCollationKey` (page 18) and `UCCompareCollationKeys` (page 8).
2003-01-13	Updated formatting; added additional information to the discussion of the `UCKeyboardLayout` data structure and the `UCKeyTranslate` function.
	Added appendix that describes the `'uchr'` resource.

Document Revision History

Index

C

CollatorRef data type 22

F

Fixed Ordering Masks 1 35
Fixed Ordering Masks 2 36
Fixed Ordering Scheme 35

K

Key Actions 36
Key Format Codes 37
Key Output Index Masks 38
Key State Entry Formats 39
Key Translation Options Flag 39
Key Translation Options Mask 40
kUCCollateCaseInsensitiveMask constant 42
kUCCollateComposeInsensitiveMask constant 41
kUCCollateDiacritInsensitiveMask constant 42
kUCCollateDigitsAsNumberMask constant 42
kUCCollateDigitsOverrideMask constant 42
kUCCollatePunctuationSignificantMask constant 42
kUCCollateStandardOptions constant 41
kUCCollateTypeHFSExtended constant 35
kUCCollateTypeMask constant 36
kUCCollateTypeShiftBits constant 36
kUCCollateTypeSourceMask constant 35
kUCCollateWidthInsensitiveMask constant 41
kUCKeyActionAutoKey constant 37
kUCKeyActionDisplay constant 37
kUCKeyActionDown constant 37
kUCKeyActionUp constant 37
kUCKeyLayoutFeatureInfoFormat constant 38
kUCKeyLayoutHeaderFormat constant 37
kUCKeyModifiersToTableNumFormat constant 38
kUCKeyOutputGetIndexMask constant 39
kUCKeyOutputSequenceIndexMask constant 39
kUCKeyOutputStateIndexMask constant 38
kUCKeyOutputTestForIndexMask constant 39
kUCKeySequenceDataIndexFormat constant 38
kUCKeyStateEntryRangeFormat constant 39
kUCKeyStateEntryTerminalFormat constant 39
kUCKeyStateRecordsIndexFormat constant 38
kUCKeyStateTerminatorsFormat constant 38
kUCKeyToCharTableIndexFormat constant 38
kUCKeyTranslateNoDeadKeysBit constant 40
kUCKeyTranslateNoDeadKeysMask constant 40
kUCTextBreakCharMask constant 43
kUCTextBreakClusterMask constant 44
kUCTextBreakGoBackwardsMask constant 43
kUCTextBreakIterateMask constant 43
kUCTextBreakLeadingEdgeMask constant 43
kUCTextBreakLineMask constant 44
kUCTextBreakWordMask constant 44
kUnicodeCollationClass constant 40
kUnicodeTextBreakClass constant 44

O

Operation Class 40

S

Standard Options Mask 41
String Comparison Options 41

T

Text Boundary Operation Class 44
Text Break Options 42
Text Break Types 43
TextBreakLocatorRef data type 22

U

UCCollationValue data type 23
UCCompareCollationKeys function 8
UCCompareText function 9
UCCompareTextDefault function 11
UCCompareTextNoLocale function 12
UCCreateCollator function 13
UCCreateTextBreakLocator function 14
UCDisposeCollator function 16
UCDisposeTextBreakLocator function 16
UCFindTextBreak function 17
UCGetCollationKey function 18
UCKeyboardLayout structure 23
UCKeyboardTypeHeader structure 24
UCKeyCharSeq data type 25
UCKeyLayoutFeatureInfo structure 26
UCKeyModifiersToTableNum structure 27
UCKeyOutput data type 27
UCKeySequenceDataIndex structure 28
UCKeyStateEntryRange structure 29
UCKeyStateEntryTerminal structure 30
UCKeyStateRecord structure 31
UCKeyStateRecordsIndex structure 32
UCKeyStateTerminators structure 33
UCKeyToCharTableIndex structure 34
UCKeyTranslate function 20