Unicode Utilities in Mac OS 8.6 and Mac OS 9.0

Important

This is a preliminary document. Although it has been reviewed for technical accuracy, it is not final. Apple Computer, Inc. is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation.

You can check <http://developer.apple.com/techpubs/macos8/SiteInfo/whatsnew.html> for information about updates to this and other developer documents. To receive notification of documentation updates, you can sign up for ADC's free Online Program and receive their weekly Apple Developer Connection News e-mail newsletter. (See <http://developer.apple.com/membership/index.html> for more details about the Online Program.)

I. Introduction

Mac OS 8.5 introduced support for direct Unicode imaging in a QuickDraw environment (ATSUI) and for Unicode input (in UnicodeUtilitiesCoreLib). Full Unicode applications need support for various other Unicode text utilities such as collation, line-breaking and word select, date and time formatting, and so on. Many of these utilities are locale-sensitive—that is, they may depend on the conventions of a particular language or region—so Mac OS 8.6 introduced a Locale system to support the Unicode Utilities (the Locale system is documented separately). The introduction of these Unicode text utilities was staged as follows:

- Mac OS 8.6 introduced support for locale-sensitive Unicode collation (comparing Unicode strings); this support is enhanced in Mac OS 9.0. Mac OS 8.6 also introduced support for locale-insensitive Unicode collation.
- Mac OS 9.0 introduces support for locale-sensitive Unicode text boundary location to determine the boundaries of characters, clusters (for cursor movement) or words, and to determine potential line break locations.
- Mac OS 9.0 also introduces support for obtaining properties of a Unicode character. This information is not locale-sensitive, and support for this feature is provided by Text Encoding Converter version 1.5.
- Future Mac OS releases are expected to provide additional Unicode utilities for formatting dates, times, and numbers, etc.

The various Unicode input and text utilities are provided and documented as follows:

- Unicode keyboard translation functions (for Unicode input) are provided in the UnicodeUtilitiesCoreLib library and are documented separately.
- Locale-insensitive Unicode character property utilities are provided in the TextCommon library and are documented with the Text Encoding Converter.
- Other locale-insensitive Unicode text utilities are provided in the UnicodeUtilitiesCoreLib library and are documented here.
- Locale-sensitive Unicode text utilities are provided in the UnicodeUtilitiesLib library and are documented here.

II. Mac OS 8.6 and 9.0 implementation information

A. General information

1. Approach

Unicode text is passed as UniChar * and UniCharCount, instead of using CFStrings. Many developers prefer using text pointer and length. In any case, text passed as a CFString has to be copied out into a Unicode array for internal processing anyway, so there is no loss of efficiency if callers who are using CFStrings have to do this to call the Unicode text utilities functions.

The locale-sensitive Unicode text utilities use the Mac OS Locales system to manage the necessary data. For a given class of locale-sensitive operations, the locale system can enumerate the supported locales. For each class of operations, the Unicode text utilities include functions to create and dispose an object which is specific to that class and to a particular locale, and functions that use that object to perform various operations.

2. Special error codes for Unicode text utilities

In addition to paramErr and error codes from the Resource Manager and Memory Manager, the Unicode text utilities may return the following special error codes:

```
enum {
  kUCOutputBufferTooSmall = -25340,       // Unicode output buffer too small
  kUCTextBreakLocatorMissingType = -25341 // Bad break type for break object
};
```

More details are provided in specific function descriptions.

3. Interface files

- Include file: UnicodeUtilities.h (the Unicode properties interfaces are in TextCommon.h)
- Implementation libraries: UnicodeUtilitiesLib (for Unicode text utilities functions that depend on LocalesLib and TextCommon), UnicodeUtilitiesCoreLib (for Unicode key translation functions and Unicode text utilities functions with no dependencies); TextCommon (for Unicode character properties functions).
- Stub libraries for linking: UnicodeUtilitiesLib, UnicodeUtilitiesCoreLib; TextCommon.

B. About Unicode collation (string comparison)

In the world of computer standards, “collation” is usually used to refer to language-dependent ordering of strings; “comparison” is generally used to refer to simple non-language-dependent ordering (e.g. by code order).

1. Linguistic requirements

At first glance, collation may seem a simple task: Given some sorting order for characters, walk through two strings to be compared until non-identical characters are found, then order the strings by the sort order of those characters. In fact, collation is much more complex. Even proper English sorting for an 8-bit character set such as Mac OS Roman involves three levels of significance, ignorable characters, and expansion of some characters into multiple elements.

The first column below shows how an English dictionary would sort the following words; the second column shows a naïve sort based on Unicode code order.

Dictionary	Code order
coop	Cooper
co-op	Coors
Cooper	co-op

co-opt	co-opt
coördinate	coop
Coors	coördinate
ŒDIPUS	o'er
o'er	z
z	ŒDIPUS

The problem is not just that the code values for the characters aren't assigned in proper collating order; the problem is that there is *no* possible assignment of characters to collating positions that will produce the correct result with a single-level ordering. What is needed is collation that supports the following basic requirements (the first three are needed even for the English example above, the others are needed for other languages):

- Multiple levels of significance (at least 3): First try to order based on primary differences (c ≠ d); if there are no primary differences, then consider secondary differences (c ≠ ç); if there are no secondary differences either, then consider tertiary differences (c ≠ C).
- Ignorables: Certain characters should be ignored completely (e.g. '-', '' in the above example) unless they are the only difference between words. This can be considered an additional level of significance.
- Expansion: Certain characters need to be treated as multiple characters for collation (at the primary level, 'Œ' should be treated as 'OE'). Expansion can produce up to 18 characters in Unicode 2.1.
- Contraction: In some languages multiple characters are treated as a single unit for sorting. Traditional Spanish sorting treats 'ch' as a single letter that comes after 'c', and treats 'll' as a single letter that comes after 'l'; 'dz' is treated as a single unit in Czech, etc.
- Different primary orderings. For example, in some languages, certain letters with diacritics have a sorting position completely different from the letter without diacritics. In Danish, for example, the following are treated as letters that sort after z: ä, ö.
- French adds an interesting twist. When processing accents as secondary differences, strings are compared from the end to the beginning.
- Unicode has both precomposed and decomposed representations of many text elements; these should sort together.

Other languages add even more interesting context-dependence issues:

- In Japanese, the katakana vowel extender is often sorted at the primary level as if it were the katakana character for the vowel of the preceding katakana syllable (simple context dependence).
- In Japanese, kanji characters can have multiple pronunciations that depend on context; phrase analysis is required to determine the correct pronunciation when sorting by pronunciation ("yomi").
- In Arabic, the primary sort position for words may be that of the corresponding three-consonant root form, which must be determined by phrase analysis.

Within a given language, there may be several sorting variants for different purposes. For example, English variants include the following:

- Dictionary order
- Bibliographic order
- Telephone book order

Chinese and Japanese sorting of Han characters may support the following variants:

- Phonetic (Japanese yomi sorting, Chinese pinyin sorting, etc.)

- Radical-stroke (first sort by radical, then by stroke count)
- Stroke-radical (first sort by stroke count, then by radical)

2. Collation language support in Mac OS 8.6 and 9.0

The fallback locale provides a default collation order for the full range of Unicode characters. This is based on the collation algorithm and tables provided by the Unicode Consortium and described in their Unicode Technical Report UTR #10, “Unicode Collation Algorithm.” One notable feature of this collation order is that most punctuation and symbols are ignorable.

This default collation order is reasonable for most of the locales supported in Mac OS 8.6 and 9.0. Locales that require different behavior override portions of the default collation table. These languages include:

- French (to specify reverse diacritical sorting)
- Spanish, modern order (ñ has unique primary weight, between n and o)
- Swedish & Finnish (w ≈ v, ü ≈ y, and the following have unique primary weight , sorting after z: å, ä, ö)
- Danish & Norwegian (ü ≈ y, and the following have unique primary weight, sorting after z: ä, ö)

No locales currently provide collation variants. Such variants would also be implemented by overriding portions of the default collation table (or portions of some other collation table).

Please note that the locale-sensitive collation orders may change slightly from one system release to the next.

3. Principles of Unicode collation in Mac OS 8.6 and 9.0

The basic sequence of functions for performing collation is:

1. For a specified locale (type LocaleRef), a specified collation variant within that locale (type LocaleOperationVariant), and a specified set of options (type UCCollateOptions), use UCCreateCollator to create a collator object (type CollatorRef).
2. Use UCCompareText to compare strings using that collator object.
3. When finished with the collator object, dispose it using UCDisposeCollator.

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

```
enum {
  kUnicodeCollationClass = 'ucol' // constant for LocaleOperationClass
};
```

Collation options can be used by clients to change the collation behavior for a given collation table or control how it used (as opposed to using variants, which generally require a separate table). The options represent typical behaviors over which clients may want to have programmatic control. The UCCollateOptions type and the masks that can be ORed together in any combination to set its option bit flags are:

```
typedef UInt32 UCCollateOptions;

enum {
  // Sensitivity options
  kUCCollateComposeInsensitiveMask = 1L << 1,
  kUCCollateWidthInsensitiveMask = 1L << 2,
  kUCCollateCaseInsensitiveMask = 1L << 3,
  kUCCollateDiacritInsensitiveMask = 1L << 4,
```

```
  kUCCollateDiacritInsensitiveMask = 1L << 4,
  // Other general options (Mac OS 9.0 and later)
  UCCollatePunctuationSignificantMask = 1L << 15,
  // Number-handling options (Mac OS 9.0 and later)
  kUCCollateDigitsOverrideMask = 1L << 16,
  kUCCollateDigitsAsNumberMask = 1L << 17
  // Bits 24-31 are for use with UCCompareTextNoLocale to
  // specify the desired locale-insensitive ordering scheme.
};

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

The first set of options control what types of differences can be tolerated in order for two strings to be considered “equivalent”.

`kUCCollateComposeInsensitiveMask`	If the corresponding bit is set, then precomposed and decomposed representations of the same text element will be treated as equivalent.
`kUCCollateWidthInsensitiveMask`	If the corresponding bit is set, then fullwidth and halfwidth compatibility forms will be treated as equivalent to the corresponding non-compatibility characters.
`kUCCollateCaseInsensitiveMask`	If the corresponding bit is set, then uppercase and titlecase characters will be treated as equivalent to the corresponding lowercase characters.
`kUCCollateDiacritInsensitiveMask`	If the corresponding bit is set, then characters with diacritics will be treated as equivalent to the corresponding characters without diacritics.

The second set includes general-purpose options, of which only one is currently defined (it is only available in Mac OS 9.0 and later):

`UCCollatePunctuationSignificantMask`	If the corresponding bit is set, then punctuation and symbols will be treated as significant instead of ignorable. This will produce results closer to the behavior of the older non-Unicode Mac OS collation functions.

The third set of options are specific to handling numbers (this set is only available in Mac OS 9.0 and later):

`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.
`kUCCollateDigitsAsNumberMask`	If the corresponding bit is set (and if the bit corresponding to `kUCCollateDigitsOverrideMask is also`

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.

Bits 24-31 of UCCollateOptions are used with UCCompareTextNoLocale to specify the desired locale-insensitive ordering scheme. The values for this field are described with UCCompareTextNoLocale, below.

The standard options are specified using the constant kUCCollateStandardOptions.

A collator object created by UCCreateCollator has the following typedef:

```
typedef struct OpaqueCollator* CollatorRef;
```

Functions such as UCCompareText that actually compare strings have two output parameters for specifying the result of comparing two strings text1 and text2:

```
Boolean *equivalent,
SInt32 *order
```

The Boolean pointed to by equivalent is set to true if text1 and text2 are equivalent for the given UCCollateOptions. The integer pointed to by order is set as follows:

- -2 if text1 sorts before text2 and there are primary differences.
- -1 if text1 sorts before text2 and there are no primary differences.
- 0 if text1 is absolutely byte-for-byte identical to text2.
- +1 if text1 sorts after text2 and there are no primary differences.
- +2 if text1 sorts after text2 and there are primary differences.

Either pointer can be set to NULL if the caller is not interested in that particular result.

The reason for having two different result values is to facilitate the different ways that collation functions are used.

- Testing strings for equivalence, using specified options. This can be much faster that determining ordering. In this case the caller would pass NULL for the order parameter; the function can then take some shortcuts, since the caller is only interested in the equivalent result.
- Sorting a list of strings in order, using specified options. In this case, the strings must always be put in a deterministic order in relation to one another, even if they are considered “equivalent” for the specified options. The ordering is only irrelevant for strings that are absolutely identical. In this case the function of the options is to make the differences for which the ordering is insensitive—case, for example—less significant than the differences for which the ordering is

the ordering is sensitive. This way all strings considered equivalent will be sorted together, but they will still be in a deterministic ordering within that group as long as they are not identical. Here the caller would pass NULL for the `equivalent` parameter and only use the `order` result.

- Checking whether a given string is equivalent to any string in an ordered list. Here both results are required. The binary search comparison function can return 0 if the `equivalent` result is set to true; otherwise it can return -1 or +1 depending on whether the `order` result is less than or greater than 0.

The same result could have been achieved by using several different functions with different results and behavior, but experience suggests that the above approach provides more flexibility.

C. Unicode collation functions

As described above, the basic collation functions are:

- UCCreateCollator: Creates a collator objects for a specified locale, a specified collation variant within that locale, and a specified set of options. Having the options parameter in UCCreateCollator permits greater efficiencies.
- UCCompareText: Compare strings using that collator object (can call this multiple times to compare different string using the same collator object).
- UCDisposeCollator: Dispose the collator object when finished with it.

When the same strings will be compared several times—as when sorting a list of strings, for example—it is often more efficient 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 (i.e. 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 be quickly compared with each other using something similar to a strcmp()-like binary comparison, without further reference to the collator object or the collation tables. The disadvantage is that the collation keys may be rather large. The following functions are provided:

- UCGetCollationKey: Create a collation key from a particular string and collator object.
- UCCompareCollationKeys: Compare two collation keys that were generated with the same collator object.

Some clients may prefer a single, simple collation function that requires minimum setup and uses the system default collation order (i.e. the collation order for a LocaleRef of NULL and a variant of 0); these clients may still need to set collation options. The following convenience function is provided for these clients:

- UCCompareTextDefault: Compare two strings using default system locale and specified options.

Finally, some clients need to fixed, locale-insensitive comparison that is guaranteed to not 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 following 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.

- UCCompareTextNoLocale: Compare two strings using fixed locale-insensitive order, with specified options (no collator object required).

Another advantage of UCCompareTextNoLocale is that it is exported from the UnicodeUtilitiesCoreLib library, which does not depend on other libraries (the other functions above are exported from UnicodeUtilitiesLib, which depends on LocalesLib and TextCommon).

These functions are described in more detail in the sections that follow.

These functions are described in more detail in the sections that follow.

1. Creating a collator object

Creates a collator object for a specified locale, operation variant, and collation options.

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

You can use `LocaleOperationCountLocales(kUnicodeCollationClass, …)` and `LocaleOperationGetLocales(kUnicodeCollationClass, …)` to determine the locales and operation variants available for collation. You can set locale to NULL to request the default system locale. You can set opVariant to 0 to request the default collation variant for any locale.

You can set bits in the options parameter to specify various options. The options and the masks used to set them are described in section B.3 above.

The collator object is allocated in the current heap.

The function can return `paramErr` (e.g. if the collatorRef parameter is NULL). It can also return resource and memory errors.

This function can move memory.

2. Disposing a collator object

Disposes a collator object.

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

The function sets *collatorRef to NULL.

It can return paramErr (e.g. if the collatorRef parameter is NULL).

3. Using a collator object to compare strings

Compares two strings using a specified collator object.

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

The collatorRef must be valid (NULL is not allowed).

Either the equivalent or the order parameters may be NULL (but not both). The use of these parameters is described in section B.3 above.

The function can return paramErr (e.g. if collatorRef or text1Ptr or text2Ptr are NULL).

4. Using a collator object to generate collation keys

Generates a collation key for a string based on a particular collator object.

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

A collation key is a UCCollationValue array:

```
typedef UInt32 UCCollationValue;
```

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 level-2 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 given options.

The caller allocates collationKey and passes its dimension in maxKeySize. This dimension should typically be at least 5*textLength (the byte length of a collation key is typically more than 16 times the number of Unicode characters in the string).

The function can return paramErr (e.g. if collatorRef or textPtr or actualKeySize or collationKey are NULL). It can also return memory errors. If maxKeySize is too small for the collationKey, the function returns kUCOutputBufferTooSmall.

This function can move memory.

5. Comparing collation keys

Compares two collation keys, both based on the same collator object.

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

Either the equivalent or the order parameters may be NULL (but not both). The use of these parameters is described in section B.3 above.

This function can return paramErr (e.g. if key1Ptr or key2Ptr are NULL).

6. Comparing strings using a default collator object

Compare two strings using the default system locale and specified options.

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

You can set bits in the options parameter to specify various options. The options and the masks used to set them are described in section B.3 above.

Either the equivalent or the order parameters may be NULL (but not both). The use of these parameters is described in section B.3 above.

This function can return paramErr .

7. Comparing strings in a locale-insensitive way

Compare two-strings in a fixed, locale-insensitive way (does not require a collator object).

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

The high-order 8-bits of UCCollateOptions are used for a value that specifies which fixed ordering scheme to use. Currently only one such scheme is provided:

```
enum {
```

```
  kUCCollateTypeHFSExtended = 1
};
```

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

The following constants are provided for manipulating the UCCollateOptions field that specifies the ordering scheme.

```
// Constants for masking and shifting the invariant order type.
enum {
  kUCCollateTypeSourceMask = 0x000000FF,
  kUCCollateTypeShiftBits = 24
};
enum {
  kUCCollateTypeMask = kUCCollateTypeSourceMask << kUCCollateTypeShiftBits
};
```

For example, to specify kUCCollateTypeHFSExtended in the options parameter, it must be shifted by kUCCollateTypeShiftBits:

```
  options = kUCCollateTypeHFSExtended << kUCCollateTypeShiftBits;
```

To extract the ordering scheme value from the options parameter:

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

Either the equivalent or the order parameters may be NULL (but not both). The use of these parameters is described in section B.3 above.

This function can return paramErr .

This function is exported by UnicodeUtilitiesCoreLib, which does not depend on any other libraries.

D. About finding Unicode text breaks

The text break functions provide a general service for locating different types of breaks or boundaries in a line of text. These 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; a Hangul syllable represented as a sequence of conjoining jamo characters; and 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.

The general idea is that a “FindTextBreak” function starts from a specified offset in a text buffer, and then proceeds forward or backward (as requested) until it finds the next break point of the specified type.The actual sequence of functions for locating text breaks is:

1. For a specified locale (type LocaleRef), a specified text break variant within that locale (type LocaleOperationVariant), and a specified set of break types (type UCTextBreakType), use UCCreateTextBreakLocator to create a text break locator object (type TextBreakLocatorRef).
2. Use UCFindTextBreak with that text break locator object to locate the next text break of a particular type (from among the types specified with UCCreateTextBreakLocator), starting from a particular offset in a string and using a particular set of options (type UCTextBreakOptions).
3. When finished with the text break locator object, dispose it using UCDisposeTextBreakLocator.

The character break type is locale-independent, and support for it is built directly into the UCFindTextBreak function; no text break locator object is required. If that is the only break type being located, it is not necessary to call UCCreateTextBreakLocator and UCDisposeTextBreakLocator.

The locales and text break variants available for collation operations can be determined by calling the Locales functions `LocaleOperationCountLocales` and `LocaleOperationGetLocales` with the `LocaleOperationClass opClass` parameter set to the following constant:

```
enum {
  kUnicodeTextBreakClass = 'ubrk' // constant for LocaleOperationClass
};
```

Break types are specified using a bit mask. The following constants are used to set the bits in the bit mask corresponding to the desired types:

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

When using UCCreateTextBreakLocator to create a text break locator object, several bits may be set in its breakTypes parameter—i.e. several bit mask constants may be ORed together to specify all of the break types for which the text break locator is being created. When using UCFindTextBreak to find a text break, exactly one bit must be set in its breakType parameter—i.e. only one break type can be specified using a single mask constant.

A text break locator object created by UCCreateTextBreakLocator has the following typedef:

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

Text break locator options are used by clients to specify whether to search forward or backward for the next text break and to control other aspects of the search. The UCTextBreakOptions type and the masks that can be ORed together in any combination to set its bit flags for the options parameter of UCFindTextBreak are:

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

These options are described in more detail below:

These options are described in more detail below:

`kUCTextBreakLeadingEdgeMask`	If the corresponding bit is set, then the startOffset for UCFindTextBreak 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 startOffset for UCFindTextBreak is assumed to be in the word containing the character preceding the offset; this is the normal case when searching backward. Control of this behavior is separated out (instead of being implicit in the setting of the GoBackwards flag) to provide more flexibility.
`kUCTextBreakGoBackwardsMask`	If the corresponding bit is set, then UCFindTextBreak searches backward from the startOffset to find the next text break. If the corresponding bit is clear, then UCFindTextBreak searches forward from the startOffset to find the next text break.
`kUCTextBreakIterateMask`	The corresponding bit may be set to indicate to UCFindTextBreak that the specified startOffset is a known break of the type specified in the breakType parameter. This permits UCFindTextBreak to perform some optimizations in searching 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 may not be known break of the specified type).

In general, when searching forward, use `kUCTextBreakLeadingEdgeMask` and not `kUCTextBreakGoBackwardsMask`; when searching backward, use `kUCTextBreakGoBackwardsMask` and not `kUCTextBreakLeadingEdgeMask`.

E. Unicode text break functions

These functions are available in Mac OS 9.0 and later.

1.Creating a text break locator object

Creates a text break locator object for a specified locale, operation variant, and set of break types.

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

You can use `LocaleOperationCountLocales(kUnicodeTextBreakClass, …)` and `LocaleOperationGetLocales(kUnicodeTextBreakClass, …)` to determine the locales and operation variants available for finding text breaks. You can set locale to NULL to request the default system locale. You can set opVariant to 0 to request the default variant of the text break locator for any locale.

The breakTypes parameter is a mask in which you set a bit for every break type that the text break locator object should support (see section D for details). The BreakChar type is built-in; if that is the only type for which UCCreateTextBreakLocator is called, it will return a NULL breakRef (with no error).

The function can return paramErr (e.g. if the breakRef parameter is NULL or if invalid bits are set in the breakTypes parameter). It can also return memory and resource errors.

The function can move memory.

2. Disposing a text break locator object

Disposes a text break locator object.

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

The function sets *breakRef to NULL.

It can return paramErr (e.g. if the breakRef parameter is NULL).

3. Using a text break locator object to find a text break

Finds the next break of a specified type using a text break locator object.

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

The breakRef parameter must have exactly one bit set to specify a single break type.

If it specifies character break, then the breakRef parameter is ignored and may be NULL (since support for character break is locale-independent and is built in).

Otherwise, the breakRef must be non-NULL and valid. It must also support the break type specified in the breakType parameter; otherwise the function returns `kUCTextBreakLocatorMissingType`.

The options parameter contains bit flags to specify the operation of UCFindTextBreak; these are described in section D above.

The startOffset parameter specifies the offset from which UCFindTextBreak will begin searching for the next text break 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.

The UniCharArrayOffset pointed to by the breakOffset parameter will be set to the offset of the text break located by UCFindTextBreak.

In normal usage (when exactly one of `kUCTextBreakLeadingEdgeMask` and `kUCTextBreakGoBackwardsMask` are set), the result in *breakOffset will not be equal to startOffset 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 in *breakOffset can legitimately be equal to startOffset.