Locales API Preliminary Documentation

For Mac OS 8.6 and 9.0

Preliminary Draft

Technical Publications

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

Simultaneously published in the United States and Canada.

Locales 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.)

Many text operations—e.g. collation, text break determination, and formatting of dates, times, and numbers—have behavior that can depend on the conventions of a language and/or geographical region. These *locale-sensitive operations* need:

- A way to specify the language and/or region whose conventions are to be used; this is a *locale tag.* This can be a numeric value, a string, etc.
- A way to obtain, for the specified language and/or region, the *locale data* that will enable the text operation to behave appropriately.

A locale system is primarily a set of protocols, with only a thin layer of supporting code.

I. Background

A. Specifying locales - discussion

There are several types of information that are needed to indicate language and region.

1. **Language:** A human language; there are something like 5000-10000 living languages, plus many more extinct ones. ISO 639 defines a list of standard language codes consisting of 2 or 3 ASCII letters (case insensitive, but typically shown in lower case). There are currently 2-letter codes for about 140 languages, and 3-letter codes for about 430 (including all the languages that also have a 2-letter code). The 2-letter codes are typically used for specifying language in UNIX, Java, and on the Internet (Windows uses its own enumeration of numeric codes). However, many languages do not have a code; moreover, some of the existing codes are too general, such as 'zh' for Chinese.
2. **Language variant:** For some languages specified with ISO 639 codes, it may be necessary to further specify a variant (if this would affect computer handling of the language)—for example, the distinction between bokmål and nynorsk for Norwegian, or the distinction among the various Chinese “dialects” (really separate languages): Mandarin, Wu/Shanghainese, Cantonese, Fujienese, Hakka, etc.
3. **Script:** Generally computers deal with written language, even in text-to-speech or vice versa (where the text is being converted to or from written form). Thus in addition to language we may need information on the script in which a language is written. This cannot always be inferred from language—for example, the Azerbaijani language can be written in Arabic or

may need information on the script in which a language is written. This cannot always be inferred from language—for example, the Azerbaijani language can be written in Arabic or Cyrillic script; Malay can be written in Arabic or Latin, and even when a language has only one "natural" script, it may be written in transliterated form (e.g. Japanese or Russian in Latin script). A draft standard ISO 15924 defines 2-letter (and 3-letter) codes for about 90 scripts, and suggests another 90 or so that might need codes (although several of these are questionable). Han, Hiragana, Katakana, Bopomofo, and Hangul are all treated as separate scripts, but the ISO codes provide aliases for sets of these, e.g. Ja for Han + Hiragana + Katakana.

4. **Script variant:** As with language, the ISO script codes do not provide enough information to discriminate among all of the script variants that affect computer processing—for example, the distinction between use of simplified or traditional characters for Chinese; the distinction between monotonic and polytonic Greek.
5. **Region:** A country, territory, or region. ISO 3166 defines a list of codes consisting of two or three ASCII letters (usually shown in upper case). There are currently ISO codes for about 240 countries, territories, and other regions (covering all existing countries); ISO updates the list periodically as countries split or merge. The 2-letter codes are used to specify country in UNIX and Java; they are used to specify a regional language variant (such as U.S. English versus Canadian English versus British English) in Internet language tags (in this case they are supposedly being used as language variant specifiers rather than region specifiers, although their interpretation is often overloaded to imply region as well). The region specifier can affect choices such as local currency symbol which are independent of language.
6. **Region variant:** Any necessary variant. This could be used. for example, to subdivide countries by time zones, or to distinguish regions without separate ISO codes.

Elements 1-4 specify a "writing system". Language tagging of text could use either a writing system tag or a language tag (elements 1-2), since in the latter case the script could be inferred from the characters themselves.

Language, script and region can be treated independently; any combination is theoretically possible. For example: Yiddish language in Hebrew script with U.S. formats for numbers and currency, Yiddish language in Latin script with U.S. formats, Yiddish language in Katakana script with Japanese formats, etc.

Locale tags need not specify all of the elements listed above; default values can be inferred for many of the elements. For example, if the language is English and the script is not specified, then Latin script can be assumed for output, and for existing text the script of the actual characters can be used. Furthermore, many locale-sensitive operations do not depend on some of the locale elements. For example, number formatting may only be sensitive to region; collation may only be sensitive to language.

On the other hand, some classes of locale-sensitive operations require additional specifiers beyond the six elements above. For example, several types of collation may be available for some languages. For western languages these might include dictionary order, bibliographic order, telephone book order, etc. For Han character sorting in East Asian languages these might include code order, radical-stroke order, stroke-radical order, and pronunciation order (several options for this in Japanese). Date and time formatting may allow specification of calendar, which in turn may provide several formatting options. These are *operation-specific variants.*

B. Specifying locales - examples

1. **UNIX (POSIX, X/Open):** A string of the form `"ll_CC.charset"` where `"ll"` is an ISO-639 2-letter code, `"CC"` is an ISO 3166 country code, and `"charset"` specifies a particular character encoding (no ISO standard for this): for example, `"en_CA.ISO8859-1"`. For certain operation

operation classes, variants can be specified by appending `"@"` followed by the variant name: e.g. (for collation) `"en_CA.ISO8859-1@dictionary"`.

2. **Java:** A string of the form `"ll_CC_variant"` in which `"_variant"` is optional and possibly implementation-specific. For Java, of course, the charset is Unicode.
3. **Internet:** RFC 1766 specifies a mechanism for language tags (the mechanism is sometimes overloaded and also interpreted as a region specifier). The tag is an ASCII string consisting of a primary tag (1-8 letters) and optional subtags separated by hyphen. Usually the primary tag is a 2-letter code from ISO 639 and the first subtag is a 2-letter country code from ISO 3166. Non-country-code subtags can also be used to distinguish dialects, script, etc. The primary tag may be `"x-"` (private tag) or `"i-"` (IANA tag), in which case the next tag may be a non-ISO language specifier. Examples: `"en-US"`, `"i-cherokee"`, `"az-arabic"` vs. `"az-cyrillic"`, `"no-nynorsk"`.
4. **Windows:** Win32 uses a 16-bit language ID (LANGID) consisting of a 10-bit primary language ID and a 6-bit secondary language ID. The secondary language may imply dialect, country, script, or encoding. The full LANGID always implies a particular set of encodings. Win32 also uses a 32-bit locale ID (LCID) consisting of a LANGID plus a 4-bit sort ID (i.e. a sorting variant) and 12 reserved bits.

C. Locale data organization and use - examples

Here we need some examples before the general discussion.

1. UNIX

In UNIX a locale is a collection of certain types of data which apply to a language and region. This data is located in one or more text files in particular directories (which depend on the specific UNIX implementation). UNIX defines the categories of data and the formats in which it is specified. Two categories are worth special mention:

- UNIX locales specify a particular character encoding, and one category of UNIX locale data is character classification (e.g. identifying a particular character as whitespace, digit, letter, etc.). This is something that is not really dependent on language or region per se, so putting it in the locale data is dubious. It is also irrelevant for a Unicode-based approach.
- UNIX locales also specify the text of system messages. This is really user-interface localization data, and does not affect the operation of locale-sensitive text operations. Having this information in the locale is also questionable.

For each category, UNIX provides an environment variable that users (or programs) can set to a particular locale in order to specify the behavior for operations that depend on that category:

- **LC_CTYPE:** Character classification, case conversion, etc. (depends on encoding).
- **LC_COLLATE:** Collation (depends on language).
- **LC_TIME:** Formats for dates and times (depends on region); includes month and day names (and so depends on language too).
- **LC_NUMERIC** and **LC_MONETARY:** Formats for numbers and currency amounts (depends on region).
- **LC_MESSAGES:** Localized text for system messages (depends on language); does not affect the behavior of text operations.

There are several environment variables that determine which locale is used for a particular operation; these variables can be set to particular locales by users with the `setenv` command, or by programs with the `setlocale()` function. If the variable LC_ALL is set to a locale, then all operations use that locale. Otherwise, if the relevant category variable for an operation is set to a particular locale, the operation uses that locale. If none of these variables are set, the operation uses the locale specified by the LANG variable. Using different locales for different categories can be

problematic, since there is some interdependence (especially between LC_CTYPE and the other categories).

Locale-sensitive functions such as `strcoll()` do not have a locale parameter, they use the locale determined by the above procedure.

2. Java

From the Java 1.1 Internationalization Specification and the Java Locale class description:

> "In Java, a locale is simply an identifier for a particular combination of language and region. It is not a collection of locale-specific attributes. Instead, each locale-sensitive class maintains its own locale-specific information…A locale is the mechanism for identifying the kind of object [e.g. Collator, NumberFomat] that you would like to get. The locale is *just* a mechanism for identifying objects, *not* a container for the objects themselves.
>
> "Java programs are *not* assigned a single global locale. All locale-sensitive operations may be explicitly given a locale as an argument…While a global locale is not enforced, a system wide default locale is available for programs that do not wish to manage locales explicitly. A default locale also makes it possible to affect the behavior of the entire presentation with a single choice."

There are functions to return user-visible names in a specified language for each part of a locale identifier—language code, country code, or variant string—as well as for the whole locale.

> "Each class that performs locale-sensitive operations allows you to get all the available objects of that type. You can sift through these objects by language, country, or variant, and use the display names to present a menu to the user."

For example, the Collator() class includes (1) a getAvailableLocales() method to get the set of Locales for which Collators are installed; (2) a getInstance(opt Locale) method which gets the Collator for the specified locale if the optional Locale parameter is present, or gets the Collator for the current default Locale if the Locale parameter is not present.

> "Java's design means that there does not have to be a single set of supported locales, since each class maintains its own localizations."

3. Mac OS X

The locale mechanism for OS X Core Foundation and Cocoa is not finalized. However, one proposal is to use the CFDictionary type. Locale primitives would return a CFDictionary that corresponds to a particular language or to a particular region. Locale-sensitive operations would have a CFDictionaryRef parameter; clients could also extract the values for particular locale items from the CFDictionary.

The language-specific data and region-specific data would be stored in files in known directories. The information they contain may be overridden by user and corporate preferences in files in other known directories.

4. Windows

The full LANGID always implies a particular set of encodings (as in UNIX). LANGIDs can be used to retrieve language-specific user-interface elements (also as in UNIX).

Win 95 and NT both have the notion of a "system locale" and a "user's default locale"; the latter can be changed by a user using the International control panel. This control panel lets a user set their default country, language, keyboard layout, measurement unit, and various formats. NT also has the notion of a thread locale, which can be set programmatically; this is the locale that is supposedly used in retrieving language-specific user interface elements in NT.

Locale-sensitive APIs have a LCID parameter.

Locale-sensitive APIs have a LCID parameter.

The EnumSystemLocales function is an overall enumerator for installed locales, while the GetLocaleInfo function can be used to obtain information about a specific locale (such as its default Windows code page).

D. Locale data organization and use - discussion

From the above examples we can extract some relevant points of similarity and axes of differentiation about various locale approaches.

1. Locale tags: Number or string?

While it is convenient to use a number to represent a language or locale, as with the Windows LANGID and LCID or with the current Mac OS language and region codes, this implies maintaining a custom enumeration of these codes. ISO already has the task of providing a standard way to identify languages and regions, and there is no need to duplicate this effort.

Most of the inconvenience associated with using a string can be avoided by having a way to convert locale tag strings into a temporary scalar value that is valid for at least the life of a program, and using this instead.

2. Locales: System-wide data collections or not?

In UNIX (and potentially Mac OS X), a locale is explicitly a collection of data which is used for various locale-sensitive operations; the CFDictionary in Mac OS X would even provide a type for referencing this collection. With this model, it is possible to provide a system-wide enumeration of locales. This corresponds to a column-oriented view of the diagram below.

In Java, a locale is explicitly just the tag used to specify the behavior of a class of operations. Each class may support a different set of locales, and each class is responsible for managing the storage of its own locale data. There is no collection type for locale data. It is possible to enumerate the supported locales for a particular operation class only. This corresponds to a row-oriented view of the diagram below.

Locale tag →	en_US	fr_FR	jp_JP
GetCurrencySymbol	string	string	string
GetCollator	Collator *	Collator *	Collator *

In either case, data may be stored in text files or binary files, with a public or private format; it may be stored in a single file or in a set of files, in a declared location or not.

Relevant issues for deciding among these approaches are:

- Is it easy to add support for new types of locale-sensitive operations, which require new types of data (operation extensibility)?
- Is it easy for users to add support for new locales (locale extensibility)?
- Many locales will differ in only a small way from some other locale. Can locales inherit from other locales or specify a parent locale? This can help minimize redundant data.
- Does the mechanism integrate smoothly with mechanisms for user preferences and defaults (after all, choosing preferred locales is one aspect of user preferences, along with choosing user interface localization, etc.)?

3. Locale defaults

A locale system must provide a way to establish defaults. These can include:

- A system default.
- A user default or user preference. This may be global (i.e. they specify an entire locale that overrides the system default) or individual (they specify preferences for individual items, which override the values from the system default).
- A program preference. A program can specify a preferred locale for all of its operations (e.g. UNIX setlocale), or can specify a locale for each locale-sensitive operation (as in Java).

4. Using locales in APIs

There are several ways in which locales can be handled in APIs for locale-sensitive operations such as collation. The first three can be used with any locale model.

a) No locale parameter (as in UNIX strcoll); API uses a default established some other way.

b) API takes a locale tag:

```
CompareStrings(string1, string2, localeTag);
```

c) API takes an object specific to operation and locale tag: One API obtains an object (immutable) or creates an object (mutable) that is specific to the locale tag and operation class, another API uses that object to perform the desired operation. This can be faster if several similar operations will be performed. Also, if the object is mutable, there can be additional APIs that operate on the object to customize or tailor it in various ways.

```
collatorObject = GetCollator(localeTag); // or CreateCollator
// Here we could have functions that operate on the collatorObject
CompareStrings(string1, string2, collatorObject);
// we may need to dispose the collatorObject if mutable
```

UNIX uses approach (a) above, Java uses approaches (b) and (c), Windows mainly uses (b).

If locales are system-wide collections and if there is a type—such as a locale object—that refers to this collection, then the following approaches are also possible.

d) API takes a locale object:

```
localeObject = GetLocale(localeTag);
CompareStrings(string1, string2, localeObject);
```

e) API takes an object specific to operation and locale object:

```
localeObject = GetLocale(localeTag);
collatorObject = GetCollator(localeObject);
CompareStrings(string1, string2, collatorObject);
```

II. Mac OS 8.6 and 9.0 implementation information

A. General approach

1. Goals

The locale system for Mac OS 8.6 and 9.0 is intended to be adequate for the needs of the Unicode Utilities in those releases, while allowing enough future flexibility to be adaptable to the various "virtual platforms" supported on Mac OS X: Java, Cocoa, etc. It may be layered on top of these platforms or layered underneath them (i.e. used to support those platforms). Thus we have the following principles:

- Do not have APIs inconsistent with the Java approach to locales.
- For now, do not expose formats for locale data, so that these can be changed easily in the future.
- In the longer term, strive for sharing of locale data among as many virtual platforms as possible.

In addition, we have the following general goals:

possible.

In addition, we have the following general goals:

- Make it easy to add new locales, modify existing locales, or add information to existing locales (in order to support new operation classes, for example).
- Minimize duplication of data in multiple locales: provide a mechanism so data can be used by multiple locales, or so locales can inherit data from a parent locale.

2. Locale tags

Locales are specified using the six-part scheme described in section I.A, in which some parts may be empty. There are two types of locale tags, a *locale part string* and an opaque *LocaleRef*.

The locale part string is an ASCII string whose full form is “lan-var.sc-v_rg-v”, where “lan-var” specifies the language and variant, “.sc-v” specifies the script & variant, and “_rg-v” specifies the region and variant. Any of those three parts can be omitted; furthermore, the variant part “-var” or “-v” within any of those parts may be omitted. This string format is generally consistent with both the Internet language tag format and the POSIX/Java locale string format, except that the script part of a locale part string replaces the charset part of a POSIX locale string (since often the POSIX charset also serves to indicate script, and since the Mac OS locale system only supports utilities that use Unicode). The locale part string can be used to tag or specify language or locale in persistent storage. In APIs it is passed as

```
const char localeString[]
```

A LocaleRef is an opaque type which—in different system versions—could be a pointer, handle, or offset. It has the following typedef:

```
typedef struct OpaqueLocaleRef* LocaleRef;
```

It is valid at least during the lifetime of an application. However, it is not suitable for use in persistent storage, since it is not necessarily meaningful across multiple launches of an application.

There are functions to convert a locale part string—or an Internet language tag or POSIX/Java locale string—to a LocaleRef (there are also functions to convert Mac OS language or region codes to a LocaleRef). The LocaleRef is more convenient to pass in APIs, and the locale-sensitive Unicode Utilities APIs typically take a LocaleRef parameter. There are also functions to convert a LocaleRef back to a locale part string for persistent storage.

During the lifetime of a single boot there is a one-to-one relationship between a particular locale and a particular LocaleRef; two LocaleRefs constructed from equivalent source (e.g. equivalent strings) will be identical. Applications should never save LocaleRefs in a file or other persistent storage; instead, applications can save the original number or string used to construct the LocaleRef, or they can convert a LocaleRef to a locale part string containing the parts they care about and save the locale part string.

3. Locale-sensitive operation classes and locale variants

Locale-sensitive Unicode Utilities operations fall into several classes, such as collation, text break determination, date and time formatting, etc. The LocaleOperationClass type is used to specify such as class:

```
typedef FourCharCode LocaleOperationClass;
```

The specific LocaleOperationClass values depend on the Locales client. Examples (from Unicode Utilities) include:

```
kUnicodeCollationClass = 'ucol'
kUnicodeTextBreakClass = 'ubrk'
```

For some of these classes (such as collation), an operation-specific variant field can be used in addition to the LocaleRef. For collation, this might specify dictionary vs. bibliographic, or radical-stroke vs. pinyin, etc. This is analogous to the "@variant" part of a POSIX locale string or the final "_VARIANT" part of a Java locale string.

```
typedef FourCharCode LocaleOperationVariant;
```

4. Locale data

A new special folder for locale data was introduced with Mac OS 8.6. The English name of this folder is "Language & Region Support"; it is located at the top level of the System folder. It is supported by the Folder Manager, and has the folder type `kLocalesFolderType = 'ƒloc'`. Locale data is mainly in files of type `'lcfl'`, with creator `'lcal'`; such files are autorouted to the Language & Region Support folder.

A fallback locale in the System resource map provides default data for each supported locale-sensitive operation: collation, text break location, and so on. It also provides the localized names for operation classes.

Data for a particular locale can be spread among several files in this folder. Locale data such as a collation table can be shared among several locales. The formats of locale files, resources, and tables are currently private to provide flexibility for changes in future Mac OS releases.

The locale data in Mac OS 8.6 and 9.0 supports at least the following languages: Chinese (Mandarin in both simplified and traditional characters), Danish, Dutch, English, Finnish, French, German, Italian, Japanese, Korean, Norwegian, Spanish, Swedish.

5. Locale defaults

A LocaleRef of NULL implies system default locale. This is the locale associated with the system's user interface localization (i.e. the language of the Finder), updated by any selections the user has made in the appropriate control panel. For example, the system default collation is affected by the choice of text behaviors in the Text control panel.

6. Locale-sensitive APIs

The Unicode Utilities in Mac OS 8.6 and 9.0 use the API style described in I.D.4.c above: A create API creates a mutable object that is specific to the locale tag and operation class, another API uses that object to perform the desired operation, and finally a dispose API disposes the mutable object.

For example, to create a collator object for the French/France locale and for a particular variant:

```
status = LocaleRefFromLangOrRegionCode(langFrench, verFrance, &locale);
// or
status = LocaleRefFromLocaleString("fr_FR", &locale);
// then
status = UCCreateCollator(locale, opVariant , options, &collatorRef);
```

To create a collator object for the default locale, default variant:

```
status = UCCreateCollator(NULL, 0, options, &collatorRef);
```

To use the collator object and then dispose it:

```
status = UCCompareText(collatorRef, … /* strings & result pointer */ );
status = UCDisposeCollator(&collatorRef);
```

7. Error codes

```
enum {
```

```
  kLocalesBufferTooSmallErr       = -30001,
  kLocalesTableFormatErr          = -30002,
  kLocalesDefaultDisplayStatus    = -30029
};
```

These are explained in relation to the functions that return them.

8. Interface files

- Include file: MacLocales.h.
- Stub library for linking: LocalesLib
- Implementation library: LocalesLib

B. Manipulating locale tags

These functions convert among LocaleRefs, locale strings, and numeric language and locale codes.

1. Converting Mac OS Language and Region codes to LocaleRefs

```
OSStatus LocaleRefFromLangOrRegionCode(LangCode lang, RegionCode region,
                                       LocaleRef *locale);
```

Callers can specify either lang or region or both. The constant `kTextLanguageDontCare` is used for an unspecified language, and the constant `kTextRegionDontCare` is used for an unspecified region (these constants are in TextCommon.h).

LocaleRefFromLangOrRegionCode can move memory, so the locale parameter should not point to memory that can move.

The function returns paramErr if neither lang nor region is specified, or if both are specified but they are inconsistent, or if a specified lang or region is invalid. It also returns paramErr if the locale parameter is NULL. It can also return memory errors or resource errors. Finally, if the resources used for mapping language and region codes are invalid, it can return kLocalesTableFormatErr (in Mac OS 8.6, paramErr is returned for this instead).

2. Converting strings to LocaleRefs

```
OSStatus LocaleRefFromLocaleString(const char localeString[],
                                   LocaleRef *locale);
```

The localeString parameter is an ASCII string containing an Internet RFC 1766-style language tag, or a POSIX-style or Java-style locale string, or a Mac OS locale part string.

LocaleRefFromLocaleString can move memory, so the locale parameter should not point to memory that can move.

The function returns paramErr if the localeString or locale parameters are NULL, or if the localeString is malformed. It can also return memory errors .

3. Converting LocaleRefs to locale part strings

The following API is for getting identifying information out of a LocaleRef in a standardized string format (for program usage, not for user display):

```
typedef UInt32 LocalePartMask;
enum {                                  // bit set requests the following:
  kLocaleLanguageMask         = 1L<<0, // ISO 639-1 or -2 language code
  kLocaleLanguageVariantMask  = 1L<<1, // custom string for language variant
  kLocaleScriptMask           = 1L<<2, // ISO 15924 script code
  kLocaleScriptVariantMask    = 1L<<3, // custom string for script variant
```

```
  kLocaleRegionMask           = 1L<<4, // ISO 3166 country/region code
  kLocaleRegionVariantMask    = 1L<<5, // custom string for region variant
  kLocaleAllPartsMask         = 0x0000003F   // all of the above
};

OSStatus LocaleRefGetPartString(LocaleRef locale, LocalePartMask partMask,
                                ByteCount maxStringLen, char partString[]);
```

The bits set in partMask determine which parts will be included in the returned string. The caller provides storage of at least `maxStringLen` bytes for partString; `maxStringLen` should be at least 4 times the number of parts requested. The full form of the returned string is “lan-var.sc-v_rg-v” where “lan-var” specifies the language and variant, “.sc-v” specifies the script & variant, and “_rg-v” specifies the region and variant. Fields not selected by the partMask (and any field separator that precedes them) will not be included in the actual returned string.

The function returns paramErr if the partString parameter is NULL, or if the locale is invalid. It returns `kLocalesBufferTooSmallErr` if maxStringLen is too small for the requested string.

4. Converting strings to Mac OS language and region codes

This function is only available in Mac OS 9.0 and later.

```
OSStatus LocaleStringToLangAndRegionCodes(const char localeString[],
                  LangCode *lang, RegionCode *region);
```

This API maps from a locale string (which may be a part string obtained from LocaleRefGetPartString) to a combination of Mac OS language code and region code. Either the lang or the region parameter (but not both) can be NULL if the caller is not interested in that value.

If the localeString parameter is NULL or the string cannot be mapped to an existing language code and region code, the function returns `paramErr`. If the required resource is invalid, the function can return `kLocalesTableFormatErr`. The function can also return Resource Manager errors.

C. Enumerating supported locales

There are no APIs that provide system-wide enumeration of all supported locales, since this is inconsistent with the Java approach. The following APIs enumerate the supported locale & variant combinations for a particular class of operations, such as collation.

Some applications may want to display the list of locales and corresponding operation variants available for a given operation class, such as in a menu that permits users to choose desired behavior. There are two approaches to doing this:

- As a flat list of all available locale and variant combinations
- As a hierarchical list in which the user first selects a locale and then selects from among the variants available in this locale.

The functions below (as well as the name functions in the next section) return flat lists as in the first approach above. However, it is fairly easy to construct a hierarchical list by parsing the flat list.

1. Count locale and variant combinations

This function counts the total number of locale and variant combinations available for a given operation class. It should be called before LocaleOperationGetLocales in order to determine how much memory to allocate for the list.

```
OSStatus LocaleOperationCountLocales(LocaleOperationClass opClass,
                                ItemCount *localeCount);
```

The function returns paramErr if opClass is 0 or if the localeCount parameter is NULL.

2. List locale and variant combinations

This function fills out a list of all the locale and variant combinations available for a given operation class. The locale and variant combinations are specified using the LocaleAndVariant type (The opVariant field of LocaleAndVariant may be 0 for entries that do not have a specific opVariant.).

```
struct LocaleAndVariant {
  LocaleRef               locale;
  LocaleOperationVariant  opVariant;
};

OSStatus LocaleOperationGetLocales(LocaleOperationClass opClass,
                                   ItemCount maxLocaleCount,
                                   ItemCount *actualLocaleCount,
                                   LocaleAndVariant localeVariantList[]);
```

The caller allocates a LocaleAndVariant array of dimension maxLocaleCount and passes it to the function; the caller should use LocaleOperationCountLocales to determine the size of array to allocate.

If maxLocaleCount is too small for all of the locales, LocaleOperationGetLocales returns `kLocalesBufferTooSmallErr`.

The function returns paramErr if opClass is 0 or if either the actualLocaleCount or localeVariantList parameter is NULL.

D. Obtaining localized names for locales and operation classes

This section describes three groups of functions.

- For a particular locale and operation variant, there are functions to obtain the corresponding name as localized for a particular display locale, or to iterate through all of the available display locales for the corresponding names as localized in each of the display locales.
- For a particular operation class, there are functions to obtain the corresponding name as localized for a particular display locale, or to iterate through all of the available display locales for the corresponding names as localized in each of the display locales.
- There is a special convenience function that, for a particular Mac OS region code, returns the name of the corresponding language in that language and in the non-Unicode Mac OS text encoding used for that region.

In some languages, the name for a language or locale may have several different grammatical forms; the correct form depends on the usage or context. For example, Swedish uses different forms of a language name depending on whether the name is applied to a collation order, to text break rules, to keyboard layouts, etc. The functions described here do not provide a way to specify such context information, and only return one form of a language or locale name; this is typically the form that would be used for the isolated language name. However, this form will not be the correct form for some usages in some languages.

Addressing this in a future Mac OS release will require providing another group of functions that combine the capabilities of the functions in the first two groups above.

1. Obtaining the localized name for a locale and/or variant

The following functions are for obtaining the localized name(s) for a locale, a locale and variant combination, or a variant alone. The functions are somewhat similar in usage to the ATSUI

functions ATSUFindFontName, ATSUCountFontNames, ATSUGetIndFontName.

The LocaleNameMask type is used to specify which parts of the name are requested: The name of the locale alone, the name of the operation variant alone, or the name for the combination.

```
typedef UInt32 LocaleNameMask;
enum {                                          // bit set requests:
  kLocaleNameMask                 = 1L<<0,    // name of locale
  kLocaleOperationVariantNameMask = 1L<<1,    // name of LocaleOperationVariant
  kLocaleAndVariantNameMask       = 0x00000003  // all of the above
};
```

```
OSStatus LocaleGetName(LocaleRef locale, LocaleOperationVariant opVariant,
                       LocaleNameMask nameMask, LocaleRef displayLocale,
                       UniCharCount maxNameLen, UniCharCount *actualNameLen,
                       UniChar displayName[]);
```

The locale and opVariant parameters indicate the locale and operation variant for which the name is requested. The displayLocale parameter indicates the requested language for the name. If LocaleGetName cannot find a name that matches the requested displayLocale, it will use a name in a default displayLocale and return `kLocalesDefaultDisplayStatus`.

If maxNameLen is too small for the requested string, then LocaleGetName returns `kLocalesBufferTooSmallErr`.

To count and index through all available names, the following functions can be used.

```
OSStatus LocaleCountNames(LocaleRef locale, LocaleOperationVariant opVariant,
                       LocaleNameMask nameMask, ItemCount *nameCount);
```

```
OSStatus LocaleGetIndName(LocaleRef locale, LocaleOperationVariant opVariant,
                       LocaleNameMask nameMask, ItemCount nameIndex,
                       UniCharCount maxNameLen, UniCharCount *actualNameLen,
                       UniChar displayName[], LocaleRef *displayLocale);
```

For LocaleGetIndName, the nameIndex ranges from 0 to nameCount -1 as in ATSUGetIndFontName; this is different from the index use in GetIndResource (for example). The displayLocale parameter indicates the language of the current name.

If maxNameLen is too small for the requested string, then LocaleGetIndName returns `kLocalesBufferTooSmallErr`.

All of the functions described in this section can move memory.

2. Obtaining the localized name for an operation class

There is a parallel set of functions to get the localized name(s) for an operation class.

```
OSStatus LocaleOperationGetName(LocaleOperationClass opClass,
                          LocaleRef displayLocale, UniCharCount maxNameLen,
                          UniCharCount *actualNameLen, UniChar displayName[]);
```

```
OSStatus LocaleOperationCountNames(LocaleOperationClass opClass,
                          ItemCount *nameCount);
```

```
OSStatus LocaleOperationGetIndName(LocaleOperationClass opClass,
                          ItemCount nameIndex, UniCharCount maxNameLen,
                          UniCharCount *actualNameLen, UniChar displayName[],
                          LocaleRef *displayLocale);
```

All of the functions described in this section can move memory.

3. Obtaining the localized language name for a region

This function is only available in Mac OS 9.0 and later.

```
OSStatus LocaleGetRegionLanguageName(RegionCode region, Str255 languageName)
```

Given a region code, this convenience API returns a string containing the name for the corresponding language in that language and in the appropriate Mac OS encoding for that region.

This function can move memory.