Internationalization Programming Topics

User Experience

2009-08-07

Apple Inc.
© 2003, 2009 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

.Mac is a registered service mark of Apple Inc.

Apple, the Apple logo, Carbon, Cocoa, eMac, Finder, iPhone, Mac, Mac OS, Objective-C, Quartz, QuickTime, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

IOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction to Internationalization Programming Topics 7

Organization of This Document 7
See Also 8

Internationalization and Localization 9

Support for Internationalization 11

How Users Specify Language Preferences 11
How Users Specify Locale Preferences 12
Bundles 13

Language and Locale Designations 17

Language Designations 17
Regional Designations 18
Language and Locale IDs 18
Language-Specific Project Directories 20
Getting Language Names from Designators 20
Using Custom Designators 21
Legacy Language Designators 21

Guidelines for Internationalization 23

Use Canonical Language and Locale IDs 23
Use Bundles 23
Support Unicode Text 23
Guidelines for Adding MultiScript Support 24
Carefully Consider Translatable Strings 25

Getting the Current Language and Locale 27

Specifying the Supported Localizations in Your Bundle 27
Getting the Preferred Localizations 27
Getting Canonical Language and Locale IDs in Mac OS X 28
Getting Language and Locale Preferences Directly in Mac OS X 28

Preparing Your Nib Files for Localization 31

Using Nib Files Effectively 31
Viewing Localizable Strings in Interface Builder 32

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Using ibtool 32
Generating Localized Nib Files 33

Localizing String Resources 35

Internationalizing Other Resources 37

Resources and Core Foundation 37
Resources and Cocoa 37

Localizing Pathnames 39

Getting Localized Path Names 39
Localizing Your Application Name 40
Localizing Directory Names 40

Notes For Localizers 43

Localizing Strings Files 43
Localizing Nib Files 44

File Encodings and Fonts 45

File Systems and Unicode Support 45
Getting Canonical Strings 46
Cocoa Issues 46

Document Revision History 47

Index 49

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Figures, Tables, and Listings

Support for Internationalization 11

Figure 1 Language preferences 11
Figure 2 Locale preferences 13
Listing 1 Structure of a Mac OS X application bundle 14

Language and Locale Designations 17

Table 1 Examples of ISO language designations 17
Table 2 Examples of ISO regional designators 18
Table 3 Custom language ID tags 19

Preparing Your Nib Files for Localization 31

Figure 1 Strings window in Interface Builder 32

Localizing String Resources 35

Listing 1 A simple strings file 35

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Introduction to Internationalization Programming Topics

Today's applications are marketed to a global audience. Selling your applications to that audience requires the customization of your software for each target market. Users in a foreign country are not going to want a user interface in a language they do not understand. Similarly, there may be images that you find acceptable but which are considered quite rude in other cultures. The problem is how do you create software in a language that you do not understand? The answer is through the internationalization technologies found in Mac OS X and iOS.

Rather than rewrite your software for each language you want to support, you can internationalize it to support any language. This process involves separating any user-visible text and images from your executable code. Once you have this data isolated into separate resource files, you can translate it into the desired languages and integrate the localized resource files back into your application bundle. This document helps you understand the steps needed to prepare your application for these processes.

Organization of This Document

This document includes the following articles:

- "Internationalization and Localization" (page 9) introduces the process and terminology associated with internationalization and localization.
- "Support for Internationalization" (page 11) describes the support for internationalization provided by Mac OS X and iOS.
- "Language and Locale Designations" (page 17) describes the conventions for identifying languages and locales in your application.
- "Guidelines for Internationalization" (page 23) provides tips to help you internationalize your software.
- "Getting the Current Language and Locale" (page 27) shows you how to find out which localization is currently in effect.
- "Preparing Your Nib Files for Localization" (page 31) provides tips for localizing your nib files including how to extract strings using `ibtool`.
- "Localizing String Resources" (page 35) introduces you to string resource files and their benefits.
- "Internationalizing Other Resources" (page 37) provides tips on how to localize programmatically-generated content using the current locale information.
- "Localizing Pathnames" (page 39) describes the Mac OS X support for localized bundle and directory names and shows you how to support this feature in your application.
- "Notes For Localizers" (page 43) provides tips for people who localize content in Mac OS X or iOS.
- "File Encodings and Fonts" (page 45) provides legacy information related to file encodings in previous versions of Mac OS, along with information about how to support those encodings in Mac OS X. It also describes some issues surrounding the use of fonts with different file encodings.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

See Also

The bundle mechanism plays a prominent role in supporting localized versions of an application. In addition, part of the internationalization process involves using resource files instead of hard coding strings and other localizable content into your executable. You should therefore read the following books for related information about the internationalization process:

- *Bundle Programming Guide* provides information about the structure of bundles and how they support localized content.
- *Resource Programming Guide* provides information about resource files (including string resource files) and how you load them into your application.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Internationalization and Localization

Internationalization is the process of designing and building an application to facilitate localization. **Localization**, in turn, is the cultural and linguistic adaptation of an internationalized application to two or more culturally-distinct markets. When users launch a well-localized application, they should feel fully at home with it and not feel like it originated from some other country.

Internationalization and localization are complementary activities. By internationalizing an application, you create the programmatic infrastructure needed to support localized content. By localizing that application, you then add resources that are customized for people of a different culture. Localizing an application often involves translating the user-visible text of the application, but that is only part of the work that must be done. Other parts of your application must also be modified, including the following:

- Nib files (windows, views, menus) must be modified to accept translated text.
- Static text must be translated.
- Icons and graphics (especially those containing culture-specific images) must be changed to be more culturally appropriate.
- Sound files that contain spoken language must be rerecorded for each supported language.
- Online help must be translated.
- Dynamic text generated by your program (including dates, times, and numerical values) must be formatted using the current locale information.
- Text handling code must calculate word breaks using the current locale information.
- Tabular data must be sortable using the current locale information.

The process for internationalizing a Mac OS X application is easy and the same on both platforms. Inside your application's bundle directory, there is one file containing your application's executable code and potentially many separate files containing the localized resources. The resource files for a single localization are stored together in a **language-specific project directory**, which is a directory whose name is a combination of the language name and the `.lproj` filename extension. A bundle may contain any number of language-specific project directories, each one representing a distinct localization that the application supports. You can even store localizations for single-byte and double-byte languages together in the same bundle.

The first step in the development of an internationalized application is to identify all culture-specific information in your application. Scour your user interface for culture-specific text, images, and sounds and put them into resource files. As you do that, update the underlying code so that it loads the data it needs from those resource files. As you go through your code, you should also look for places where your user interface displays date, time and currency values and make sure you are formatting them using the current locale information.

Once your application's user interface is frozen (which may be before the code freeze), you can begin to localize it. For each localization, the localization team creates language-specific versions of the nib files, text, images, and sound files. If your application is properly internationalized, the process doesn't require modifications to your source code. You can therefore create the translations in-house or contract with an outside localization service.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Even if you do not plan to support multiple languages immediately, internationalizing your application up front is a good idea. There is also no penalty for building internationalization support into your application. Even if your application is monolingual, you still need to test all available code paths. If your code is properly internationalized for one language, it requires little additional testing to support other languages. In addition, storing resources outside of your executable file makes it easier to change the appearance of your application later without modifying the underlying code.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Support for Internationalization

Both Mac OS X and iOS support internationalization and localized content in a variety of ways.

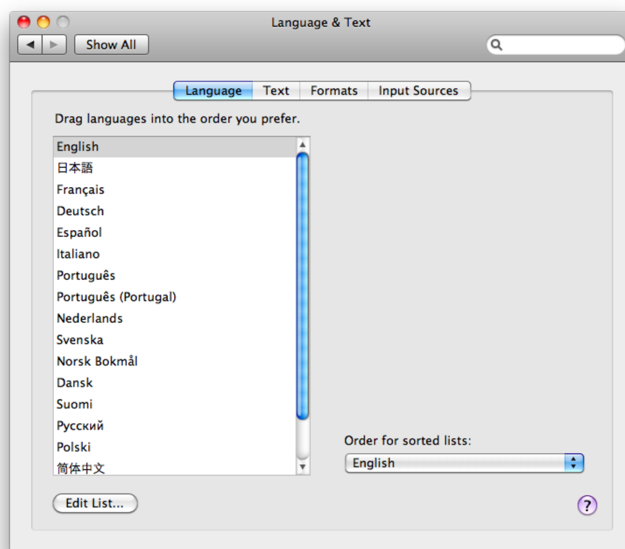
- They allow the user to specify a set of preferred languages.
- They provide a mechanism for storing multiple language-versions of resources within an application.
- Mac OS X allows additional languages to be added to an application, even at run time.

The notion of a preferred language set and the mechanisms of bundles, localized resources, and runtime binding are behind this support.

How Users Specify Language Preferences

Users specify the languages they would prefer to see using the preferences of the target platform. Figure 1 shows the interface used to select language preferences in both Mac OS X (v10.6) and iOS.

Figure 1 Language preferences



Mac OS X **iPhone OS**

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

In iOS, the user designates a preferred language by navigating to General > International > Language in the Settings application. Because iOS-based devices support only one user, only one language at a time is selected. In Mac OS X, the Language & Text system preference panel lets the user designate both a preferred language and one or more fallback languages in case the preferred language is unavailable.

The system stores the list of languages as a per-user default under the `AppleLanguages` key. It is not recommended that you retrieve the value of this key directly though. The codes found in the database may not include the canonical forms of the language or locale IDs. Instead, you can use the `preferredLanguages` method of the `NSLocale` class or the `CFLocaleCopyPreferredLanguages` function to retrieve the user's list of preferred languages.

When your application requests the path to a resource file in its bundle directory, the bundle interfaces search for a resource whose localization most closely matches the preferred languages of the user. The bundle interfaces consist of the `NSBundle` class in the Foundation framework and the functions supporting the `CFBundleRef` type in Core Foundation. These interfaces provide support for locating resource files inside your application's bundle directory. When searching that directory, they always return the localized resource that is most appropriate for the current user. If a localization does not exist for the user's preferred language, the search continues using the user's preferred fallback languages (where appropriate). If none of the bundle's localizations match the user preferences, the interfaces return the resource associated with the localization used during development of the software.

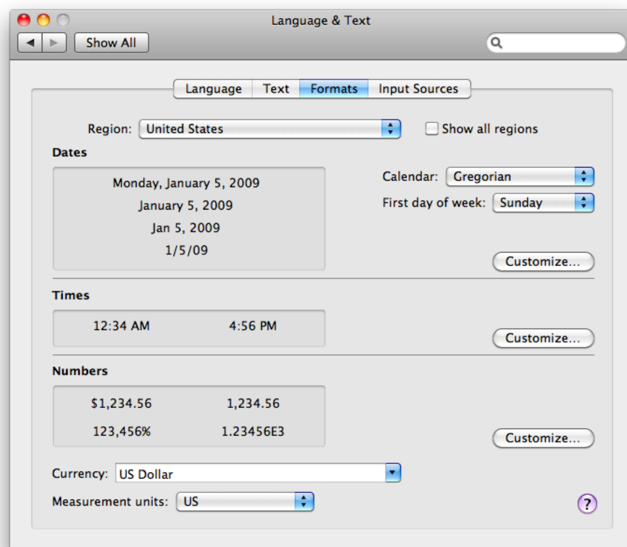
When searching for resources in Mac OS X, the bundle interfaces take into account the user's preferred region settings in addition to the preferred language settings. For languages that have multiple dialects, an application can specify custom resource files for each region. For example, for the English language, an application might include different sets of resource files for users in the United States, Great Britain, Canada, and Australia. By using the user's preferred region settings, the bundle interfaces are able to return even more precisely localized content to the user. This support is available in Mac OS X only, though; in iOS, only the preferred language is taken into consideration.

How Users Specify Locale Preferences

Locale preferences convey regional differences for the way dates, times, and numbers are displayed. Mac OS X and iOS specify predefined locales for many world regions, as shown in Figure 2. In iOS, the user can select the desired region only, but in Mac OS X the user can also customize the information for any of the default locales to further personalize the system.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Figure 2 Locale preferences



Mac OS X **iPhone OS**

In most cases, your code should never have to worry about applying locale preferences. Most APIs take the user's preferences into account when getting or formatting locale-sensitive data. However, there may still be situations where you need to get locale information. For example, you might want to display the current locale settings, do comparisons of different locales, or apply locales other than the current locale to a specific piece of data. In those situations, the Foundation framework provides the `NSLocale` class and the Core Foundation framework provides the `CFLocaleRef` opaque type for retrieving locale information.

Bundles

An **application bundle** is a directory containing your application's executable binary and any associated resource files. Although iOS supports only application bundles, Mac OS X supports many different kinds of bundles, including framework and plug-in bundles. Bundles provide a way to organize complex pieces of software to make them more manageable. The bundle directory acts as a top-level container for all of the files needed to make the application run. The structure inside that directory is then used to separate code and resources and to organize them in ways that make it easier to find things.

> **Note:** In code, the term "bundle" is also used when referring to an `NSBundle` object or a `CFBundleRef` opaque type. These types are the programmatic representation of the bundle directory and are used to locate files inside the bundle.

In order to understand how resource files for multiple languages may be stored inside a single application bundle, it is worth taking the time to examine the basic bundle structure of an application. Listing 1 shows the partial structure of a Mac OS X application that contains a few localized resource files. The text on the left identifies the directories and files in the bundle while the descriptions on the right indicate the purpose of the key items.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Listing 1 Structure of a Mac OS X application bundle

```
MyApp.app/                      The bundle directory
    Contents/
        MacOS/
            MyApp               The application executable
        Info.plist
        Resources/              All resources go here
            MyApp.icns          Non-localized resources
            PeaceSign.tiff
            en.lproj/           Resources localized for English
                StopSign.eps    A localized image
                Localizable.strings    Contains localized strings
                MyApp.nib       A localized nib file
            fr.lproj/           Resources localized for French
                StopSign.eps
                Localizable.strings
                MyApp.nib
            ja.lproj/           Resources localized for Japanese
...
```

As you can see from this structure, an internationalized application needs one `.lproj` directory for each localization it supports. The directories are named according to the language-version of the resources they contain. By default, a project directory contains only a single language directory corresponding to the language you are using to develop the application.

Although you could simply copy the files in your development language directory and translate them, the preferred way to create new localized resource files is using Xcode. When you select a resource file and open an inspector window, the General tab includes a Make Localizable button. Clicking this button for a resource file tells Xcode that it should place a copy of the original resource file in a language-specific project directory at build time. Each copy is based on the initial contents of the resource file. As your translators localize the files though, you can replace the initial file with the newly translated version and rebuild your project.

In an application bundle, there are several standard types of resources that require localization:

- **Nib files** - Nib files are created by the Interface Builder application and comprise an encoded set of objects, including the configuration of those objects and the connections between them. A nib file can also include references to other resource files in your project, including image and sound resources.

  Translation teams usually localize nib files directly using Interface Builder. A localizer takes a nib file (or, more typically, your application's set of nib files), translates all the strings, and makes other adjustments as necessary. For example, if the new strings do not fit correctly, the translator may need to resize user interface objects to accommodate the new strings. Translators may also need to replace culture-specific images and sounds.

  It is a good idea to minimize the number of objects stored in a single nib file. A common scenario is to include a single window (and any objects needed to support that window) in a nib file and place different windows in different nib files. Organizing your nib files in this manner typically improves the performance of your application and also allows localization to proceed incrementally.

- **Images**- Applications can store image files using any number of image formats, including `TIFF`, `GIF`, `JPEG`, and `PNG` among others. You can also include QuickTime movie files.

- **Localized text strings** - String resources are stored in a special type of file known as a **strings file**. A strings file is a text file with a `.strings` filename extension. Any string resources that might be displayed to the user should be placed in strings files. For more information, see "Localizing String Resources" (page 35).

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

If your application displays resource file names to the user (such as image file or sound file names), you should store the resource file names themselves in a `.strings`file. This way, you can translate the name of the resource file and its contents. To load the resource, you first request the resource file name from the corresponding `.strings` file. Once you have the name, you can load the resource as usual.

- **Sounds** - Like images, sound resources can be stored in any number of formats and opened using Core Audio or other audio technologies.
- **Online help** - Online help files are typically stored as HTML files. The application's property list allows you to specify which file your application should open when the user chooses the Help command. A localized variant of your help should be placed in the appropriate `.lproj` directory for each targeted localization.

At localization time, you send the source-language `.lproj` folder (for example, `en.lproj` for developers in English-speaking countries) to a localization service, or to your in-house translation department. You also need to send the compiled application to allow the localizer to view dynamically-loaded resources in context, to ensure appropriate translations and to ensure adequate dimensions for user interface elements that display dynamically-loaded resources. For each target language, the translator sends back a `.lproj` folder in which each resource file has been appropriately localized.

The tool used by localization teams to edit resource files varies by resource type. Nib files are typically edited using the Interface Builder application. Other resource files can be edited with an appropriate editor. For strings files, you can use most text editors in Mac OS X. Whenever possible though, use an editor that can save text in Unicode format, such as the TextEdit application. String files saved in Unicode format can be used in a running application directly, while other file formats must be converted to Unicode before they can be used.

For more information about bundles, the structure of bundles (including the structure of iPhone applications), and how you use the bundle interfaces to locate resources, see *Bundle Programming Guide*.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Language and Locale Designations

Mac OS X and iOS support existing and forthcoming International Organization for Standardization (ISO) standards for the identification of languages and locales. Specifically, they support the language and locale codes that are defined by the BCP 47 specification. These codes are used in the naming of language-specific project directories and in other places where language and locale information is needed.

Important: If your Mac OS X software runs in versions of Mac OS X prior to version 10.4, you must continue to use the existing ISO language and locale ID conventions. Use of the tags found in the BCP 47 specification will not work on versions of Mac OS X prior to 10.4.

Using the available conventions, you can distinguish between different languages and between different regional dialects of a single language. The following sections show you how to specify this information in your code.

Language Designations

For language designations, you can use either the ISO 639-1 or ISO 639-2 conventions. The ISO 639-1 specification uses a two-letter code to identify a language and is the preferred way to identify languages. However, if an ISO 639-1 code is not available for a particular language, you may use the three-letter designators defined by the ISO 639-2 specification instead. Table 1 lists ISO designators for a subset of languages. Note that there is no ISO 639-1 designator for Hawaiian and so you must use the ISO 639-2 designator.

Table 1 Examples of ISO language designations

Language	ISO 639-1	ISO 639-2
English	`en`	`eng`
French	`fr`	`fre`
German	`de`	`ger`
Japanese	`ja`	`jpn`
Hawaiian	no designator	`haw`

For a complete list of ISO 639-1 and ISO 639-2 codes, go to http://www.loc.gov/standards/iso639-2/php/English_list.php.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Regional Designations

For regional designations, you can use the ISO 3166-1 conventions. This specification uses a two-letter, capitalized code to identify a specific country. By concatenating a language designator with an underscore character and a regional designator, you get a designator that identifies the locale for a specific language and country. Table 2 lists the regional designators for a subset of languages and countries.

Table 2 Examples of ISO regional designators

Regional dialect	Region Designator
English (United States)	`US`
English (Great Britain)	`GB`
English (Australian)	`AU`
French (France)	`FR`
French (Canadian)	`CA`

For a complete list of ISO 3166-1 codes, go to http://www.iso.ch.

Language and Locale IDs

A **language ID** designates a written language (or orthography) and can reflect either the generic language or a specific dialect of that language. To specify a language ID, you use a language designator by itself. To specify a specific dialect of a language, you use a hyphen to combine a language designator with a region designator. Thus, the English language as it is spoken in Great Britain would yield a language ID of `en-GB`, while the English language spoken in the United States would have a language ID of `en-US`. To specify the generic version of the English language, you would use the language ID `en` by itself.

Important: In iOS, the bundle interfaces do not take dialect or script information into account when looking for localized resources; only the language designator code is considered. Therefore if your project includes language-specific project directories with both a language and region designator, those directories are ignored. The bundle interfaces in Mac OS X do support region designators in language-specific project directories.

A **locale ID** identifies a specific location where a given language is spoken. To specify a locale ID, use an underscore character to combine a language designator with a region designator. The locale ID for English-language speakers in Great Britain is `en_GB`, while the locale for English-speaking residents of the United States is `en_US`. Although locale IDs and language IDs might seem nearly identical, there is a subtle difference. A language ID identifies a written and spoken language only. A locale identifies a region and its conventions and has a more cultural context.

To illustrate the difference between language IDs and locale IDs, consider the following example. The dialect for a resident of Great Britain is specified by the code `en-GB`. The commonly used locale for that same person is `en_GB`. If you wanted to be very precise when specifying the locale, you could specify the locale code as

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

en-GB_GB. This specifies a person who speaks the British dialect of English and who resides in Great Britain. If that same person moved to the United States, the appropriate locale would be en-GB_US, which would identify a person who speaks British English but uses the regional settings associated with the United States.

In Mac OS X v10.4 and later (and iOS), you can use the language ID tags defined in the BCP 47 specification. In addition to the ISO 3166-1 region codes, the draft of this standard (available at http://www.rfc-editor.org/) adds support for tags ranging in length from 3 to 8 characters. The use of these tags makes it possible to separate dialect or script information from a specific region or country.

Particularly in Chinese dialects, a region code is not always the best way to specify the proper dialect or script. For example, traditional Chinese (Han) is the default language spoken in Taiwan and is identified by the code zh_TW in Mac OS X v10.3.9 and earlier. However, traditional Chinese is also commonly spoken in Hong Kong and Macao, which means the zh_TW designator is not entirely accurate in those locations. The new standard defines new tags for the traditional Chinese (Hant) and simplified Chinese (Hans) scripts. Thus, traditional Chinese spoken in any country uses the code zh-Hant. Traditional Chinese, as it is spoken in Taiwan, now uses the locale code zh-Hant_TW.

Table 3 lists some of the other custom tags that identify a particular dialect or script.

Table 3 Custom language ID tags

Language ID	Description
az-Arab	Azerbaijani in the Arabic script.
az-Cyrl	Azerbaijani in the Cyrillic script.
az-Latn	Azerbaijani in the Latin script.
sr-Cyrl	Serbian in the Cyrillic script.
sr-Latn	Serbian in the Latin script.
uz-Cyrl	Uzbek in the Cyrillic script.
uz-Latn	Uzbek in the Latin script.
zh-Hans	Chinese in the simplified script.
zh-Hant	Chinese in the traditional script.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Important: In Mac OS X v10.4 and later, you can use the new `Hant` and `Hans` tags instead of the older `zh_TW` and `zh_CN` tags for `.lproj` directory names if you choose. You must not use these tags (or any of the newer tags) in Mac OS X v10.3.9 and earlier, however. For these older applications, you should use the Core Foundation and Cocoa routines to obtain the canonical form of a given language or locale tag before using that tag in your code. For information on how to get the canonical tags, see "Getting Canonical Language and Locale IDs in Mac OS X" (page 28).

Language-Specific Project Directories

The more general you make your localized resources, the more regions you can support with a single set of resources. This can save a lot of space in your bundle and helps reduce translation costs. For example, if you did not need to distinguish between different regions of the English language, you could include a single `en.lproj` directory to support users in the United States, Great Britain, and Australia. More importantly, you must use a single language directory on platforms such as iOS, which do not recognize dialect-specific resource files.

Important: Even if you support region-specific localizations, you should always provide a complete set of common resources that are not region-specific.

When searching for resources, the system bundle routines try to find the best match between the `.lproj` directories in your bundle and the user's language and region preferences. In iOS, the bundle routines look for the requested resource in the generic language directory for the user's preferred language, followed by any other generic language directories. In Mac OS X, the bundle routines look for the requested resource in any region-specific directories first, followed by more generalized language directories. For example, if your Mac OS X application had localizations for United States, Great Britain, and Australian users, the bundle routines would search the appropriate region directory (`en_US.lproj`, `en_GB.lproj`, or `en_AU.lproj`) first, followed by the `en.lproj` directory. The same application on the iPhone would look only in the `en.lproj` directory.

For more information about how the bundle routines locate resources, see "Accessing a Bundle's Contents" in *Bundle Programming Guide*.

Getting Language Names from Designators

Prior to Mac OS X v10.4, the ISO language and region designators are the recommended way of identifying languages in the system. However, few users can recognize languages by their ISO designators. If you need to display the actual name of a language to a user, you can use the Carbon functions defined in `MacLocales.h` to convert the designators into localized language names. For more information, see *Locale Utilities Reference*.

If your software runs in Mac OS X v10.4 and later, you should not use the functions in `MacLocales.h`. Instead, use the `CFLocaleCopyDisplayNameForPropertyValue` function to get the correct display name for the language or locale ID.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Using Custom Designators

It is possible (albeit discouraged) to use a language or locale abbreviation that is not known to the `NSBundle` class or Core Foundation bundle functions. For example, you could create your own language designations for a language that is not yet listed in the ISO conventions.

If you choose to create a new designator, be sure to follow the rules found in sections 2.2.1 and 4.5 of BCP 47. Tags that do not follow these conventions are not guaranteed to work. When using custom tags, you must ensure that the abbreviation stored by the user's language preferences matches the designator used by your `.lproj` directory exactly.

Legacy Language Designators

In addition to the ISO language designators, the bundle routines also recognize several legacy language designators. These designators let you specify a language by a user-readable name, instead of by a two or three character code. Designators included names such as `English`, `French`, `German`, `Japanese`, `Chinese`, `Spanish`, `Italian`, `Swedish`, and `Portuguese` among others. Although these names are still recognized and processed by the `NSBundle` class and Core Foundation bundle functions, their use is deprecated and support for them in future versions of Mac OS X or iOS is not guaranteed. Use the codes described in "Language Designations" (page 17) and "Regional Designations" (page 18) instead.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Guidelines for Internationalization

The following sections provide recommendations on how to internationalize your software to take advantage of the support available on your target platform.

Use Canonical Language and Locale IDs

Mac OS X 10.3 added support for obtaining canonical locale IDs through Core Foundation functions associated with the `CFLocaleRef` type. In Mac OS X v10.4 and later (and iOS), support was added for getting canonical language IDs. In Mac OS X v10.4, the `NSLocale` class was added, bringing similar support to the Foundation framework. (These features are also available in iOS.) If you get the user's preferred locale ID through the `NSBundle` class or Core Foundation bundle functions, then you should already be receiving a canonical locale ID.

For information about language and locale IDs, see "Language and Locale Designations" (page 17). For information on how to get canonical language and locale IDs, see "Getting Canonical Language and Locale IDs in Mac OS X" (page 28).

Use Bundles

The bundle mechanism simplifies the process of localization by letting you place localized resources in separate directories named for the language or region they support. Both the `NSBundle` class and `CFBundleRef` functions use this collection of resource folders to localize a program based on the current user's language and region settings. This mechanism makes it possible for a bundled program to display itself in different languages for different users. Organizing resources by directory also makes it easy to add new localizations later.

Support Unicode Text

Applications should use Unicode-based technologies in all situations that involve displaying text to the user. Unicode support is essential for multibyte languages and the system supports some languages (including Arabic, Thai, Greek, and Hawaiian) only through Unicode. Regardless of whether you localize your software for multibyte languages, you should still support Unicode so that users can enter text data in their native language.

The system provides extensive support for managing and displaying Unicode text. Core Foundation and Cocoa strings provide a way to store Unicode strings in your application. In Mac OS X, technologies such as Core Text, Quartz, and Cocoa Text all leverage Unicode in the display of text. Use of these technologies makes it possible to support virtually any language.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

While Unicode is important for text that is displayed to the user, you do not need to use it everywhere in your application. For example, you might not need to use Unicode for debugging text or internal logging mechanisms, unless you want those messages to be translated and displayed to users.

Guidelines for Adding MultiScript Support

An application that provides multiscript support can accurately display text in various scripts simultaneously. Such an application can accept text input, display text, and print text containing the scripts of different languages in the same document, regardless of a user's language preferences. If an application were not prepared to offer multiscript support, some of this text would not appear correctly.

Multiscript support is becoming an increasingly important and expected feature not only for operating systems but for third-party applications. Users expect to be able to create a document in one language, change their language preference, and then open the document as they last saved it.

To add multiscript support to an application, you must use the appropriate Unicode technologies and API. The classes of the Foundation framework automatically provide multiscript support. For Carbon and other non-Cocoa applications, you should use the following technologies:

- Wherever possible, use the `CFStringRef` type from Core Foundation String Services instead of C or Pascal strings.

  `CFStringRef` types store and handle Unicode data internally without requiring you to have any specific knowledge of the global character-set standard. If you need to convert between Unicode and C and Pascal strings in other encodings, use the facilities that Core Foundation provides. However, avoid converting `CFStringRef` types to and from C strings or Pascal strings if possible. Such conversions are computationally expensive and frequently introduce bugs affecting multiscript presentation. If you cannot find any or Unicode-aware interfaces to use in a certain situation, convert the string to the application's text encoding.

- To format dates use the `CFDateFormatterRef` type and associated functions in Core Foundation.
- To format numbers, use the `CFNumberFormatterRef` type and associated functions in Core Foundation.
- For localized strings, use the `CFCopyLocalizedString` macro (and related macros) of the Core Foundation framework instead of the `GetIndString` function.

  Strings returned by `GetIndString` are specific to the current script system and thus cannot represent multiscript text. `CFCopyLocalizedString` returns Unicode-based `CFStringRef` opaque types, which can represent multiscript text. Generally, your code should use dynamic text processing over static text in your code. For more information about how to load localized strings in your code, see "String Resources" in *Resource Programming Guide*.

- Never assume text data to be in the MacRoman encoding.

  You can not assume that all text data uses the MacRoman encoding or ignore text encoding issues altogether. Untagged text data unaccompanied by script code is not necessarily in the system script (Roman). If you make this assumption, users may be presented with incoherent text. In the worst case, your text processing routines could misinterpret the text and either corrupt user data or crash the system.

- In Mac OS X, avoid accessing system layers below the Core Services framework.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

You should be able to obtain most of the functionality available in the kernel environment (particularly BSD and Mach calls) using the Core Services frameworks. As much as possible, avoid calling BSD functions directly. For accessing the file system, use Core Foundation URL Services (`CFURLRef` type) or the `NSFileManager` object in Cocoa.

- In Mac OS X, avoid using the TextEdit API.

  The TextEdit API is capable of dealing with multiscript text. However, it requires you to manage script fonts and style runs yourself. The Core Text framework is the preferred way to manage text.

For more information on file encodings in Mac OS X, see "File Encodings and Fonts" (page 45).

Carefully Consider Translatable Strings

When writing text for your application, try to think about how that text might be translated. Grammatical differences between languages can make some text difficult to translate, even with the formatting options available in strings files. In particular, strings that require differentiation between singular and plural forms (because of a count variable, for example) may require different strings in different circumstances. Talk to your translation team about ways to eliminate potential translation problems before they occur.

For more information about creating strings files, see String Resources in *Resource Programming Guide*.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Getting the Current Language and Locale

If your application is properly internationalized, you should rarely have to choose which localization to apply to use. The routines associated with the `NSBundle` class and `CFBundleRef` opaque type automatically apply the user's preferences to determine which localized resource files to return in response to a programmatic request. However, there may be situations beyond simply loading resources from a bundle that would require your program to know which localization was in use. For those situations, both `NSBundle` and `CFBundleRef` provides ways for you to get that information.

Specifying the Supported Localizations in Your Bundle

Before discussing the techniques of how to get localizations, it is important to remember how a bundle communicates its supported localizations to the system. The directory containing the bundle's resources can also contain one or more language-specific project (`.lproj`) directories. Each of these `.lproj` directories contains the resources associated with a specific language or regional dialect. The methods of the `NSBundle` class and functions associated with the `CFBundleRef` opaque type look for these directories and use them to build a list of supported localizations. However, this is not the only way to build this list.

An application can notify the system that it supports additional localizations through its information property list (`Info.plist`) file. To specify localizations not included in your bundle's `.lproj` directories, add the `CFBundleLocalizations` key to this file. The value for the key is an array of strings, each of which contains an ISO language designator as described in "Language and Locale Designations" (page 17).

Getting the Preferred Localizations

If you are using the functions associated with the `CFBundleRef` opaque type to manage your program bundle, you can use the `CFBundleCopyPreferredLocalizationsFromArray` function to get the most relevant localizations. When calling this method, you must pass in a list of localizations your software supports.

You can also use the function `CFBundleCopyBundleLocalizations` to generate this list for you using the bundle information. This function compares your application's supported localizations against the user's language and region preferences. The function then returns an array of strings with a handful of localizations. One of them is the language-only localization most appropriate for the user. If a region-specific localization is also available, the function returns it as well, giving it precedence over the language-only localization.

If you are writing your application in Objective-C, you can use the `preferredLocalizationsFromArray:` and `localizations` methods of the `NSBundle` class to implement the same behavior as the `CFBundleCopyPreferredLocalizationsFromArray` and `CFBundleCopyBundleLocalizations` functions. You can also use the `preferredLocalizations` method as a shortcut to perform both actions with a single method. As with the Core Foundation functions, the `preferredLocalizations` and `preferredLocalizationsFromArray:` methods return an array of strings containing the language designators in use by the bundle.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Getting Canonical Language and Locale IDs in Mac OS X

Prior to Mac OS X v10.4, language dialects were specified by combining a language designator with a region designator. With the introduction of support for custom dialect codes (see "Language and Locale IDs" (page 18)), getting the appropriate language code is now somewhat more complicated. Fortunately, Mac OS X provides routines to help you determine the appropriate language and locale codes based on the information you have.

In Mac OS X v10.3, the `CFLocaleRef` opaque type was introduced in Core Foundation. One of the functions introduced with this type is the `CFLocaleCreateCanonicalLocaleIdentifierFromString` function, which takes the locale code you specify and returns an appropriate canonical version. This function is particularly useful for converting older locale strings, such as the older, English-based .lproj directory names, into the ISO-compliant names.

In Mac OS X v10.4, the `CFLocaleCreateCanonicalLanguageIdentifierFromString` function was added to perform the same canonical conversion for language and dialect codes. For example, this function converts the old specifier for traditional Chinese (`zh_TW`) to the more modern version (`zh-Hant`). Also in Mac OS X v10.4, Cocoa added the `NSLocale` class to provide Objective-C wrappers for the corresponding Core Foundation functions.

If you use a `CFBundleRef` type or `NSBundle` object to retrieve language-specific resources from your bundle, then you do not need to worry about language identifiers directly. The `CFBundleRef` and `NSBundle` routines automatically handle language and locale IDs in canonical and non-canonical forms.

If your code requires Mac OS X v10.4 or later, you should start using the new canonical forms for language and locale IDs. Some older language codes are replaced by newer codes in v10.4. In addition to several of the Chinese language codes, support for the newer Norwegian ISO code (`nb`) is now available and should be preferred over the older version.

Note: If your program also supports versions of Mac OS X prior to 10.3, you may need to maintain your own table of canonical IDs.

Getting Language and Locale Preferences Directly in Mac OS X

There may be situations where you want to get the preferred locale ID or the list of languages directly from the user preferences. Mac OS X stores each user's list of preferred languages in that user's defaults database. The list of preferred languages is identified by the defaults key `AppleLanguages` and is stored in the global variable `NSGlobalDomain`. You can access that list using the `NSUserDefaults` class in Cocoa or the Core Foundation preferences functions.

Important: If you get the user language preference from the defaults database, you must get the canonical form using the `CFLocaleCreateCanonicalLanguageIdentifierFromString` function (in Mac OS X v10.4 and later) or `CFLocaleCreateCanonicalLocaleIdentifierFromString` function (in Mac OS X v10.3 and later) before using the identifier.

The following example shows you to get the list of preferred languages from the defaults database using Cocoa. The returned array contains the languages associated with the `AppleLanguages` key in the user's preferred order. Thus, in most cases, you would simply want to get the first object in the array.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

```
NSUserDefaults* defs = [NSUserDefaults standardUserDefaults];
NSArray* languages = [defs objectForKey:@"AppleLanguages"];
NSString* preferredLang = [languages objectAtIndex:0];
```

The locale for the current user is also stored in the defaults database under the `AppleLocale` key.

Important: Although you can get the user's preferred settings from the defaults database, it is recommended you use the `CFBundleRef` functions or `NSBundle` class instead. The associated functions and methods of those objects return the preferred language or locale that is also supported by your application. (Bear in mind that the returned values may not correspond directly with the user's exact preferences.)

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Preparing Your Nib Files for Localization

The native integrated development environment (IDE) for Mac OS X consists of Xcode, Interface Builder, and a suite of build, debugging, and performance tools. Developers use Interface Builder to create user interfaces for their applications. Interface Builder saves these interfaces as XML archives called nib files. You can localize nib files just as you can localize image and sound files.

Nib files store the user interface of your application, including windows, dialogs, and user-interface elements such as buttons, sliders, text objects, and help tags for these elements. A nib file also stores the connections between these objects that cause actions to be performed when the user activates controls.

> **Note:** This section talks about Interface Builder and its nib files. However, much of the discussion is applicable to localized user-interface archives created by other tools.

Using Nib Files Effectively

Translators typically localize the pieces of the user interface all at once, adjusting graphic elements to account for changes in string lengths. In any medium-size or large application, it's usually a good idea to put each window or panel (that is, dialog) in its own nib file. This practice not only makes it possible to load the user interface lazily (that is, to load pieces as they're used), but it also permits localization to progress in more incremental steps. It's also a good idea to put the menus of the application in a separate nib file.

Translators can use a combination of Interface Builder, AppleGlot, and `ibtool` to localize nib files. Using these tools, the translator would open the nib files in a given *language*`.lproj` directory, localize the strings, change the sizes of the user-interface elements to accommodate the new strings, and save the changes back to the nib files. There are a few other things to watch out for:

- Objects in a nib file typically have connections between them that should not be broken. Make sure you lock all connections before handing your nibs off to translation. For information on how to lock down your nib file, see Localization in *Interface Builder User Guide*.
- Dialogs and windows usually have minimum or maximum sizes that are specified through the inspector. If a dialog or window must be made wider for a given language, the translator should check to make sure that the minimum size is also updated to an appropriate value.
- Some user-interface objects support help tags—explanatory text that appears when the user moves the mouse over the object for a short period. You can define the help tags for an object in Interface Builder's inspector, where the translator can also localize those strings.

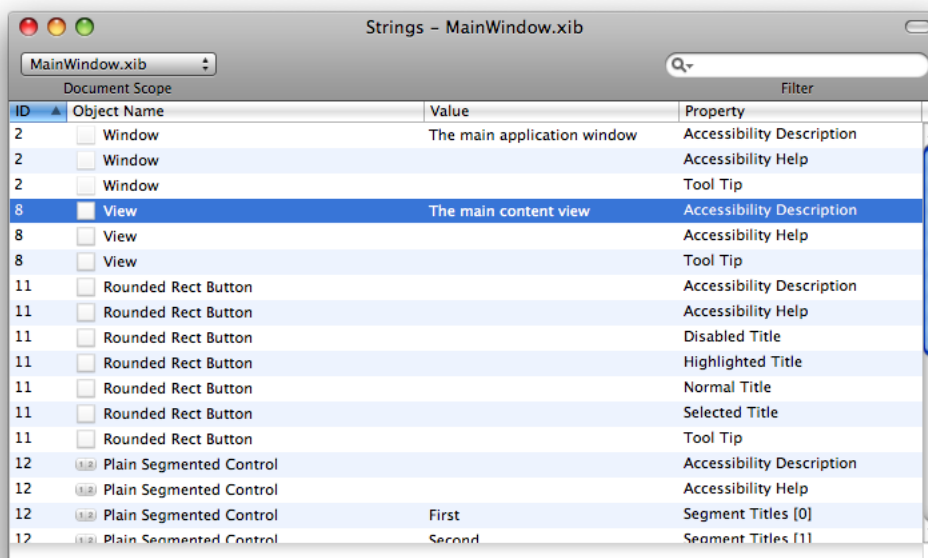
In your nib files, it is also important to remember that localization will likely change the size of visible text in your windows. If text labels do not have room to expand, the localizers may have to adjust the size of your window or the positions of the label or other controls. In general, you should expect text labels in English to expand by up to 40% in length during translation.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Viewing Localizable Strings in Interface Builder

In Interface Builder 3.0 and later, you can obtain a list of the strings in your nib file by choosing Tools > Strings. The window that appears (Figure 1) contains a list of the objects in your nib file along with entries for each property that contains a string that might be displayed to the user. The Value column shows the string (if any) that is set for each property when the nib file is loaded into memory.

Figure 1 Strings window in Interface Builder



In addition to viewing all of the strings in a nib file, developers and translators can also use the Strings window it to edit the string values. Double-clicking the value for an entry makes that entry editable. If you specify a new value, Interface Builder applies that value to the specified property.

For more information about using the Strings window, see *Interface Builder User Guide*.

Using ibtool

It can be tedious to translate nib files manually and even more tedious to propagate subsequent design changes to the already localized nib files in your project. Fortunately, the `ibtool` command-line utility makes propagating changes to other nib files much easier. This tool operates on a dictionary of all the strings in your nib file. Using it, you can extract the strings from the nib file, translate them, and then inject them back into the nib file. Alternatively, if you already have valid translations, you can inject the already-translated strings into a master nib file containing any recent design changes to update the localized nib files.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Note: The `ibtool` command-line utility replaces the `nibtool` utility that was present prior to Mac OS X v10.5.

The following example shows you how to extract the English strings from the main nib of a project into a new strings file. You can store a copy of the resulting file for use later or give the file to the translation team.

```
ibtool --generate-strings-file MainMenu.strings en.lproj/MainMenu.nib
```

The following example takes the original English-based nib file and merges it with the translated strings to create a localized version of the nib:

```
// The translated strings are in the de.lproj directory
//  with the same name as the original file.
ibtool --strings-file de.lproj/MainMenu.strings --write de.lproj/MainMenu.nib
en.lproj/MainMenu.nib
```

Important: Remember that translating the strings in your nib file is only one step of the localization process. The localizer may need to adjust the layout of views and controls to account for changes in string length. Also, `ibtool` does not automatically update date and time formats associated with formatter objects.

For more information about how to use `ibtool`, see Localization in *Interface Builder User Guide*.

Generating Localized Nib Files

Internationalizing applications for Mac OS X involves, in part, putting localized resource files in the proper location in your bundle. Fortunately, Xcode provides help with organizing your localized resource files. To internationalize a resource file in Xcode 3.1, do the following:

1. Add the resource file to Xcode.
    a. Expand your project in the Groups & Files pane.
    b. Select the Resources folder.
    c. Choose Project > Add to Project….
    d. Use the file browser to navigate to the resource, select it, and click Open.
    e. In the dialog that Xcode displays, select "Copy items into destination group's folder", choose one of the "relative" reference styles (for example, "Relative to Project"), and select the desired target. Click Add.
2. Select the resource file and open the Inspector window (File > Get Info).
3. If the file is not already localizable, click the Make File Localizable button in the General tab of the Inspector window.
4. In the General pane again, click the Add Localization button. In the sheet that appears, select or type the locale you want to add. Xcode copies the resource to the other `.lproj` directory.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

If your resource file is a plain text file, you should be sure to set the File Encoding for each localized copy. You can set the default localization of the file before you make the file localizable. You can use different encodings for different localized versions of the file. To set the encoding for a specific localization, choose that localization in the Groups and Files pane and change the setting in the Inspector window.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Localizing String Resources

An important part of the localization process is to localize all of the text strings displayed by your application. Strings located in nib files can be readily localized along with the rest of the nib file contents (as described in "Preparing Your Nib Files for Localization" (page 31)). Strings embedded in your code, however, must be extracted, localized, and then reinserted back into your code.

Resource files that contain localizable strings are referred to as **strings files** (with the deliberate extra 's' in the word "strings") because of their filename extension, which is `.strings`. You can create strings files manually or programmatically depending on your needs. The standard strings file format consists of one or more key-value pairs along with optional comments. The key and value in a given pair are strings of text enclosed in double quotation marks, separated by an equal sign, and terminated by a semicolon. (You can also use a property list format for strings files. In such a case, the top-level node is a dictionary and each key-value pair of that dictionary is a string entry.) Although the inclusion of comments is optional, they do provide a useful way to communicate contextual information to the translator about how each string is used. Listing 1 shows a simple strings file with two non-localized entries for the default language.

Listing 1 A simple strings file

```
/* Insert Element menu item */
"Insert Element" = "Insert Element";
/* Error string used for unknown error types. */
"ErrorString_1" = "An unknown error occurred.";
```

A typical application has at least one strings file per localization, that is, one strings file in each of the bundle's `.lproj` subdirectories. The name of the default strings file is `Localizable.strings` but you can create strings files with any file name you choose.

Note: It is recommended that you save strings files using the UTF-16 encoding, which is the default encoding for standard strings files. It is possible to create strings files using other property-list formats, including binary property-list formats and XML formats that use the UTF-8 encoding, but doing so is not recommended. For more information about Unicode and its text encodings, go to http://www.unicode.org/ or http://en.wikipedia.org/wiki/Unicode.

To make the maintenance of your code easier, there are tools in Mac OS X that you can use to extract the strings from your iPhone or Mac OS X applications and put them into strings files. For detailed information about how to create strings files using these tools, and how to prepare your code to use the strings those tools generate, see String Resources in *Resource Programming Guide*.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Internationalizing Other Resources

Some user-interface objects may localize their content automatically when they parse or format data. For example, if you have a Mac OS X text field configured with a number formatter, the formatter automatically accounts for the user's locale preference when formatting the number. However, other objects might require you to specify the current locale explicitly.

Resources and Core Foundation

If you are using the Core Foundation framework and need to format or scan date or number information, use the `CFDateFormatterRef` and `CFNumberFormatterRef` types. These types use the current locale information to ensure that dates and numbers are formatted correctly. For more information, see *CFDateFormatter Reference* and *CFNumberFormatter Reference*.

Resources and Cocoa

If you are formatting or scanning dates or numbers yourself using low-level objects such as `NSString`, `NSDate`, or `NSScanner`, you should use the current locale information if the resulting text will be seen by the user. The `NSLocale` class in the Foundation framework also makes it easier to get locale information. In addition to that class, several Foundation framework classes offer methods that provide an explicit locale argument:

```
NSString:
    - (id)initWithFormat:(NSString *)format locale:(NSDictionary *)dict, ...;
    + (id)stringWithFormat:(NSString *)format, ...;
    + (id)localizedStringWithFormat:(NSString *)format, ...;

NSScanner:
    - (void)setLocale:(NSDictionary *)dict;
    + (id)scannerWithString:(NSString *)string;
    + (id)localizedScannerWithString:(NSString *)string;

NSObject:
    - (NSString *)description;
    - (NSString *)descriptionWithLocale:(NSDictionary *)locale;

NSDate:
    - (id)initWithString:(NSString *)description calendarFormat:(NSString *)format
    locale:(NSDictionary *)dict;
```


2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

A locale is represented as a dictionary, using key/value pairs to store information about how the localization should be performed. Some of the possible keys in this dictionary are listed in `Foundation/NSUserDefaults.h`. Typically you pass either `nil` or the dictionary built from the standard user defaults. To create the dictionary from the user preferences, you would use the following code:

```
[[NSUserDefaults standardUserDefaults] dictionaryRepresentation]
```

This code returns a dictionary with a flattened view of the user's defaults and languages in order of user preference. The user's defaults take precedence, so any language-specific information might be overridden by entries in defaults (which are typically set from user's preferences).

The Application Kit framework makes localization easier by providing cover methods that ask for a “localized” version of an object, such as:

```
- (id)initWithFormat:(NSString *)format locale:(NSDictionary *)dict, ...;
+ (id)localizedStringWithFormat:(NSString *)format, ...;
```

The second method calls the first one with the dictionary representation of the user’s current defaults:

```
[[NSUserDefaults standardUserDefaults] dictionaryRepresentation]
```

In versions of these methods without a locale argument, the processing is done in a non-localized manner.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Localizing Pathnames

In order to provide a better user experience, Mac OS X supports the ability to display localized names for system and application directories. Localized pathnames make it easier for international users to find information, but do so in a way that does not impede your application. The implementation of this feature merely replaces the name of some directories with a localized version when they are displayed to the user. The actual path information on the hard disk does not change.

There are two ways to support localized pathnames. The first is to display them in your application. The second is to localize the names of any directories associated with your application.

The following sections explain the process for getting localized pathnames and for localizing your application's path information. For more information on localizing your application menus and content, see "Preparing Your Nib Files for Localization" (page 31).

Important: Localized pathnames are not supported in iOS. Mac OS X does not display localized pathnames for items in the Darwin and Classic environments.

Getting Localized Path Names

You need to be aware of localized path names in your application and display them appropriately. Localized path names are for display purposes only and should never be used when accessing the file system. You should continue to use the actual pathname when working with files and directories in your code, including when you need to write to caches or user preferences. The only time you should use a localized path name is when you want to display that path to the user through your application's user interface.

Mac OS X provides several functions for obtaining the localized name of a path. You should always use one of these functions to convert a path to its localized name immediately prior to display. All applications can use the Launch Services methods `LSCopyDisplayNameForRef` and `LSCopyDisplayNameForURL` to retrieve localized display names. Cocoa applications can also use the `NSFileManager` methods `displayNameAtPath:` and `componentsToDisplayForPath:` to retrieve this information.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Note: These functions do more than just localize path names. They also hide filename extensions when called for by the current file and Finder settings. Thus, they provide an abstraction between the actual file system and the file system as presented by the Finder. By using display names, your application helps to enforce this abstraction.

Localizing Your Application Name

If you have a bundled application, you can specify a localized display name for your application. The Finder displays localized bundle names based on the current language settings. Other applications can request your application's localized name as well and display it as appropriate.

Note: Mac OS X does not support localized names for non-bundled applications.

You specify localized names for your application using the existing bundle localization mechanism. The Resources folder of your application bundle contains one or more language-specific resource directories. (See "Bundle Structures" in *Bundle Programming Guide* for information about bundle resource directories.) In each of these language-specific directories, you can include an `InfoPlist.strings` file with a list of localized property-list keys. One of the keys you can include in this file is the `CFBundleDisplayName` key, whose value you can set to the localized name of your bundle.

At all times, Mac OS X prefers user-customized display names over the default and localized names you specify in your bundle. If the on-disk application name is different than the non-localized version of your bundle display name—that is, the name associated with the `CFBundleDisplayName` key in your `Info.plist` file—the system assumes the user made the change and returns the customized name. If at some later time, the user changes the application name back to the original name, the system reverts to using the localized values from your application bundle.

Important: If you want your localized display names to appear, you must include the `LSHasLocalizedDisplayName` key in your application's `Info.plist` file, set the type of its value to Boolean, and set the value to true. The functions that access localized display name information check for the existence of this key before retrieving the information.

Localizing Directory Names

If your application package installs any custom support directories, you can provide localized names for those directories; however, doing so is not required. If you want to localize your application's support directories, you should do so only for directories whose names are known in advance by your application.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Note: Localized directory names appear only if the "Always show file extensions" option is disabled in the Finder preferences. Localized names do not appear until the next time the user logs in.

To localize a directory name, add the extension `.localized` to the directory name. Inside your directory, create a subdirectory called `.localized`. Inside this subdirectory, put one or more strings files corresponding to the localizations you support. Each strings file is a Unicode text file. The name of the file is the appropriate two-letter language code followed by the `.strings` extension. For example, a localized Release Notes directory with English, Japanese, and German localizations would have the following directory structure:

```
Release Notes.localized/
    .localized/
        en.strings
        de.strings
        ja.strings
```

Inside each of strings files, include a single string entry to map the non-localized directory name to the localized name. For example, to map the name "Release Notes" to a localized directory name, each strings file would have an entry similar to the following:

```
"Release Notes" = "Localized name";
```

For information on creating a strings file, see String Resources in *Resource Programming Guide*.

Note: System-defined directories, such as `/System`, `/Library`, and the default directories in each user's home directory, use a different localization scheme than the one described here. For these directories, the presence of an empty file with the name `.localized` causes the system to display the directory with a localized name. Do not delete the `.localized` file from any of these directories.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Notes For Localizers

This task provides some assistance for localizers of strings files and nib files.

Localizing Strings Files

Strings files generated by `genstrings` have content that looks like this:

```
/* Comment */
"key" = "key";
```

That is, the value string is the same as the key string.

Translators should type the localized version of the key string in place of the second string. Use the comment, if necessary, to understand the context in which the string is displayed to the user:

```
/* Comment */
"key" = "French version of the key";
```

If a string contains multiple variable arguments, you can change the order of the arguments by using the "$" modifier plus the argument number

```
/* Message in alert panel when something fails */
"Oh %@! %@ failed!" = "%2$@ blah blah, %1$@ oh!";
```

Just as in C, some characters must be prefixed with a backslash to be included in the string properly. These characters include double-quote, backslash, and carriage return. You can also specify carriage returns with "`\n`":

```
"File \"%@\" cannot be opened" = " ... ";
"Type \"OK\" when done" = " ... ";
```

Strings can include arbitrary Unicode characters with "`\U`" followed by up to four hexadecimal digits denoting the Unicode character; for instance, space, which is hexadecimal 20, is represented as "`\U0020`". This option is useful if strings must include Unicode characters which cannot be typed for some reason.

Strings files are best saved in Unicode format. This allows them to be encoding-independent, and simplifies the encoding to use when an application loads strings files. The TextEdit application can save in Unicode format. The encoding can be selected either from the Save panel, or as a general preference in TextEdit's Preferences panel.

For more information about modifying strings files, see String Resources in *Resource Programming Guide*.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Localizing Nib Files

You use Interface Builder to create nib files, and you should use Interface Builder to localize nib files. Typically you open all of the nib files in a *language*.`lproj` directory, localize all the strings, change the sizes of UI elements if necessary, and save the nib files. There are a few things to watch out for:

- Objects in a nib file typically have connections between them that should not be broken. Make sure you lock all connections before handing your nibs off to translation. For information on how to lock down your nib file, see Localization in *Interface Builder User Guide*.
- Panels and windows usually have minimum or maximum sizes that are specified through the inspector panel. If you must make a panel or window wider for a given language, it's likely that the minimum size also needs to be modified.
- Some UI objects support “tool tips” – little blurbs that come up when the user hovers on the UI element for a short while. You enter these strings in the inspector, where they should also be localized.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

File Encodings and Fonts

Unicode is generally considered the native encoding for Mac OS X and should be used in nearly all situations. Previous versions of Mac OS supported file encodings such as MacRoman but most modern Mac OS X libraries support Unicode inherently. If you use Cocoa or Core Foundation routines, then you will probably never need to worry about other file encodings. If your software supports legacy file formats, however, you might need to consider file encoding issues when importing legacy file formats. The following sections describe some of the issues related to Unicode support and legacy file encodings.

File Systems and Unicode Support

Different file systems in Mac OS X have different levels of Unicode support:

- Mac OS Extended (HFS+) uses canonically decomposed Unicode 3.2 in UTF-16 format, which consists of a sequence of 16-bit codes. (Characters in the ranges U2000-U2FFF, UF900-UFA6A, and U2F800-U2FA1D are not decomposed.)
- The UFS file system allows any character from Unicode 2.1 or later, but uses the UTF-8 format, which consists mostly of 8-bit ASCII codes but which may also include multibyte codes. (Characters in the ranges U2000-U2FFF, UF900-UFA6A, and U2F800-U2FA1D are not decomposed.)
- Mac OS Standard (HFS) does not support Unicode and instead uses legacy Mac encodings, such as MacRoman.

Locking the canonical decomposition to a particular version of Unicode does not exclude usage of characters defined in a newer version of Unicode. Because the Unicode consortium has guaranteed not to add any more precomposed characters, applications can expect to store characters defined in future versions of Unicode without compatibility issues.

Note: Because of implementation differences, erroneous Unicode in filenames on HFS+ volumes may display correctly when entered on Mac OS 9 but appear garbled on Mac OS X. Similarly, erroneous Unicode entered on Mac OS X may appear garbled in Mac OS 9.

All BSD system functions expect their string parameters to be in UTF-8 encoding and nothing else. Code that calls BSD system routines should ensure that the contents of all `const *char` parameters are in canonical UTF-8 encoding. In a canonical UTF-8 string, all decomposable characters are decomposed; for example, é (0x00E9) is represented as e (0x0065) + ´ (0x0301). To put things into a canonical UTF-8 encoding, use the "file-system representation" interfaces defined in Cocoa and Carbon (including Core Foundation).

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Getting Canonical Strings

Both Cocoa and Core Foundation provide routines for accessing canonical and non-canonical Unicode strings. Cocoa string manipulations are all handled through the `NSString` class and its subclasses. In Core Foundation, you can use the `CFStringGetCString` and `CFStringGetCStringPtr` functions to obtain a C string with the desired encoding.

Cocoa Issues

Cocoa employs Unicode for character encoding, making any Cocoa application capable of displaying most human languages. Although Cocoa supports vertical and bidirectional text, the `NSTypesetter` class only supports layout for horizontal text. If you want to lay out vertical text, you need to define your own custom typesetter class.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Document Revision History

This table describes the changes to *Internationalization Programming Topics*.

Date	Notes
2009-08-07	Updated the information about the language-directory support for iOS.
	Removed the detailed information about string resources. That information is now available in *Resource Programming Guide*.
2009-01-06	Fixed the ibtool examples to use the updated syntax.
2008-09-09	Updated nibtool references to ibtool, which is the new name for the tool. Fixed some additional bugs.
2005-09-08	Merged the Cocoa Internationalization document into this document.
	Merged information from several articles aboug creating and managing strings files into one article.
	Added references to CFLocale and NSLocale.
	Updated the guidelines related to the use of language and locale IDs.
	Added information about how to use `nibtool` and `plutil`.
	Changed the title from "Internationalizing Your Software".
2005-03-03	Updated information on genstrings and usage of UTF-8 and UTF-16 in strings files.
2004-04-15	Added an article containing general guidelines. Moved the MultiScript guidelines to this article.
	Added an article on how to get the current localization information.
	Updated the language designation article to reflect the recommended use of ISO standards for designating language and region information.
	Fixed minor bugs.
2003-08-07	First revision of this programming topic. Some information in this programming topic originally appeared in *System Overview*.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

Index

A

AppleLanguages key 12
application bundle. *See* bundles
application name, localizing 40

B

BCP 47 specification 19
bundles
  applications 13
  guidelines 23
  plug-ins 13
  structure 13

C

canonical text encoding 45
CFBundleCopyPreferredLocalizationsFromArray function 27
CFCopyLocalizedString macro 24
CFDateFormatter opaque type 24, 37
CFLocale opaque type 13
CFNumberFormatter opaque type 24, 37
CFString objects 24
CFURL class 25
Core Foundation
  String Services 24

D

dialects, specifying 18

F

file systems
  localizing names 41
  organization

I

ibtool 32
icons 9
IDE. *See* integrated development environments
images, supported formats 14
integrated development environments 31
Interface Builder 31
internationalization 9
ISO 3166 18
ISO 639 17

L

language IDs
  and preferences 28
  canonical form 23, 28
  custom 21
  defined 18
  getting 28
  legacy support 21
language-specific project directories 20
locale IDs
  and preferences 28
  canonical form 23, 28
  defined 18
  getting 28
locale preferences 12
Localizable.strings file 35
localization
  defined 9
  directory names 40
  file names 41

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.

getting 27
`lproj` directories
defined 20
in bundles 14
nib files in 31

M

MacRoman encoding 24, 45
MLTE (Multilingual Text Engine) 25
multiscript support 24

N

nib files
in bundles 14
localizing 33, 44
tools support 31
translating 9
NSDate class 37
NSFileManager class 25
NSLocale class 37
NSScanner class 37
NSString class 37
NSTypesetter class 46

O

online help 9, 15

P

path display names 39
plug-ins. *See* bundles
`preferredLocalizationsFromArray` method 27

S

sound files 9
`strings` files 15
comments 43
localizing 43
overview 35
strings
overview 35
translating 25
system preferences 11

T

tablular data 9

U

UFS (UNIX File System) 45
Unicode 23, 24, 45
URL Services 25
UTF-16 encoding 45
UTF-8 encoding 45

W

word breaks 9

X

Xcode 31, 33

2009-08-07 | © 2003, 2009 Apple Inc. All Rights Reserved.