New Technical Notes

Macintosh

Developer Support

OV 20 – Internationalization Checklist

Overview

Written by: Norbert Lindenberg June 1993

This checklist discusses internationalization issues that may arise with various features that are common in Macintosh software. For each feature, it states which problems may arise, gives advice on what to do and what not to do, and refers you to additional reading material that can help you to implement a world-ready solution.

Topics

Introduction

This checklist discusses internationalization issues that may arise with various features that are common in Macintosh software. Some issues will show up in almost all software (for example, some font usage), others are less common. Still, this list cannot cover all possible features that your software may have. For the functionality that is specific to an application, the project team has to verify that all features are usable worldwide.

The "Why It Is Relevant" entry in each section is intended to give you an idea which internationalization problems you have to watch out for and to let you decide whether you have to read the rest. Other items show you good or bad designs of these features, give you some advice on what to do and what to avoid, and point you to further readings.

The document makes the following assumptions:

- that you have access to the *Text* volume of *Inside Macintosh,* second edition, and the revised system software interfaces that go along with it. Check your bookstore for the book or the June 1993 Developer CD for a DocViewer version, E.T.O. 11, or the June 1993 Developer CD for the interfaces.

- that you have read the chapter "Introduction to Text on the Macintosh" in *Inside Macintosh: Text* or some similar introductory material, so you understand terms such as *script system, primary script, system direction,* and *localization.*

- that your software uses TextEdit for all editing, text measuring, or text highlighting needs and does not install TextEdit hook routines. Doing these operations correctly for all script systems without TextEdit or with TextEdit hook routines is quite complicated, and discussing all the issues involved is beyond the scope of this checklist.

- that you write software that will run on System 7 or later. If you have to support System 6, the section "Running on System 6.x or 7.0" gives you some hints.

References

This checklist may refer you to the following documentation:

IM: *Inside Macintosh;* if a name follows, we are talking about a volume of the second edition, if a Roman number follows, about a volume of the first edition; published by Addison-Wesley.

MHIG: *Macintosh Human Interface Guidelines;* published by Addison-Wesley.

TN: Macintosh Technical Notes; available on the Developer CD and from many other sources.

WWD: *Macintosh Worldwide Development: Guide to System Software* (beta draft); available from APDA.

Creating a Generic User Interface

Why It Is Relevant

Ideally, to best serve Macintosh users worldwide and to maximize your revenues, you should localize your software products for all countries that Macintosh system software supports. In practice however, localization comes at a cost, and so most products get localized only for some countries, and some products (such as developer tools) don't get localized at all. Still, you should provide a minimally usable solution for the countries for which you don't localize. This way you can obtain some additional revenue from these countries, and you avoid disappointing power users worldwide who will often go to great lengths to obtain your software even if it is not available in their country.

This solution would be a version of your software that is fully functional on any international version of system software and uses a "generic" user interface that looks good on any version of system software. The user interface should probably use English, because that's the most commonly understood language among Macintosh users, but otherwise not make any assumptions about the system software localization. If you ship only one version of your software, and your software is not extremely country-specific (such as tax preparation software), it should be a generic version.

Negative Example

An application designed for the United States uses the alert text "The file "^0" could not be found.", and at run time uses ParamText to insert a filename. The byte sequence for the substitution pattern and the surrounding curly quotes is $D2 5E 30 D3. On Korean system software, ParamText checks for 2-byte character boundaries before substituting text. The byte sequence however has a different meaning in the Korean character set. It happens that $D2 can be the first byte of a 2-byte character, and so ParamText sees $D2 5E as a 2-byte character. There is no caret character in the text, so ParamText does not insert the filename, and the user never gets to know which file is missing. Using """ ($22) would avoid this problem.

Negative Example

An application dialog box has static text items whose rectangles extend beyond the right edge of the window. When the dialog box is shown on a right-to-left system, only part of the text is shown—the rest is clipped because the text is shown right-aligned in the rectangle.

Negative Example

Some products use the trademark symbols ™ and ® in filenames. The character codes will often be interpreted in a different way and displayed as gibberish on nonroman systems. Use of the ® symbol may also be fraudulent in some cases (see "Trademark Symbols" later in this Note).

What to Do

For user interface elements such as dialog boxes and menu items, as well as for filenames, use only those printing ASCII characters that are shared among all Macintosh character sets: $20–$5B and $5D–$7E, plus $11–$14 for system fonts ($5C is the slash [\] in most character sets, but the Yen symbol [¥] in the Japanese character set). The shared characters will display

correctly in any script. Here is a list of recommended substitutions for the most common non-ASCII roman characters found in U.S. applications:

Unicode Name	Character		Substitution		
bullet	•	$A5	*	$2A	
registered trademark sign	®	$A8	(R)	$28 52 29	
copyright sign	©	$A9	(c)	$28 63 29	
trademark	™	$AA	TM	$54 4D	
horizontal ellipsis	…	$C9	...	$2E 2E 2E	
nonbreaking space		$CA	space	$20	(or empty string)
en dash	–	$D0	-	$2D	
em dash	—	$D1	-	$2D	
double turned comma quotation mark	“	$D2	"	$22	
double comma quotation mark	”	$D3	"	$22	
single turned comma quotation mark	‘	$D4	'	$27	
single comma quotation mark	’	$D5	'	$27	

Where you have to use non-ASCII characters (for example, for the copyright message or your company name), display them in a way that lets you specify the font, for example, a picture. Make sure that the fontForce flag is turned off when the text is drawn—otherwise QuickDraw might use a different font (see “Disabling the fontForce Feature” later in this Note).

Design your dialog boxes so that they show a clean layout independent of the system direction:

- The appearance and alignment of some standard controls depend on the system direction. For example, radio buttons can have the actual button at the left or the right end of the control rectangle. To make a row of radio buttons always look clean, make all buttons equally wide and align them on both sides.

- Text items are by default aligned following the system direction. Make sure that their rectangles are fully contained in the window so that the text does not get clipped at the window edge. If you have control over the alignment (for example, when using MacApp views), make multiline English text left-aligned—even users of right-to-left systems tend to prefer this.

If you provide language-specific information or extensions with your software (such as spelling checkers for a word processor, or character conversion tables for a telecommunication product), make all of them available to users of the generic version in some form—either by bundling them or as readily available add-on products.

Define defaults in your software so that they serve the majority of the customers that you

want to reach with this version. For example, for the generic versions of Apple printer drivers, the default paper size is A4.

Don’t depend on any specific file or folder names in the system software environment. A localized application might get away with defining the name of the Preferences folder in a string resource, but a generic version has to use the FindFolder routine.

Test your software at least on Arabic and Japanese system software, and verify that everything looks and works fine.

If you provide an Installer with your product, again make sure that the Installer script does not depend on any specific file or folder names. See the “Installer Scripts” section below.

For a generic version that is separate from the U.S. version, Apple uses the language code “Z” in human-readable strings and the region code 37 in the 'vers' resources.

What to Read

IM Macintosh Toolbox Essentials 7-54–7-56 (for FindFolder).

Running on System 6.x or 7.0

Why It Is Relevant

Some of the international features documented in IM Text were introduced in System 7.0 or 7.1, and do not exist in earlier system software. Unfortunately, IM Text, which mainly documents the functionality available in System 7.1, does not tell you in enough detail which features were introduced at which time, and omits documentation for some older routines that you will need if your application has to run on older system software.

What to Do

Design your application using the information in IM Text—it’s the most coherent documentation of the international features that’s available, and its Appendix D is very useful if you want to modify existing code to take full advantage of System 7.1. Add the usual calls to Gestalt to verify that all special features that you need and that Gestalt knows about are available. Prime candidates for this are the Text Services Manager and the Dictionary Manager.

Then go back and check that everything that you need is actually available on all systems that your software has to run on. A future Technical Note may help you with this task.

Finally, of course, test your software on all system software versions that it is supposed to run on. Don’t forget to test with system software versions in other languages than your own —especially in System 6 there may be some surprising differences between, say, the Japanese and the Arabic versions of the same system software release.

What to Read

IM Text 6-6–6-9, 7-17, 8-11, D-1–D-7; WWD.

Working With Language Kits

Why It Is Relevant

Language kits contain the system extensions and fonts necessary to install a secondary script on any version of System 7.1. The Japanese Language Kit was the first one to ship recently. While language kits primarily deliver on the promise of WorldScript—to provide a true multiscript environment that lets users create multilingual documents—they come with a new

extension that slightly changes the programming model of WorldScript: the Language Kit Extension.

The purpose of this extension is to allow users to use applications that are localized for a language that uses a script other than the system script. For example, a user can run an application with a Japanese user interface on Russian system software, and have menus and dialogs of that application correctly display in Japanese.

To achieve this, the Language Kit Extension effectively defines an “application script” that is separate from the system script. It modifies the global variables that define the application font and the system font on a per-application basis. If the application script is correctly defined for the user interface language of an application, all text defined by the application and drawn using the system and application fonts is readable. Where application and system software work together to present user interface elements, for example, in print dialog boxes, the Language Kit Extension does some similar tricks to achieve the most readable interface possible.

Except for GetAppFont and GetSysFont, the Language Kit Extension does not affect the operation of the routines documented in *Inside Macintosh: Text.* In particular, it does not affect the decision as to which resources are by default used for generating date and time strings or for sorting in your application or the results returned from GetScriptManagerVariable and GetScriptVariable.

This behavior unfortunately invalidates an assumption that up to now was safe to make, and that many applications have relied on: that you can always use the system or application font to display text obtained from the Text Utilities routines with default parameters, and that your system and application fonts always belong to the system script.

The Language Kit Extension determines the script to be used for an application by looking at two sources: the application’s 'vers' resource and the Language Register database. The Language Register is a utility that comes with the language kits and that allows the user to specify the “language,” really the script, of an application. In case of conflict, the Language Register database is used.

Note: In spite of its name, the Language Kit Extension does not do anything about languages—it’s not a magic localization tool. It’s just that users understand the term *language* much better than *script,* and so we have language kits instead of script kits.

Negative Example

Some applications that are localized for Japanese check on launch whether the system script is Japanese, and refuse to run otherwise. Obviously, these applications are useless on a non-Japanese system with the Japanese Language Kit.

Negative Example

Some applications display date information about documents by using the obsolete IUDateString routine or by passing nil to the DateString routine, and then displaying the resulting text in the system font. If, for example, the application is localized to Japanese and runs on German system software, it would get German date strings and display them with a Japanese font, which sometimes results in gibberish: the Japanese font would replace the “är” in the long date string “Samstag, 7. März 1992” with a Kanji character.

What to Do

If you need to check for the presence of specific script systems, call `GetScriptVariable`(the`ScriptID`, `smScriptEnabled`).

Make sure that text that you get from the Text Utilities routines or pass to them, and the fonts that you use to display such text, match. Simple choices are to take everything from the system script or everything from the application script—the system script may be preferable because that's the only one whose formats the user can change with the Date and Time and Numbers control panels. For text that goes into documents you should let the user choose the script or language. You can find the application script code by calling `FontToScript`(`GetAppFont`) with the fontForce flag turned off (see "Disabling the fontForce Feature" later in this Note), for the system script you simply use smSystemScript. Once you have the script code, you would explicitly request resources and fonts for this script—for example, to display the short date:

```
theResourceID := GetScriptVariable(theScript, smScriptNumber);
theResource := GetResource('itl0', theResourceID);
DateTime(yourDateTime, shortDate, theDateString, theResource);
TextFont(GetScriptVariable(theScript, smScriptSysFond));
DrawString(theDateString);
```

In most places where you let the user edit text, you should allow for multistyled text or let the user choose the script or font to be used. A Find dialog box for a multiscript word processor is not very useful if it limits the user to search strings in the application script.

Include 'vers' resources in all your files, and use the correct region code for the region that the files are localized for.

If your application uses separate files containing resources that are logically part of the application, for example, optional Balloon Help files, make sure to use the application signature as the creator for all files. This lets the Language Kit Extension know that it should use the same script for user interface elements defined in the separate files as for those defined in the application itself.

If your application is multilingual, that is, contains user interface resources in multiple languages and at launch time determines which language to use for its user interface, you should look at the font returned by `GetAppFont` to determine whether the user has registered the application for a specific script. Unfortunately, there is no way to find out which language the user has really selected in the Language Register application, so you should use the default language of the application script if you support it. You can call `GetScriptVariable` (`FontToScript` (`GetAppFont`), `smScriptLang`) to find the default language (you have to call InitFonts first, because that's where the Language Kit Extension sets the application font). If you don't have resources for that language, you can either fall back to your generic user interface resources or let the user choose from a list of languages that use the application script. You can use the 'itlm' resource to find out which languages belong to which script.

What to Read

IM Text 4-54 (`GetAppFont`, `GetSysFont`), Script Manager (`FontToScript`, `GetScriptVariable`), B-12–B-17 ('itlm' resource).

Disabling the fontForce Feature

Why It Is Relevant

Since some applications have hard-coded fonts, a feature was introduced that allows system software to return information for a font belonging to the system script when an application actually requested information for a font belonging to the roman script. This affects many different operations, from the low-level `FontToScript` routine up to the QuickDraw `DrawText` routine. The feature is controlled by the Script Manager's fontForce flag. Users of some nonroman script systems can set or clear the flag using the Text control panel in System 7.1 or similar control panels in some earlier system software versions. If you use the Script Manager the way it is intended, but do not turn off the fontForce flag, you may get some surprising results (the descriptions of many routines were updated in IM Text so that they now match the results).

What to Do

You should disable the fontForce flag for your application, but not modify it for other applications. Note that the flag is a systemwide global variable, so you cannot just turn it off in your initialization sequence. You need two routines in your application: one that saves the current state of the fontForce flag and clears it, the other that restores the flag to its saved state. You must call the state-saving routine at application startup and immediately after each call to `GetNextEvent/WaitNextEvent`, and the restoring routine immediately before each call to `GetNextEvent/WaitNextEvent` and when quitting your application. If you call other toolbox routines that may in turn call `WaitNextEvent` (for example, AESend), you have to wrap them just like `WaitNextEvent`. This way, your application gets correct results from the Script Manager, and the fontForce feature can still have its effect on other applications.

What to Avoid

Do not leave the fontForce flag turned on while your application is running (assuming that your code uses the Script Manager as intended). Do not turn off the fontForce flag for other applications.

What to Read

IM Text 6-21–6-25.

Finding Fonts

Why It Is Relevant

When your application displays text, it has to use a font that a) is available on the system, b)

supports the character set that was used to write the text, c) is the one selected by the user or by the application if it's available (otherwise you need to find a substitute), d) can be found given the information that your application keeps about it. If you hard-code a font in your program (by name or by ID), or if you don't store enough information with your documents, you will not be able to meet all these requirements. The font you pick may not be available in all countries, or may use a character set that is inappropriate for the text that is being displayed. In addition to that, font IDs are unreliable because fonts may get renumbered during installation.

In addition to fonts, you may want to specify font sizes, styles, or alignment in your application. However, not all font sizes and styles make sense in all scripts: Characters smaller than 12 point can be very hard to read in some scripts, and underline is not supported for Japanese fonts. If you specify text alignment, it should usually depend on the script of the text.

What to Do

Use strategies that vary depending on where the text comes from. The following hints talk about fonts only; if you want to specify font sizes, styles, or text alignment, extend the recommendations logically.

- To display filenames or other text that comes from the operating system, use a font belonging to the system script (for example, the system font or the application font of the system script). Note that when you are running with a language kit installed, the default system and application fonts that you access by using the `systemFont` and `applFont` constants or the `GetSysFont` and `GetAppFont` routines may not belong to the system script—they belong to the application script, which may have been redefined by the Language Kit Extension. To get, say, the system font of the system script, you have to call `GetScriptVariable`(`smSystemScript`, `smScriptSysFond`).

- To display text that you obtained from Text Utilities routines or international resources, use a font whose script matches the script of the resource used. See “Working With Language Kits” earlier in this Note for more details.

- To display text that is defined in a resource in your application and for which you don’t rely on the system font, use another resource to define the font to be used so both can be changed during localization. Use both a font name and script information to refer to a font.

- To display text that the user enters, let the user pick the font. Use both a font name and script information to refer to a font when you save the information.

What to Avoid

Don’t use any font ID constants other than systemFont and applFont in your code (and these only if you are sure you want fonts of the application script—see “Working With Language Kits” for details). Don’t hard-code font names in your code.

Displaying Font Menus and Lists

Why It Is Relevant

In the WorldScript environment, fonts for several different script systems can be installed in a system. To display the names of these fonts correctly, you have to use a font of the same script, because often the font names contain characters unique to the script that the font belongs to.

What to Do

If possible, use a font menu that is created by calling `AppendResMenu`. This routine automatically adds script information to the menu items so that they are displayed in the system fonts of their respective scripts.

If you choose any other approach, your application is responsible for using the right font. You have to display each font name either in the font itself or in the system font of the font's script. For menu items, you can specify the script to be used by setting the keyboard equivalent field to $1C and the icon number field to the script code; the item will then be displayed in the system font of the specified script. To use the named font itself in a menu, you have to write your own menu definition procedure, and for font lists, you also have to roll your own.

For any approach, you should use `AppendResMenu` to create an initial list of fonts—this routine finds fonts of all types that exist on the Macintosh, and also screens out some names that are not intended for the user. See "TE 04 - Font Names" technical note for details.

If your application does not support editing in all script systems (for example, your text engine does not handle bidirectional text yet), you might consider screening out fonts that belong to a script that you don't support.

What to Read

TN Font Names.

Hard-Coding Coordinates

Why It Is Relevant

When your program is run on a different version of system software, text (especially in dialog boxes) may be displayed in a different font and in a different text layout, including different line breaks. Also, when the application is localized, the size of the text changes, again resulting in a different text layout. When an application is localized for right-to-left systems (currently Arabic, Hebrew, and Persian), the dialog box layout is usually completely redesigned to flow from right to left instead of from left to right. If you make assumptions about the text layout or the dialog box layout in your code, it will look strange in either case.

What to Do

Specify the layout of your windows in resources, and leave some room to grow even for unchanged text (the various system fonts do not all have the same metrics, not even for the ASCII characters). An even better, although somewhat ambitious, solution would be to have your application calculate locations and sizes of dialog items from a description of the

relative location of the items, the actual text to be displayed, and system information such as text measurements and the system direction.

What to Avoid

Don’t use constants in your code to determine the layout of your windows.

Hard-Coding Strings and String Lengths

Why It Is Relevant

Strings that are embedded in your code cannot be localized, as localizers usually do not have access to source code. During localization, string lengths will change.

Names of files and folders in the system environment may be changed during localization or by the user, so your application should not make any assumptions about them.

What to Do

Put all strings that may ever be displayed to the user into resources. Use strlen (C) or Length (Pascal) to find out about the byte length of a string, or a routine using `CharByte` or `CharacterByteType` to find out about the number of characters in the string. The number of characters may be interesting for the user (some word processors display character, word, and paragraph counts); Macintosh toolbox routines only care about the byte length.

To get access to files or folders in the system environment, for example, the Preferences folder or specific control panels, use the `FindFolder` routine or search for a file using its type and creator.

What to Avoid

Any constant strings in your code other than debugger messages, any constant string lengths, any assumptions about names outside your application.

What to Read

IM Macintosh Toolbox Essentials 7-54–7-56 (for `FindFolder`).

Concatenating Strings

Why It Is Relevant

If you concatenate two strings, you implicitly make three assumptions: that the strings are written using the same script system, that the order of the two strings in the resulting string is fixed, and that the forms of the two substrings do not need to change. These assumptions often do not hold when the strings are written in different languages than the one you had in mind. Different languages have different sentence structures and use inflection in different ways, and may have to be represented in different script systems.

Negative Example

Your application concatenates Undo and a command name such as Cut to obtain a more specific Undo Cut menu item. If the application then is localized to German, the concatenation reads “Widerrufen Ausschneiden,” which does not make sense—the reverse order, “Ausschneiden Widerrufen,” would be correct.

Negative Example

Your application concatenates an adjective and a noun in an application developed in English. In French, not only would the order be noun–adjective, but also the adjective would need to be adjusted to match the gender and the number of the noun.

What to Do

Store entire sentences in your resources whenever possible. If you need to insert some text that can be determined only at run time, for example, text that the user has entered or a filename, use `ParamText` or some other mechanism for string substitution to insert this specific text into an otherwise complete sentence. See the next section for details.

What to Avoid

Routines such as strcat, sprintf (C), concat (Pascal), CString::operator+ (MacApp).

String Substitution

What it is

Replacing a pattern in an otherwise complete sentence (for example, "^1" within "Could not find file "^1."") with actual data (for example, the name of the file your program could not find). The Dialog Manager's `ParamText` routine is one mechanism that lets you do this; the `ReplaceText` routine may be used in other contexts such as menus.

Why It Is Relevant

Using string substitution instead of string concatenation to construct dialog box messages is a good first step towards localizability. However, you have to be careful not to make assumptions about sentence structure and other syntactic peculiarities of languages. If you can put quotation marks around the text that you are inserting, you are probably safe. Otherwise, you have to consider carefully what the completed sentence would look like in various languages.

A special problem occurs if the text you are inserting is in a different script than the template that it is being inserted into. For example, you may want to tell the user that you could not find a Hebrew font that a document is using, and the error message template has been localized to Russian. Neither the Dialog Manager nor the MacApp views used for error message display can handle multiscript text. Therefore the Hebrew font name will usually be displayed as gibberish on any non-Hebrew system unless you write your own alert mechanism using multistyled `TextEdit`.

Positive Example

You want to tell the user that a file could not be found, and use the string “Could not find file ““^1.”” This makes it easy to localize the message, and also easy to insert the filename.

Positive Example

The Finder often has to insert numbers into strings, for example, “5 files and 1 folder are in the Trash for a total of 52K.” As the form of the nouns has to be adjusted to the numbers in ways that vary widely from language to language, the Finder uses a pattern language that lets a programmer or localizer specify which forms to use for 0, 1, or many. This makes it possible to display correct messages for most languages (unfortunately not for Russian, which has more number-dependent forms of a noun).

Negative Example

You define a string such as “^0 ^1 ^2.” in a resource and use it in your program to construct various different messages. This string is functionally equivalent to concatenation, so it usually causes all the problems mentioned in the “Concatenating Strings” section. Localizers may be able to correct the problem with the order of substrings, say, by changing the string to “^1 ^0 ^2.”, but only if they know all the messages constructed with this string and the new order applies to all of them. Usually, however, this is not possible.

Negative Example

You want to save some space and therefore split three similar messages into four partial strings, stored in a resource: “Caution! This disk is ^0.”, “locked”, “damaged”, “not initialized”. Substituting the last string into the first nicely results in “Caution! This disk is not initialized.” The Italian version of this however should read “Attenzione! Questo disco non è inizializzato.”, and this can not be achieved by simply translating the partial strings and substitution, because in Italian the “non” (not) goes before the verb “è” (is). If the strings are not used in other contexts, a localizer may fix the problem by moving the verb into the insertion strings, but this solution is not generally possible.

What to Do

Use ParamText or other substitution mechanisms only to insert filenames, font names, or text entered by the user into otherwise complete sentences. If you can put quotation marks around the substitution pattern, as in “Could not find file “^1”.”, you are usually safe.

If you implement a mechanism similar to `ParamText` to insert text into strings that are not shown in a dialog box, check for 2-byte characters when looking for the “^” character—the same byte value can also be used as the second byte of a 2-byte character.

If your text is in a handle and both the inserted text and the template are written in the same script, you can use the `ReplaceText` routine, which will automatically check for character boundaries; otherwise you have to use the `CharByte` or `CharacterByteType` routines to do it yourself.

If you insert text that may require changes to the surrounding text in any language, or if the

inserted text needs to be adjusted, you have to provide a way for localizers to describe the correct sentence forms. Typical candidates are:

- inserting numbers: It must be possible to adjust the nouns, for example, “0 files,” “1 file.”

- inserting nouns: It must be possible to adjust articles, adjectives, and verb forms to the number and gender of the noun. See the next item for an example.

- using pronouns: It must be possible to adjust the pronoun and related articles, adjectives, and verb forms to the number and gender of the noun that the pronoun refers to. An example for this and the previous item is the sentence “The ^0 could not be ^1 because it could not be found.”, with possible substitution strings “application” and “files” for “^0” and “opened” for “^1”. In French, you cannot just insert the corresponding nouns “application” and “fichiers” and the participle “ouvert” into a template “Le ^0 n’a pas pu être ^1 car il est introuvable.” Instead, you have to adjust the article “le,” the auxiliary verb “a,” the participle “ouvert,” the pronoun “il,” the verb “est,” and the adjective “introuvable” to arrive at the complete sentences “L’application n’a pas pu être ouverte car elle est introuvable.” and “Les fichiers n’ont pas pu être ouverts car ils sont introuvables.”

If the inserted text can be in a different script than the template (for example, you insert a font name or text taken from a multiscript document), you cannot use `ParamText` or `ReplaceText` for the substitution, and you cannot use the Dialog Manager for displaying the resulting string. In this case you have to use multistyled `TextEdit` to construct a text record with appropriate style information and to display it. In cases where the real script of the inserted text cannot be determined (for example, filenames), this additional effort may not make sense. Here users may have to live with an occasional display of gibberish.

What to Avoid

Do not use `ParamText` to construct sentences. Do not use Munger or similar routines that search byte-by-byte to perform text substitutions. Don’t insert font names or text entered by the user into template text for menu items—it is not possible to have multiscript menu items without writing your own menu definition procedure.

What to Read

IM Toolbox Essentials 6-46–6-48, 6-129–6-130; IM Text 5-74–5-75; IM Text—`TextEdit`; “Working With Individual Characters” later in this Note.

Truncating Strings

Why It Is Relevant

There are two very different situations where you may need to truncate text:
a) the data structures you use only allow for limited string lengths and you have to shorten user input to fit into the string length,
b) you want to draw a string into a field (on screen or on paper) of limited width.

In case a), you have to make sure that you copy only entire characters from the user input to the string in your data structures. For example, if your string only allows for 31 bytes, but

the user enters sixteen 2-byte characters, you have to truncate to fifteen 2-byte characters or 30 bytes. Truncating to 31 bytes would leave one meaningless byte at the end of the string.

In case b), you also have to watch out for 2-byte characters, but there are additional complications such as zero-width characters (often used for applied marks such as Thai vowels), which should never be separated from the preceding base characters, and ligatures, which are usually less wide than the individual component characters. Also, you should use the

appropriate character for indicating the truncation. For languages using the Latin writing system, "…" is commonly used, but other languages may use different characters.

What to Do

In case a), use the `CharByte` or `CharacterByteType` routines to check the last byte that would fit into your string, and remove it if it is the first byte of a 2-byte character.

In case b), use the `TruncString` or `TruncText` routines to truncate the text. These routines use the appropriate truncation indicator for the language being used, handle zero-width characters and ligatures appropriately, and at least for truncation at the end also handle 2-byte characters correctly.

Unfortunately there are some problems if you specify truncation in the middle: Truncated ASCII text on a right-to-left system may look strange because the ellipsis is a right-to-left character and so the text pieces are displayed in reverse order, and due to a bug in System 7.1, 2-byte characters are not always handled correctly. It is therefore better not to use the `truncMiddle` specifier.

If you want to write your own truncation routine, you should get the character to indicate the truncation from the default 'itl4' resource's untoken table using the `tokenEllipsis` selector.

What to Read

Case a): "Working With Individual Characters" below.

Case b): IM Text 5-71–5-74.

Working With Individual Characters

Why It Is Relevant

Some Macintosh character sets contain both 1- and 2-byte characters. Operations that deal with individual characters have to allow for 2-byte characters and make sure that they treat 2-byte characters as an entity.

What to Do

If you store individual characters, you have to provide space for 2-byte characters (you can use the `WideChar` data type). If you do character operations on strings, you have to check for character boundaries. If your application requires System 7.1 or later, you can use the new `CharacterByteType` routine for this, otherwise you have to use the older `CharByte` routine. `CharacterByteType` takes an explicit script parameter, while

CharByte requires you to set the font in the current `grafPort` corresponding to the script used for the text.

What to Avoid

Do not assume that a character equals 1 byte. Do not assume either that a character of a 2-byte script always needs 2 bytes.

What to Read

When requiring System 7.1 or later: IM Text 6-84–6-85.

Otherwise: IM V-306–307; WWD 142–143.

Sorting Strings

Why It Is Relevant

There are two very different purposes for which you may need to sort text: for internal data structures and persistent storage, or for display to the user. For the first purpose, you need a stable sorting order that allows for efficient access; the sorting order must not depend on the system software version your software runs on or on user selections. For the second purpose, you need a sorting order that makes sense to the user.

If your application allows the user to use multiple script systems (it should), you also have to take the script(s) used for each string into account.

Correct sorting according to the user's needs, especially in a multiscript environment, can be quite complicated. A discussion of trade-offs and possible solutions will be provided in a future Technical Note.

What to Read

IM Text 5-9–5-18, 5-54–5-63.

Editing Text

Why It Is Relevant

Most applications let the user edit text in dialog boxes or documents. The requirements for text editing behavior differ dramatically among script systems—an editing engine has to be able to delete 2-byte characters, move the caret in mixed 1- and 2-byte or in bidirectional text, make discontinuous selections, line-wrap text that does not contain space characters, display contextual text in the correct form, and more. Often text in a mixture of several scripts has to be supported.

What to Do

If at all possible, you should use `TextEdit` for the text editing needs of your application. `TextEdit` is not very fancy, but it works correctly with all scripts. If you use multistyled

`TextEdit`, users will be able to mix several scripts in one chunk of text, otherwise they will be limited to one script system.

If you choose to implement your own text engine, the documentation listed below will provide you with some, but unfortunately not with all the information you will need.

If you use `TextEdit`, but provide hook routines to extend its functionality, you have to make sure that the combination still works correctly with all scripts. This can be a difficult undertaking, and you need to have a good understanding of the issues, as if you were developing your own text engine.

Note: Two-byte characters that are input by the user will be passed to you through two consecutive calls to WaitNextEvent, with each call providing 1 byte in the event record. If you use TextEdit, simply pass them on to TEKey as if they were two 1-byte characters; TEKey will assemble the 2-byte character for you. If you develop your own engine, you should buffer the first byte and wait for the second before inserting the full 2-byte character into your document.

What to Avoid

Don't use a third-party text engine without verifying first that it works correctly with all scripts. Unfortunately, at this point, we are not aware of any third-party text engine available for licensing that fulfills this requirement.

What to Read

When using `TextEdit`: IM Text—`TextEdit`.

When implementing your own text engine: IM Text—Introduction to Text on the Macintosh, QuickDraw Text, Font Manager, Text Utilities, Script Manager; MHIG 267–303.

Inline Input

Why It Is Relevant

Inline input means entering and converting characters from a large character set (for example, Japanese) in the context of a document or dialog box instead of in a floating input window. Apple Japan now routinely requires inline input for new Apple software that is to be shipped in Japan.

What to Do

If your application uses `TextEdit` as its only text engine, only allows for unstyled text, does not offer Undo for text editing, and supports Apple events (at least the required ones), you can use TSMTE. TSMTE is an extension that provides Apple event handlers to handle all events passed on by the Text Services Manager and translate them into appropriate operations on the current TextEdit record. Using TSMTE, inline input can be implemented in about a day. Currently, TSMTE is available only with the full KanjiTalk 7 system, not with the Japanese Language Kit, but it will be integrated into future versions of system software —so make sure to check for its presence using Gestalt.

If the above conditions don’t apply to your application, you have to work with the Text Services Manager directly and provide your own Apple event handlers to communicate between TSM and your text engine.

In either case, make sure that your application passes on null events to TSMEvent, since some input methods rely on receiving them.

What to Read

TSMTE Technical Note (currently available only on the KanjiTalk Update CD); IM Text—Text Services Manager.

Passwords

Why It Is Relevant

For security reasons a password should never be displayed. When your software asks the user for a password, it should display a series of identical characters corresponding to the characters that the user has entered.

The Japanese, Chinese, and Korean scripts, however, need input methods for text input and conversion. If you do not implement inline input, input methods are handled by the floating input window, which will happily display the text as it is entered. If you do implement inline input, you can display the text as bullets, but still Japanese and Chinese input methods will occasionally display the text in pop-up menus to let the user choose one of several possible conversions (Korean is different in that conversion from Jamo to Hangul is deterministic).

Currently, two different approaches are used to solve this problem if the system script is a 2-byte script (don't apply either of them if the system script is 1-byte):

- Some software switches the keyboard to roman when asking the user for a password, thus encouraging the user to use an ASCII password instead of one in the native script. Users can still switch back to the native script using the keyboard menu and use a password in their native script. If your application uses inline input, users can also avoid having their password displayed by choosing text that does not have ambiguous conversions.

- Some software switches the keyboard to roman and locks it, so users cannot switch back to the native script, thus forcing them to use an ASCII password. This is the most secure solution, but some users don't like it.

What to Do

The character to be used to display the password should be obtained from the default 'itl4' resource's untoken table using the `tokenCenterDot` selector.

You will have to balance security concerns and user-friendliness in your decision how to handle input for users of 2-byte systems.

To switch and lock the keyboard, you use the `KeyScript` routine, first with the script code smRoman and then again with either the `smKeyDisableKybds` or the

`smKeyDisableKybdSwitch` selector.

You should not filter out any characters except control characters.

What to Read

IM Text 6-36–6-37, 6-61, B-54–B-55 (bullet), 6-17–6-19, 6-80–6-81 (`KeyScript`).

Text Searching

Why It Is Relevant

When searching a body of text for a given string, you have to be aware that the meaning of any given byte in the text depends on the character set being used for the text. Finding equal bytes is only one step in finding a matching string. If you offer options such as case-insensitive search or matching “similar” characters (for example, by stripping diacritics), you have to be aware of the script- or language-dependent meanings these options can take.

Negative Example

Your application searches for the character { in text containing Japanese, and selects a byte that is actually the second byte of a 2-byte Kanji character.

Negative Example

Your application searches for the character © in multistyled text, and selects the one-byte Katakana character having the same code.

Negative Example

Your application searches for the string “Mac” in multistyled text, and misses an occurrence because the user applied a Cyrillic font to it. This might happen if your implementation checks the script of style runs and only looks at the bytes in style runs with the same script code as the search string.

What to Do

Consider the script being used for each piece of text when deciding whether a sequence of bytes that matches your string actually has the same meaning. You have to make sure that you select byte sequences consisting only of entire characters (your text may contain both 1- and 2-byte characters). If either the body of text or the search string can be multiscript, or if they can be of different scripts, you should try to verify that equal bytes really represent the same characters in the scripts being used (this may not always be possible). If you offer case-insensitive searching as an option, you have to take the rules for the script system(s) in use into account—the `LowerText` routine helps with this. If you offer diacritic-insensitive

searching as an option, this feature depends on the language used—English users might consider “ä” equivalent to “a,” German users don’t. You may also want to offer the option to look for all possible representations of the same sound—for example, for Japanese, the Hiragana, 1-byte Katakana, 2-byte Katakana, 1-byte Romaji, and 2-byte Romaji representations.

What to Avoid

Don’t consider the Munger routine or Computer Science textbook algorithms more than a starting point. Don’t try do implement case-insensitive search by adding or subtracting constants to byte values, by using the C functions tolower/toupper, or by using the toolbox routines `UprText/UprString`.

Moving and Hiding Dialog Items

Why It Is Relevant

Localizers often have to adjust dialog box layouts to accommodate longer strings or to switch to a right-to-left system direction. To do this, they need access to all dialog items in a resource editor such as `ResEdit`. On the other hand, sometimes applications have dialog items that are shown only if some other option is selected, and a common way to hide dialog items is to move them outside the dialog box. If you do this in your resource definition however, graphical resource editors will not show the item.

A few applications also move dialog items from one location to another programmatically. This tends to make it difficult for localizers to adjust the layout, especially if you specify the new coordinates in your source code.

Note: With “move dialog items” I don’t mean the case that you recalculate the entire layout of a dialog box before showing it, taking all items, string lengths and the system direction into account. If you do that, great. Problems show up when your application moves some dialog items without respecting the overall layout.

What to Do

First of all, try not to move or hide dialog items. Both practices conflict with the human interface principle that an application should provide a stable environment. Making a dialog item inactive and showing it in gray is much nicer. But if you have good reasons to move or hide dialog items, read on.

When defining a dialog item that may be hidden at some point, put it at the location in the dialog box where it shows up when it is not hidden. Mark the dialog resource as initially invisible. Then, to open the dialog box, load it using `GetNewDialog`, check the current state of the relevant data and call `HideDialogItem` for all items that should not be shown, and finally show the dialog box using ShowWindow.

When moving dialog items, make sure that you get all necessary information from resources. For example, you can have additional items in the dialog box that are never shown but are used only to indicate the location that another item gets moved to. Or (not quite so nice), you can have separate 'nrct' resources that define the new item rectangles. In either case,

document well how you are interpreting the rectangles, and make the information available to the localizers.

Number Formatting

Why It Is Relevant

Number formats vary from region to region, and in some cases very different number formats are used in the same region depending on the context. System software defines a default number format for each region, and starting with System 7.1 lets users modify the default using the Numbers control panel. Your software should use the default number format defined by system software and the user, and for advanced number handling (for example, in spreadsheets), give the user the option to define and use specialized number formats.

Negative Example

Your application displays 123456 as “123,456” on a German system where the user has not modified the default setting (German uses the period as the default thousands separator and the comma as the decimal separator).

Negative Example

Your application does not accept the string “123,5” on this German system or interprets it as 1235.

Positive Example

Your application displays 123456 as “123.456” on a German system where the user has not modified the default setting.

Positive Example

Your application accepts the string “123,5” on this German system and interprets it as 123 and a half.

Positive Example

Your application lets the user choose a number format using format specification strings that have been adjusted to the user’s language and preferences (format specification strings are used in some spreadsheet and database applications and look like “###.###,##;(###.###,##);0”).

What to Read

IM Text 5-35–5-44, 5-91–5-101; *Apple Numerics Manual* (second edition) 26–27.

Currency Formats

Why It Is Relevant

Currency symbols differ, as does the preferred formatting of currency values. System software defines a default currency symbol and format in the system's default 'itl0' resource, and starting with System 7.1, users can change this default symbol to suit their own needs using the Numbers control panel. But your application should not assume that the default currency symbol is the only one the user ever needs—many companies import, export, or have

investments in other countries. Even individuals often have investments in and income from multiple countries.

Positive Example

Your spreadsheet offers the user the option to display numbers with currency symbols. It uses the default currency symbol from the system's default 'itl0' resource as the default, but lets the user pick a different one. It then uses the currency formatting information in the system's default 'itl0' resource to format the numbers.

What to Do

Use the default currency symbol defined in the 'itl0' resource as a starting point.

As with all text obtained from International Resources, you have to make sure that the resource and the font you use belong to the same script. See the "Working With Language Kits" section earlier in this Note for details.

You should use at least 32 bits for all currency values—16 bits are not enough even for small amounts of Yen or Cruzeiro. Double-precision floating-point numbers might be an even better choice.

What to Read

IM Text B-22–B-25.

Date and Time Formats

Why It Is Relevant

Date and time formats vary from country to country, and starting with System 7.1 users can define their own date and time formats using the Date and Time control panel.

Negative Example

Your application displays March 7, 1992, as "3/7/92" while running on a German system where the user did not change the date and time format (German users would expect 7.3.92).

Positive Example

Your application displays this date on German system software as "7.3.92", or as "Samstag, 7. März 1992" using a roman font.

Positive Example

In a document, your application lets the user choose which language’s date and time formats to use.

What to Do

Use the Text Utilities manager to create strings in the appropriate format, or to interpret date and time strings. Make sure that you provide enough space to accommodate four digits for the year even in the short date format.

As with all text obtained from Text Utilities, you have to make sure that the resource and the font you use belong to the same script. See the “Working With Language Kits” section earlier in this Note for details.

What to Read

IM Text 5-29–5-35.

Other Formats

Why It Is Relevant

Other formats, for example, for addresses and phone numbers, also vary from country to country. Therefore you cannot assume any particular format in your application. You can also not assume that one format per localized version of your application is sufficient, as many users will have business partners or friends in other countries.

What to Do

The easiest way out is to use free-form text fields that allow the user to enter arbitrary text as addresses or arbitrary strings of digits and separators for phone numbers. This method however does not allow you to interpret the data in any way, for example, for entry validation or for sorting addresses by region.

A more ambitious approach would be to use an extensible mechanism that allows the definition of several different formats for various countries and picks the right one based on country selection. Your application would probably supply some format definitions for the countries that your users are most likely to interact with, and allow the users to define more formats. This approach would allow your application to validate input data and to extract information for processing.

Paper and Envelope Sizes

Why It Is Relevant

Standard paper and envelope sizes vary among countries.

What to Do

If you develop a printer driver, or if your application supports customized paper sizes, specify all paper sizes and related format information in resources so that they can be modified, and allow for a variable number of paper sizes so that localizers can add or remove them.

Measurement Systems

Why It Is Relevant

Most countries in the world use the metric system (with millimeters, meters, kilometers), but a few countries still use the old English system of measurement (with inches, feet, miles). If your application displays any kind of measurement or any kind of ruler, it should default to the appropriate measurement system.

What to Do

Use the IsMetric function to find out whether your application should default to metric or English measurements. Except for very simple applications, let the user select the units to use—depending on the functionality of the application, you may offer cm, mm, inch, and add domain-specific units such as pica, point.

What to Read

IM Operating System Utilities (not available yet); IM I-505.

Custom Resource Types

Why It Is Relevant

Often resources need to be modified during localization, for example, to translate text, adjust window layouts, replace culturally specific icons with others. Localizers usually are nontechnical people who are not very comfortable with using Rez and DeRez to get at the content of your resources; they prefer tools such as AppleGlot, ResEdit, or Resorcerer. Therefore it helps with localization if resource types are used that are already supported by these tools, or for which templates for these tools can be easily created.

What to Do

For resources that might need to be modified during localization, use in order of preference: resource types that are already supported by these tools or resource types that can be described in the template formats supported by these tools. If neither is possible, develop a ResEdit picker or a different graphical editor for your resource type along with your application.

What to Avoid

Creating new resource types for resources that need to be localized without providing the necessary localization tools. Note that an extended MacApp 'view' or 'View' resource is a new type that needs specialized tools.

What to Read

AppleGlot 2.0 User Guide—Appendix F; *ResEdit Reference* for ResEdit version 2.1; *Resorcerer 1.0 User Manual.*

Menus

Why It Is Relevant

The width of the menu bar on the smallest monitors is 512 pixels, and that's not a lot. When designing your menus, you have to take into account that System 7.1 installs up to five icon menus (Apple, Application, Keyboard, Input Method, Help); some system extensions may add even more, and the titles of your own menus may expand substantially during localization.

Also, menus of course need to be defined in resources to be accessible for localization. If you change menu items dynamically (for example, from Show Clipboard to Hide Clipboard or from Undo to Undo Style Change to Redo Style Change), you have to ensure that a localized version of the application has equivalent localized menu items.

What to Do

Limit the number of menus in your application so that even with the maximum number of icon menus and a small monitor there is still room for the menu titles to grow during localization. If you have great difficulty reducing your menus to fit onto small screens, you might consider using abbreviated menu titles for small screens. In this case, make sure to define the abbreviations in resources so they are accessible for localization.

If you change menu items dynamically, the best solution is to define all possible strings in 'STR#' resources so that they can be easily localized. If you cannot avoid constructing them on the fly, read the "Concatenating Strings" and "String Substitution" sections earlier in this Note.

What to Avoid

Avoid adding commands to the system menu bar (for example, to the Finder or other applications). Several system extensions are doing this, but there is not enough room for all of them, and the mechanisms they use are unsupported and may break. Just don't do it.

Keyboard Equivalents (Command Keys)

Why It Is Relevant

Keyboard equivalents that you define for menus or dialog buttons (Command-key combinations) should be accessible on any keyboard worldwide. Keyboard layouts, however, vary considerably from region to region and among keyboard models. The good news is that you don't have to worry about nonroman characters here—whenever the user presses the Command key, the system switches to a layout that contains only roman characters. The bad news is that many characters are on different keys and in different layers

than you may expect, so a special character that is readily available on your keyboard may require several modifier keys on some other user's keyboard. This is complicated by the fact that in some older keyboard layouts the Shift key is ignored when the Command key is held down, in which case many characters are not available at all.

One problem that is now less common than it used to be involves Command-Period, which is used by many applications as an equivalent of Cancel or Stop. On some keyboards (for example, Italian), the period is produced by a key combination including the Shift key. Before System 7, the Shift key was usually ignored when the Command key was held down, so a

Command-Period combination could not be produced on these keyboards. Most keyboard layouts have been modified for System 7 so that now Command-Period is usually available, but there is no guarantee that this is true for all current and future keyboard layouts.

A somewhat different situation arises with Command-?, which some applications want to treat as a keyboard equivalent for Help. To type a question mark on a U.S. keyboard, the Shift key has to be pressed, so Command-? can not be produced on these keyboards (this has not changed in System 7). Therefore many applications accept Command-/ instead of Command-?. This however does not make sense to most non-U.S. users, because most non-U.S. keyboards have "?" and "/" on completely different keys. Often the "/" requires a Shift key as well, so Command-/ is not any more accessible than Command-?.

All keyboard equivalents involving the space bar are reserved for use by the Script Manager.

What to Do

In general, try to use only alphabetic characters (A–Z) as predefined keyboard equivalents. It seems that these are available with the Command key on all keyboards. If 26 keyboard equivalents are not enough for your application, you probably should consider making keyboard equivalents user-definable instead of hoping that your users have the same keyboard as you.

If you support Command-Period for canceling (which has become a standard and so should be supported in spite of its potential problems), or if you have to use Command-Question-Mark, you have to make sure that you recognize them even if they require the Shift key. Whenever the user presses a key combination involving the Command key while your application is in a state where Command-Period or Command-? would make sense, check whether the same key combination without the Command key would translate to a period or question mark. The "TE 23 - International Canceling" tech note explains how to do this. If you really want to provide an alternative key that can be used as an equivalent for keys that require the Shift key (for example, "/" for "?"), define them in a resource so that they can be changed during localization. Much more useful however than providing an equivalent for the equivalent is to make the original command accessible, so make sure that your application always has readily accessible menus or buttons for Cancel, Stop, or Help.

What to Avoid

Don't change keyboard equivalents for localized versions of your application. As most commercial Macintosh software is published in the United States first, it is quite common for users in other countries to use newly released U.S. software temporarily and later switch to a localized version when it becomes available—and in some cases this never happens, so they continue to use U.S. software. As a result, users often run U.S. and localized software at the same time. It would be very confusing if the keyboard equivalents changed whenever they switch from one application to another, or when they switch from the U.S. to a localized version. Keyboard equivalents help only if users don't have to think about them.

What to Read

MHIG 128–129; TN International Canceling.

Graphics

Why It Is Relevant

Graphics are often culturally dependent. Graphics that are meaningful or funny to a French user may be meaningless or offensive to an Arabic user, or vice versa.

Text that is embedded in graphics usually has to be localized.

What to Do

Try to use culturally neutral graphics. Put the graphics into resources that can be changed during localization if necessary.

If you need to have text in the graphics, consider using separate text items instead that you get from string resources and draw over the graphics (but make sure that the coordinates and the font information are also obtained from resources so they can be changed if necessary). If that's not possible, make sure to use object-oriented graphics (for example, drawn with MacDraw) instead of bitmaps (as produced by MacPaint). This makes it much easier to change the text. Use only standard fonts that are available on every Macintosh that your application is likely to reach, and avoid special effects that may not make sense in other languages.

Sound and Voice

Why It Is Relevant

Similar to graphics, sound can be culturally dependent. Again, sounds that are meaningful or funny to users in one country may be meaningless or offensive to users in another.

Sound names (for example, system beeps) need to be localized.

Voice annotations need to be localizable just as other recorded sound, but you have to watch out for additional problems if the voice annotation can be considered part of your user interface (as opposed to document content). For example, Japanese users prefer computers to use a female voice.

What to Do

Put all recorded sound into resources or separate QuickTime sound tracks that can be easily replaced during localization.

What to Avoid

Do not generate sound directly from your program code unless you have verified that the sound is culturally neutral. Do not assume that the name of a sound is the same in all countries.

Installer Scripts

Why It Is Relevant

When looking for files or folders on the disk that you are installing onto, you cannot assume any specific names, because names are often changed during localization or by the user. Also, the technique that is usually recommended to allow for localization—putting strings into separate resources—does not work here, because many products are shipped in a generic version that can be installed into system software of any language.

Negative Example

Some Installer scripts try to replace old versions of control panels with newer ones, but look for them by name. If a generic version of such a script is used to update a localized system, it may not be able to find and delete the old control panel because it has a localized name, but it may still install the new version. Having two versions of the same control panel in the System Folder may have undesirable side effects.

What to Do

If you need access to the System Folder or folders inside the System Folder, use the Installer special names (for example, special-extn for the Extensions folder) or call the `FindFolder` routine from your external functions. If you need to check for or delete specific files on the disk, use a user function in which you search for them by type and creator, not by name.

Note: Using the Installer special names is actually somewhat tricky because the Installer relies on 'fld#' resources and looks for them in several places. The search order of the current Installer, version 3.4.2, is Installer script, target system, Installer, booted system. What you usually want is the target system, so at least for a generic Installer script you should not have an 'fld#' resource in the script.

What to Avoid

Don’t use names of files or folders on the target disk directly in your Installer scripts. Don’t put the names into separate string resources either. Get them from system software or use a user function.

What to Read

Installer 3.3 Scripting Guide 36; IM Macintosh Toolbox Essentials 7-54–7-56 (for FindFolder).

Encryption

Why It Is Relevant

Encryption is subject to legal restrictions in some countries. For example, U.S. law usually makes it illegal to export software that uses any serious encryption technique, no matter where it was invented and how well published and used the technique already is outside the United States. Capable lawyers may be able to obtain an export permission.

What to Do

Contact your lawyers if you consider using encryption in your software.

Trademark Symbols

Why It Is Relevant

You can use the symbol ® for a trademark only in countries where the trademark has been registered—use in other countries is fraudulent.

What to Do

Use the ® symbol only for names and on material for which it is known that it will reach only countries where the name has been registered. On materials that are shipped worldwide (for example, packaging, manuals, and so on) or for which the ultimate destination is not always known, don't use the ® symbol. Instead, use a "universal" credit line such as Apple's "Apple and the Apple logo are trademarks of Apple Computer, Inc., registered in the U.S. and other countries."

You can freely use the ™ symbol, as it has no legal effect.

Acknowledgements

This checklist would not have been possible without all the people who taught me everything, gave me all the examples, asked questions, answered questions, and reviewed the document: Andrew Wilson, Akira Nonaka, Chris Hansten, Dave Bice, Debbie Gordon, Doug Scott, Eric Chen, Eric Soldan, Gidi Shalom-Bendor, Hani Abdelazim, Jay Schaffer, Jean-Pierre Ciudad, Jeannette Cheng, Jennifer Han, Joan Trainer, Joel Cannon, John Harvey, John McConnell, Joseph Maurer, Kenny Tung, Kerry Laidlaw, Lorenzo Sangalli, Maha Hassan, Mark Zeren, Michael Silver, Mimi Obinata, Mina Noroña, Mohamed Shoukry, Peri Altan, Peter Edberg, Ron Metzker, Sue Bartalo, Susan Torres, Vichai Lelapinyokul, Yishai Steinhart, and many others. If, in spite of all their help, this document still contains errors, that's of course my fault.

Further Reference:

- *Inside Macintosh: Text,* Introduction to Text on the Macintosh
- *Inside Macintosh: Text,* TextEdit
- *Inside Macintosh: Text,* QuickDraw Text
- *Inside Macintosh: Text,* Font Manager

- *Inside Macintosh: Text,* Text Utilities
- *Inside Macintosh: Text,* Script Manager
- *Inside Macintosh: Text,* Text Services Manager
- *Inside Macintosh: Text,* International Resources
- *Inside Macintosh: Macintosh Toolbox Essentials,* Menu Manager
- *Inside Macintosh: Macintosh Toolbox Essentials,* Dialog Manager
- *Inside Macintosh: Macintosh Toolbox Essentials,* Finder Interface

Further Reference (continued):

- *Inside Macintosh,* Volume I, The International Utilities Package
- *Inside Macintosh,* Volume V, The Script Manager
- *Macintosh Human Interface Guidelines,* Menus
- *Macintosh Human Interface Guidelines,* Behaviors
- Macintosh Technical Note: “TE 23 - International Canceling”
- Macintosh Technical Note: “TE 04 - Font Names”
- Extension: TSMTE
- *Macintosh Worldwide Development*: *Guide to System Software* (beta draft)
- *Apple Numerics Manual,* second edition
- *AppleGlot 2.0 User Guide,* Appendix F
- *ResEdit Reference* for ResEdit version 2.1
- *Resorcerer 1.0 User Manual,* Mathemæsthetics, Inc.
- *Installer 3.3: Scripting Guide*