Localizing to Other Languages and Regions

You can use existing resources or create new ones to localize your applications to other languages or regions. The sections that follow contain guidelines and tips for localizing your applications successfully. See *Guide to Software Localization* for details on the localization process.

Using Resources

Be certain that you place the following information in an appropriate resource:

- all user-visible text
- lengths of string and text
- dialog box formats
- menus and command keys
- character, word, phrase, and text translation tables
- font family numbers, names, sizes, styles, and widths of numeric fields
- special characters and delimiters

Note: Other information may vary from locale to locale such as address formats, including zip codes and telephone numbers.

Text and Dialog Translation Tips

Follow these guidelines for translating visible text in your applications to other languages:

- Do not place in the program code any text that the user will see.
- Do not assume that all languages or regions have the same rules or conventions for punctuation, word order, and alphabetizing.
- Be aware that grammar problems may arise from error messages, natural programming language structures, and so forth.
- Text needs room to grow-up, down, and sideways! (This is especially true for numbers.) For example, translated text data can be 50 percent larger than the U.S. English text data; therefore, do not rely on strings having a particular length.
- Text location within a window should be easy to change.
- Note that the arrangement of dialog-box items may vary with localization. The low-memory global variable TESysJust indicates the system line direction. On Arabic and Hebrew systems, the line direction usually defaults to right to left, so text in dialog boxes should generally be right to left. When creating a column of check boxes or radio buttons, make sure the text boxes are the same sizes. Then when the line direction is reversed, the check boxes or radio buttons align

correctly.

> **Note:** Natural programming language structures refer to programming languages that attempt to use human-like command structures. For example, it is unreasonable to write a command in one language and expect it to survive the process of translation without making accommodations for word order at the very least. An English command like "Put It Into Field 7" could translate into Japanese word order as "It (as-for) Field 7 Into Put."

Adapting Text Operations

Follow these recommendations for adapting text operations to other languages and regions:

- Use the **Script Manager** routines for displaying and measuring text and testing for mouse-down events in text, or use **TextEdit**, which now uses the **Script Manager** for these operations.
- Use language-specific string comparison and sorting. For example, be sure that names are sorted using the correct sorting order for the system on which your application is running. To do this, use **IUCompString** or a similar function.
- Use language-specific word-break and word-wrap routines. Consider word boundaries and their impact on word wrap, selection, search, and cut and paste. Use the **FindWord** or **NFindWord** procedure to specify word breaks.
- Use language-specific character type information. Consider character boundaries and their impact on search, replace, sort, word wrap, backspace, delete, and cut and paste. Use the **CharType** function and its associated constants to obtain more information about character type. See **Script Manager Data** for details on the **CharType** constants.
- Use language-specific case conversion. Use the **UpperText** and **LowerText** procedures for localizable uppercase and lowercase conversion of characters.
- Keep in mind that the length of text may vary from language to language.
- Avoid assumptions about the number of letters in the alphabet. For example, if your program relies on properties of the ASCII code table or uses data compression codes, remember that not all alphabets have the same number of characters.
- Do not break text into arbitrary blocks for drawing, measuring, and so forth. As a minimum, always group text on the same line into style runs for drawing and measuring.

Using Fonts

Here are some hints for planning for font-related issues in your applications:

- Diacritical marks, used in many languages, extend up to the ascent line.

- Some system fonts contain characters that extend to both the ascent and descent lines.

Avoiding Special Character Codes as Delimiters

Your application may need to use a character code or range of codes to represent noncharacter data (such as field delimiters). Character codes below 0x20 are never affected by the script system. Some of these can be used safely for special purposes. Note, however, that most characters in this range are already assigned special meanings by parts of the Macintosh Toolbox, such as **TextEdit**, or certain programming languages like C.

The low ASCII characters (with hexadecimal representations) that you should avoid as delimiters are as follows:

Character or key	**Code**	**Character or key**	**Code**
Null	0x00	Page Up	0x0B
Home	0x01	Page Down	0x0C
Enter	0x03	Carriage return	0x0D
End	0x04	F1 through F15	0x10
Help	0x05	System characters	0x11, 0x12, 0x13, 0x14
Backspace	0x08	Clear	0x1B
Tab	0x09	Arrow keys	0x1C, 0x1D, 0x1E, 0x1F
Line feed	0x0A		

Using the Standard Roman Character Set

Be aware that the "traditional" Macintosh character set (that is, the original set described as the Macintosh character set) stops at code 0xD8 and contains a limited set of European accented forms. The standard Roman character set now includes the remaining character codes (0xD9-0xFF); it supplies uppercase versions of all of the lowercase accented forms in the traditional set, new symbols, and other forms. These characters are available in most LaserWriter and TrueType fonts, but not in the Apple bitmap versions of Chicago, Geneva, or Monaco. See the **Font Manager** for an illustration of the standard Roman character set. See *Macintosh Worldwide Development: Guide to System Software* for further information on the standard Roman character set (there referred to as the extended Roman character set).

Since System 6.0.4, there has been full support for the standard Roman character set. This version has supplied more completeness and consistency in the handling of accented forms in the fonts that contain these forms.

Also with System 6.0.4, the U.S. keyboard resource 'KCHR' (0) has been modified to make it possible to enter the accented forms with dead keys. Users can enter all the accented forms in the original Macintosh character set with dead keys.

> **Note:** With System 7.0, the keyboard entry of the following four characters is now possible using the U.S. 'KCHR' resources: cedilla (¸), double-acute accent (˝), ogonek (˛), and hacek (ˇ). These characters are represented by codes 0xFC-FF.

The tables used by the **Script Manager**'s **CharType**, **Transliterate**, and

FindWord routines (in 'itl2'), by the **IntlTokenize** function (in 'itl4'), and by the **International Utilities Package** (in Pack6) have been modified in the U.S. system for proper treatment of the character-set extensions as well as the traditional characters ß, ø, and Ø.

Adapting Keyboard Equivalents

Applications that make extensive use of keyboard equivalents face numerous challenges on the worldwide market. When multiple script systems are installed on a system, the challenges increase. When the Command key is pressed, some characters, such as the period (.), cannot be produced on certain keyboard layouts. This is mainly a problem if symbols are used as keyboard equivalents. To make Command-key handling work in these cases, it may be necessary to determine which character would have been produced if the Command key were not pressed. The code example below illustrates one approach.

Code example:Making keyboard equivalents work with multiple scripts

```
// Assuming inclusion of <MacHeaders>

#include <Script.h>

#define menuIDMask       0xFFFF0000
                        // mask for menu ID in MenuKey result
#define newModifierMask     0xFC00
                        // high byte of modifiers, without cmdKey bit
#define Ascii1Mask       0x00FF0000
                        // ASCII 1 in Key Trans result
#define Ascii2Mask       0x000000FF
                        // ASCII 2 in Key Trans result

main ()
{
    EventRecord   myEvent;
    char          myChar;
    long          menuResult;
    short         myNewModifiers;
    short         myVirtualCode;
    short         myKeyCode;
    Ptr           myKCHRPtr;
    long          myDeadState;
    long          myNewChars;
    char          myNewChar;

    // Assume that here is a key-down or auto-key event.

    myChar = myEvent.message & charCodeMask;
    if (myEvent.modifiers & cmdKey) {       // Command key is down.

        if (myEvent.what == keyDown) {

            menuResult = MenuKey(myChar);

            if (!(menuResult & menuIDMask)) {
                // Didn't match, so see if there is a match with
                //  the character that would have been produced if
                //  Command were not down.

                // First, make keyCode parameter for KeyTrans,
                //  but turn off the Command key bit.

            myNewModifiers = myEvent.modifiers & newModifierMask;
            myVirtualCode = (myEvent.message & keyCodeMask) >> 8;
                myKeyCode = myNewModifiers | myVirtualCode;

            // Now, get current 'KCHR' pointer. This requires version 7.0 or                t
// later.

                myKCHRPtr = (Ptr) GetEnvirons(smKCHRCache);

                // Now set dead state to 0 and call KeyTrans.

                myDeadState = 0;
                myNewChars = KeyTrans(myKCHRPtr, myKeyCode,
                        &myDeadState);

                // If there is a nonzero result in the high word, try it;
                //  else if there is a nonzero result in the low word,
                //  try it.

                myNewChar = (myNewChars & Ascii1Mask) >> 16;
                if (!myNewChar)
                    myNewChar = myNewChars & Ascii2Mask;
                if (myNewChar)
                    menuResult = MenuKey(myNewChar);
            // Note that the menu ID field of menuResult may still be 0.

            }
            // Now do menu handling based on menuResult.
        }
    }
    else {
        // Not a Command key; do other handling.
    }
}
```

Many applications extend the set of standard Macintosh interface modifier-plus-key combinations for specific purposes. Be sure to supply alternative methods of gaining access to functions. Avoid keyboard equivalents that use the Space bar in combination with the Command key and other modifier keys. The **Script Manager** and various script systems reserve and may use these combinations.

> **Note:** The **Script Manager** removes from the event queue any Command key combinations involving the Space bar if that Command key combination indicates a useful function on the current system. For example, if multiple script systems are installed, the **Script Manager** strips the Command-Space bar combination (which indicates changing scripts) from the event queue. If multiple

script systems are not installed, this event is not removed, so users can use it in Command key macros. Applications, however, should never depend on Command key combinations involving the Space bar.

Modifying the Representation of Dates, Times, and Numbers

Be sure to allow for variations in the representation of dates, times, and numbers in all the localized versions of your application.

- Dates, times, and numbers should be displayed in local format as specified by the system script's 'itl0' and 'itl1' resources.

- Parsing of dates, times, and numbers should work for local formats (the **Script Manager** provides routines to do this). See the sections, **Date and Time Utilities** and **Number Utilities** for details.

- Units of measure should be localized. For example, lines per inch is meaningless for metrics. **International Utilities Package** provides routines for determining the appropriate units of measurement.