Handling Unicode Text Editing With MLTE

Text & Fonts > Carbon

2008-10-15

Contents

Figures and Listings

MLTE Introduction

Multilingual Text Engine (MLTE) is an application programming interface (API) that allows your application to provide Carbon-compliant Unicode text editing. MLTE is a replacement for TextEdit that offers more features than those in TextEdit—features such as document-wide tabs, full justification of text, support for more than 32 KB of text, built-in scroll bar handling, built-in printing support, support for inline input, support for the advanced font features of Apple Type Services for Unicode Imaging (ATSUI), and support for multiple levels of undo.

You can use MLTE to replace TextEdit functions in your existing applications. With it, you can significantly reduce the number of lines in your code because MLTE handles most of the low-level tasks you had to code in the past.

MLTE provides a quick and easy solution for static display of Unicode text and for creating Unicode-compliant text-editing fields within an application. You can also use MLTE if your application needs to provide text editing support within a full-size window.

For information about ATSUI (which MLTE supports), see *ATSUI Programming Guide*.

MLTE Concepts

This chapter provides an overview of Macintosh text handling and how MLTE in particular handles Unicode text editing. If you are new to text handling on the Macintosh, you should read the entire chapter. If you have already worked with international text on the Mac OS and are familiar with text handling, you can skip the first two sections and go to "How MLTE Handles Text" (page 10).

This chapter covers concepts rather than implementation or programming details. For information about using MLTE functions in your application, and to see sample code, see "MLTE Tasks" (page 25).

Introduction to International Text on the Mac OS

This section defines the terms used to discuss international text handling on the Mac OS. If you would like a more detailed introduction to typography on the Mac OS as well as more information on Unicode and related topics, see *ATSUI Programming Guide*.

Characters, Character Encodings, and Unicode

A **character** is a symbolic representation of a letter, a number, a punctuation mark, or any other mark used in text; it is the concept of, for example, "lowercase a" or "number 3."

In computer memory, text is stored as **character codes**, where each code is a numeric value that defines a particular character. A **character encoding** is the organization of the set of numeric codes that represent all the meaningful characters of a **script system** in memory. There are two fundamental classes of Mac OS character encodings: single-byte and double-byte.

A **writing system**is a set of characters and the basic rules for using them to create a visual depiction of language. Examples of writing systems are Roman, Japanese, Arabic, and Hebrew. **Unicode**is an international standard that combines the characters for all commonly used writing systems into a single, coded character set, based upon a 16-bit character encoding standard. With a universal character encoding such as Unicode, the character sets of separate writing systems do not overlap. Furthermore, Unicode resolves the issue of conflicting character encodings within a single writing system.

Keyboards and Input Methods

The user typically enters text through a keyboard that your application then stores as character codes. The system reports the user's key-down, key-up, and auto-key events to your application. **Key-down** and **key-up events** indicate the user pressed or released a key, respectively. **Auto-key events** indicate the user has held a key down for a certain amount of time. For keyboard-related events, your application receives both the virtual key code and the character code for the key that is pressed, as well as the state of any **modifier keys**(For example, Shift, Caps Lock, Command, Option, and Control) at the time of the event.

For languages with large character sets, it is impractical to manufacture keyboards with keys for every possible character. In such a case, it is usually the job of an input method, working in conjunction with a keyboard, to handle text input. An **input method** is a software module, often independent of the application it serves, that performs complex processing of text input, prior to the application's processing of the text. A typical example of an input method is a translation service that converts character codes that can be entered from the keyboard into character codes that cannot; text input in Japanese, Chinese, and Korean usually requires an input method.

Overview of Text Handling

An application with simple text editing capabilities has to do several tasks. First of all, the application must get text input from the user. The application must be prepared to convert character sets in the event a user pastes text from a document created on another platform, the Internet, or from any document that uses a character set different from that of the application's default **script**.

As the user enters text, the text must be manipulated and stored. **Text manipulation** refers to system-level procedures used to sort and compare characters, determine line breaks, determine **text directionality**, and keep track of character properties, such as case. Calculating line breaks is a difficult text manipulation task for scripts that do not delineate words with spaces, such as the Thai script. **Display order** is complicated for scripts that are **bidirectional**, such as Hebrew. For example, in Hebrew, words are displayed right-to-left while Roman numerals are displayed left-to-right.

Characters are stored as codes to facilitate searching and other text manipulation tasks. Before text can be displayed, characters must be rendered. **Character rendering** is the process of properly preparing the characters for display, taking into account **line direction**, contextual rules, and character reordering. For example, the formation of **ligatures** and **diphthongs** occurs during the display of text.

Once text is displayed, an application needs to provide ways for the user to edit the text. The user may want to change the font, add or delete text, or change **font attributes** such as size, color, and **style**. Users may also want to align text, set tabs, or change other attributes that affect text layout.

How MLTE Handles Text

When you use MLTE, your application does not need to contain all the code necessary to handle text. It can rely on MLTE and other Mac OS text technologies to do most of the grunt work for you. This section describes what tasks MLTE and other Mac OS text technologies handle for you, and what tasks your application needs to handle. If all your application needs to do is to display static text in a text box, you can skip this section and go to "Displaying Static Text" (page 25). You need only to use one of two MLTE functions to display static text.

Note: If you have ever used TextEdit in an application, be prepared to write fewer lines of code when you use MLTE. You'll see that MLTE handles many tasks that TextEdit did not handle.

MLTE supports standard text input from a keyboard. Once your application creates an MLTE document and associates the document with a window, anything your user types is displayed in the window. Your user may type on any keyboard layout using any script currently supported on the Mac OS. MLTE automatically supports **inline input** for Chinese, Japanese, Korean, other double-byte scripts, and Unicode text.

When your application calls the MLTE initialization function `TXNInitTextension`, MLTE enables the **Text Services Manager** automatically. The Text Services Manager (TSM) is the part of the Mac OS that provides an environment for applications to use input methods that support text entry for languages such as Chinese, Japanese, and Korean.

In Mac OS X, once you initialize MLTE, you do not need to do anything else to assure support for text input. Behind the scenes, MLTE calls TSM functions as needed, and TSM passes text input directly to an active MLTE document.

By default, MLTE stores text as Unicode. If a user enters text in an encoding other than Unicode, MLTE calls the appropriate **Text Encoding Conversion (TEC) Manager** functions to convert the text to Unicode. You do not need to call the TEC Manager functions directly.

MLTE uses **Apple Type Services for Unicode Imaging (ATSUI)**and QuickDraw Text functions to render and draw text. ATSUI handles complex scripts, supports bidirectional text, and implements a number of other advanced typographical features.

Your application does not need to call any ATSUI functions directly; MLTE does it for you. However, if you want to display and support editing text that uses advanced typographic features, such as ligatures or diphthongs, your application must supply ATSUI information to MLTE functions. For example, if you want MLTE to display text using a specific rare ligature, your application needs to pass an ATSUI constant to specify the ligature. ATSUI constants are defined in the reference documentation for ATSUI. The section "Displaying Static Text" (page 25) indicates the functions for which ATSUI constants may be used.

MLTE has a variety of functions your application can use to make changes that affect an entire MLTE document or to change text attributes. For example, when a user changes the style of a selection, your application must call the appropriate MLTE function to set the style attributes for the selection.

Your application must provide the appropriate user feedback when the user takes any actions that affect what is displayed in any text-related menus, such as the Edit and Format menus. For example, if the user changes the justification for a selection, you application must call the appropriate Menu Manager function to display a check mark next to the appropriate item in the Format menu that applies to the selection.

Note: If you set up MLTE to handle edit-command events, you do not need to update the Edit menu.

There are a number of user actions that MLTE handles automatically for your application, such as highlighting selected text, dragging selected text, and moving the **caret** in response to arrow key presses.

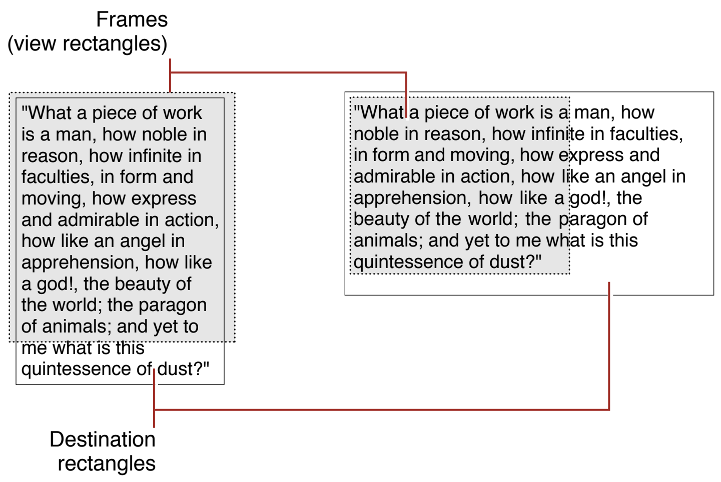
Text Objects (TXNObject)

MLTE supports onscreen text editing by maintaining information about where the text is stored, where to display it, and the text attributes (style, font, and so forth). This information is contained in an opaque structure called a `TXNObject`, or **text object**. From the user's perspective, a text object could take many forms. For example, if your application is a simple text editor, a user views the text object as a document that can be saved, opened, and edited. If your application is a drawing program that uses MLTE for labeling support, the user views a text object as a label or caption associated with a graphic object. If your application is a natural-language query program that uses MLTE for query input, a user views the text object as an input window.

When your application calls the MLTE function `TXNNewObject` to create a text object, you specify the window in which the text is to be drawn. The window defines the **destination rectangle**—the area in which the text is drawn. You also have the option to specify the frame. The **frame** is that portion of the window within which the text is actually displayed. It is sometimes referred to as the **view rectangle**.

Figure 2-1 shows two sets of frames and destination rectangles. The frames are shaded and defined by dotted lines. The text is drawn in the destination rectangle; the part of it that is displayed is defined by the frame.

Figure 2-1 Frames and destination rectangles


Frames
(view rectangles)
"What a piece of work is a man, how noble in reason, how infinite in faculties, in form and moving, how express and admirable in action, how like an angel in apprehension, how like a god!, the beauty of the world; the paragon of animals; and yet to me what is this quintessence of dust?"
"What a piece of work is a man, how noble in reason, how infinite in faculties, in form and moving, how express and admirable in action, how like an angel in apprehension, how like a god!, the beauty of the world; the paragon of animals; and yet to me what is this quintessence of dust?"
Destination
rectangles

A `TXNObject` is an opaque structure, so your application cannot access it directly and the MLTE reference documentation does not define explicitly the fields in the `TXNObject`. A `TXNObject` contains references to the window frame and

- the text associated with the text object
- the options you want the frame to support, such as scroll bars and a size box
- the file type of the text object, such as plain text or RTF, and whether the text can contain graphics, movie, and sound data
- the **text encoding** to use; Unicode is the default
- the attributes of the text associated with the text object
- information about the layout of the text object

MLTE provides a number of functions that operate on the text object for you, allowing you to get and set attributes of the entire text object or of runs of text associated with the object.

Text Attributes

When the MLTE function `TXNNewObject` creates a text object, MLTE creates other structures that maintain information about the styles and attributes of the text associated with a text object.

MLTE keeps track of character attributes as runs. A **run** is a sequence of characters that are contiguous in memory and share a set of common attributes.

MLTE has two functions your application can use to get or set text attributes. Attributes are passed to and from these functions in a type attributes structure. This structure supports your application's use of ATSUI font features and font variations, should you want to use them.

A **font feature** is the set of typographic and layout characteristics that creates a specific appearance for a glyph. For example, displaying text as small caps is a font feature. A **font variation**is a range of typestyles along a **variation axis**. For example, a "width" font variation allows your application to condense or expand a line of displayed text. If you want to get or set such ASTUI font features or variations, your application can pass ATSUI font features or variations information in the type attributes structure. See "Accessing and Displaying Advanced Typographical Features" (page 49) for an example of how to get ATSUI font features.

The MLTE User Interface

MLTE functions follow Mac OS user interface guidelines, so using these functions ensures the presentation of a consistent user interface in your application.

Menu Support

Although MLTE does not handle any menu except for the Font menu, it provides applications with all the necessary functionality and information to support the standard text editing menus as specified in *Apple Human Interface Guidelines*. MLTE supports the Undo, Redo, Cut, Copy, Paste, Clear, and Select All items in the Edit menu, but does not support Publish and Subscribe.

MLTE supports undo for the Cut, Copy, Paste, and Clear commands as well as applying a font, size, or style to a non empty selection. However, applying a font, size, or style to an insertion point cannot be undone. Undo cannot be applied to a Select All command because Select All is a selection operation.

An uninterrupted sequence of keystrokes, whether done as standard input on a keyboard or as inline input using an input method, is treated as a single typing command for purposes of the Undo command. Events that interrupt a typing sequence include selection operations (except for those handled by input methods), deactivating a window, printing, or any undoable command other than typing.

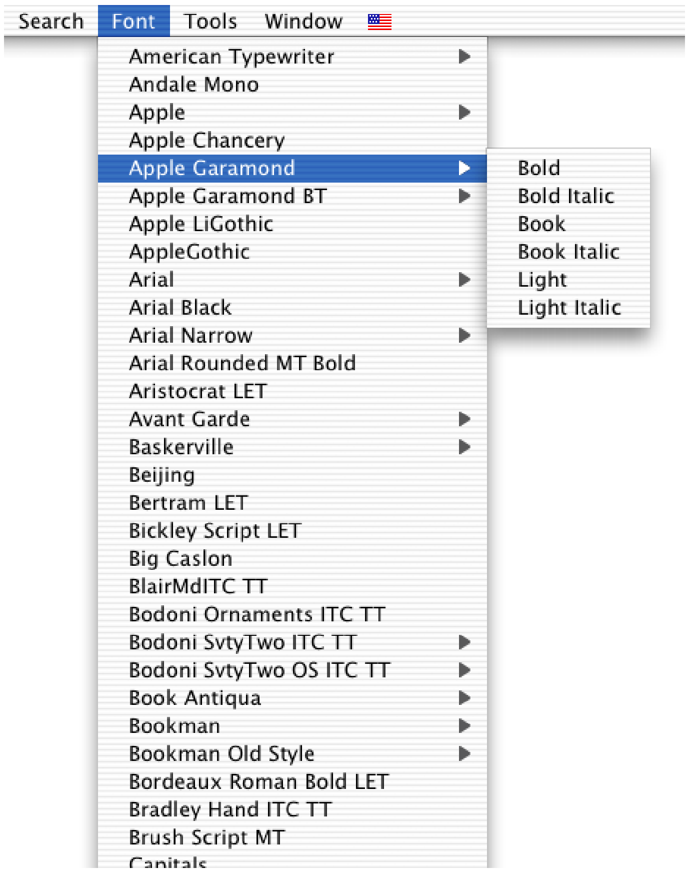
Font Menu

Because MLTE supports both QuickDraw Text and ATSUI without requiring your application to know which is being used, MLTE provides utility functions for creating and handling a standard Font menu (where standard is defined as what is most appropriate for the imaging system in use).

MLTE provides an opaque structure called `TXNFontMenuObject` and the functions `TXNNewFontMenuObject` and `TXNDoFontMenuSelection` that you can use to build a Font menu and handle user interaction with the Font menu. If you prefer to have your application build its own Font menu, you can do so. But if you use the MLTE Font menu functions you'll have fewer lines of code to write.

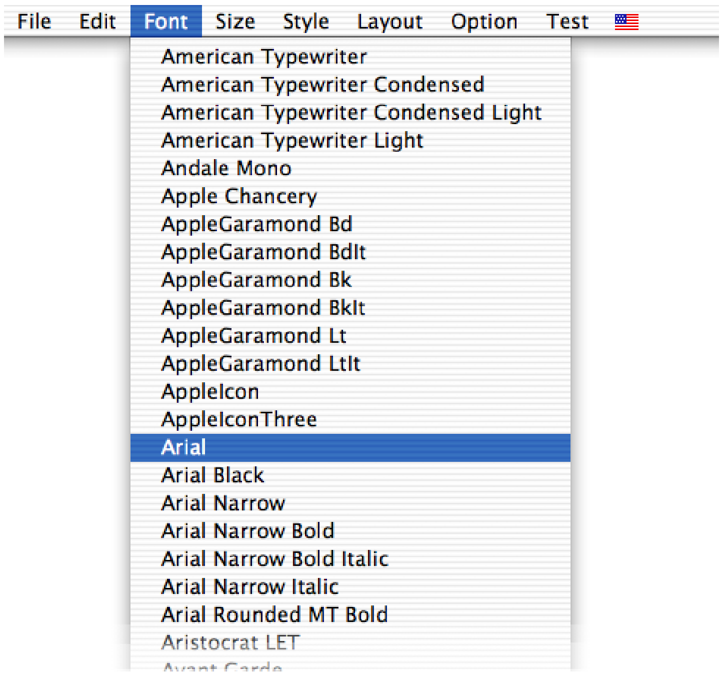
When your application uses MLTE on a system that has ATSUI (which is the default), the Font menu automatically includes hierarchical submenus for ATSUI fonts that share a family name but have different style names. MLTE draws each Font menu item in a single system font. Figure 2-2 (page 14) shows an MLTE Font menu drawn in Mac OS X using ATSUI.

Figure 2-2 Font menu on a system that uses ATSUI



An MLTE Font menu on a system that uses only QuickDraw Text looks similar to a menu created by calling the Menu Manager function `AppendResMenu`, as shown in Figure 2-3.

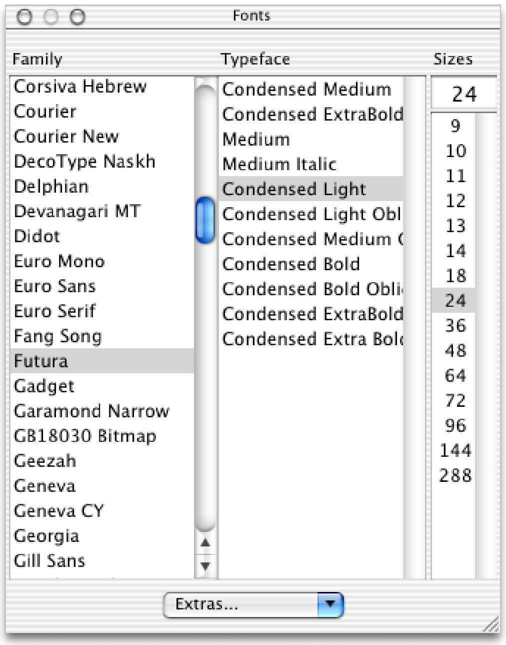
Figure 2-3 A Font menu drawn in Mac OS X using QuickDraw



Fonts Window

Prior to Mac OS X version 10.1, the Font menu was the only Font interface available to Carbon applications. With the release of Mac OS X version 10.2, Carbon applications can provide a Fonts window, as shown in Figure 2-4. The Fonts window is the preferred user interface for applications that run only in Mac OS X. The columns in the Fonts window improve the viewing and selection of large collections of fonts compared to the hierarchical Font menu shown in Figure 2-2 (page 14). The columns provide easy access for users to select font, style, and size.

Figure 2-4 The Fonts window



MLTE does not handle the Fonts window for you. If you choose to use a Fonts window in your application, your application must perform the following tasks:

- show and hide the Fonts window
- handle a selection event in the Fonts window
- programmatically set a selection in the Fonts window
- handle a change of user focus from one document to another

For more information and programming examples on supporting a Fonts window in your application, see *Apple Type Services for Fonts Programming Guide*.

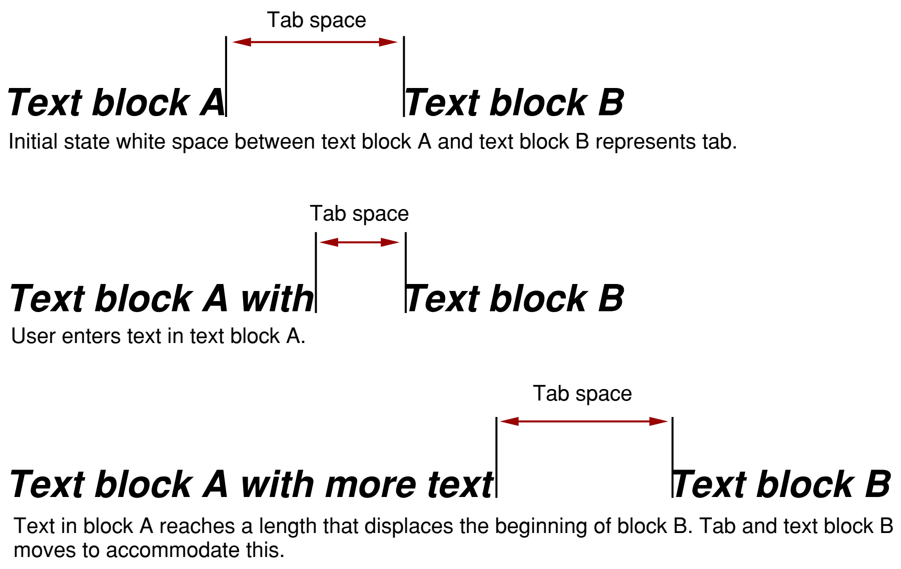
Document-Wide Formatting

Tab settings and text **alignment** are characteristics that apply to an entire text object.

Tab Settings

MLTE interprets tab characters based on the one-tab-per-document standard found in many simple text editors. Each tab character maps to an initial width. As MLTE flows text onto a line, each tab is replaced by the width value necessary to place the start of the text following the tab at a given position on the line. As the text placed before the tab grows, the white space appears to shrink until the preceding text becomes long enough to envelop the entire tab. At that point, the tab assumes its full width and the text following the tab jumps ahead. Figure 2-5 illustrates this point.

Figure 2-5 Text entry and tab behavior


Tab space
Text block A
Text block B
Initial state white space between text block A and text block B represents tab.
Tab space
Text block A with
Text block B
User enters text in text block A.
Tab space
Text block A with more text
Text block B
Text in block A reaches a length that displaces the beginning of block B. Tab and text block B moves to accommodate this.

MLTE flows tab widths in the line direction for the line being formatted. If text is wrapped automatically and a tab width extends past the trailing margin (the right margin on a Roman system), MLTE wraps the line and the next visual line begins with the tab width.

Text Alignment

MLTE allows you to specify the alignment of the lines of text, that is, their horizontal placement with respect to the left and right edges of the **text area** or **destination rectangle**. The different types of alignment that MLTE supports accommodate script systems that are read from right to left as well as those that are read from left to right. MLTE supports the following types of alignment:

- Default alignment—MLTE positions text according to the line direction of the **system script**. The alignment can be either left or right. Line direction is the direction in which text in a particular language is written and read. The English language has a left-to-right line direction. Arabic and Hebrew have a primarily right-to-left line direction.
- Center alignment—MLTE positions text so it is centered between the right and left edges of the destination rectangle.
- Right alignment—MLTE positions text so it is flush with the right edge of the destination rectangle.
- Left alignment—MLTE positions text so it is flush with the left edge of the destination rectangle.
- Full **justification**—MLTE positions text so it is flush against both right and left edges of the destination rectangle.

Typing and Inline Input

MLTE assumes that your application filters out all characters it wants to handle. MLTE uses the following list of rules to process the characters it handles. (Unicode encodings are designated as Uxxx and Mac OS encodings are designated as $xxx, where xxx is replaced by a value.)

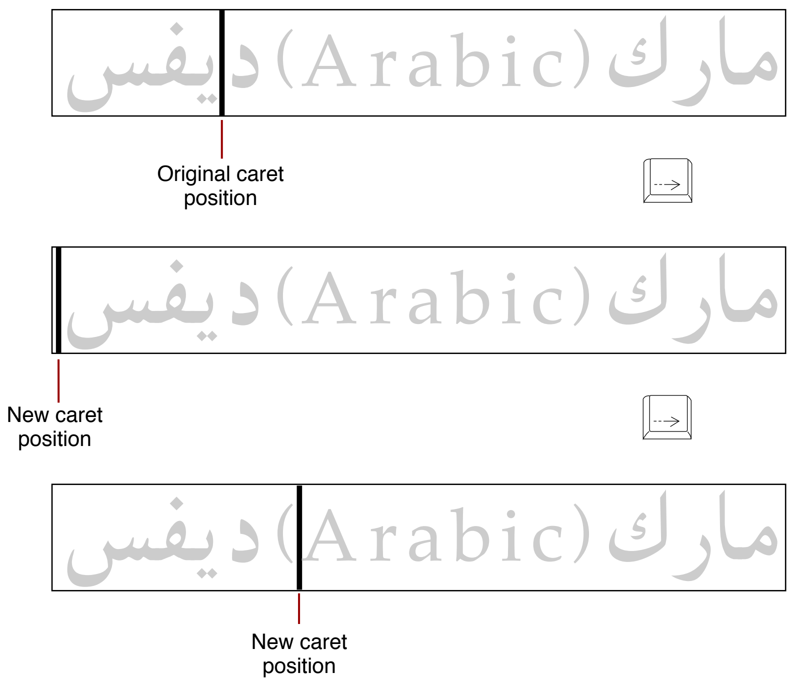
- Inserting: All single-byte and double-byte characters starting at ($20, U0020) except Forward Delete ($7F, U007F) are inserted into the text. Return ($0D, U000D) and the Tab character ($09, U0009) are inserted. Characters entered through inline input are always inserted.
- Selecting: All combinations involving the arrow keys ($1C–$1F, U001C–U001F) are interpreted as selection operations (see "The Selection Range, the Insertion Point, and Highlighting in MLTE" (page 20)). Other than as specified in *Apple Human Interface Guidelines*, they do interrupt typing commands in MLTE.
- Scrolling: The Home ($01, U0001) and End ($04, U0004) characters are interpreted to scroll the text block to its logical beginning or end as specified in *Apple Human Interface Guidelines*.
- Paging: The Page Up ($0B, U000B) and Page Down ($0C, U000C) characters are interpreted to scroll the text up or down once according to the height of the currently visible portion. They are not part of typing commands, but they also don't interrupt typing commands.
- Deleting: The Backspace ($08,U0008) and Forward Delete ($7F, U007F) characters first delete the currently selected text (if the selection is nonempty), then delete individual characters logically preceding (Backspace) or following (Forward Delete) the insertion point . They are part of typing commands.
- All other characters are ignored. This includes all key combinations involving the Command key but not involving the arrow keys. They are not part of typing commands, but do not interrupt the typing commands.

Caret Position and Movement

MLTE uses a caret to mark the position in the displayed text where the next editing operation is to occur. When your application uses MLTE to paste text into a text object, MLTE positions a caret after the pasted text. For **unidirectional text**, when the user presses an arrow key to move the caret left or right across the text, MLTE moves the caret in the direction of the arrow key. If the document contains **embedded objects**—graphics, movie, or sound data embedded in text—the caret treats the embedded object as a single character.

For bidirectional text, the **caret position** at a direction run boundary depends on the direction of the **keyboard script**; **split carets** are not supported. Figure 2-6 shows a sequence of two Right Arrow key presses and their impact on caret display and movement in a line containing bidirectional text. In this example, the **primary line direction** is right to left.

Figure 2-6 Caret movement across a direction boundary



In the first instance of the **text segment**, the caret is positioned within the Arabic text. When the user presses the Right Arrow key once, the insertion point is positioned on a **direction boundary** and the caret jumps to the left side of the Arabic text. When the user presses the Right Arrow key again, MLTE displays the caret at the right side of the left parenthesis in the Roman text.

For read-only documents, you can choose among three behaviors for your application. The first behavior is to display a blinking caret and allow selection and copying of text. The second behavior is not to display a caret or allow text to be selected. This is the model used by SimpleText for read-only documents. The third behavior is to display a non-blinking caret and allow selection and copying of text.

Horizontal arrow keys move in a direction dependent on the line direction of the text. The arrow key moving in the line direction (right for Roman) starts at the trailing edge of the highlight region in the last line of the selection and simulates successive clicks at each character boundary moving in the line direction until it hits the trailing edge of the visual line. At that point, selection wraps to the leading edge of the next visual line. The character boundaries are determined by the **storage order** and not the **display order**.

The arrow key moving against the line direction (left for Roman) starts at the leading edge of the highlight region in the first line of the selection and simulates successive clicks at each character boundary moving against the line direction until it hits the leading edge of the visual line, then wraps to the trailing edge of the previous visual line.

For the horizontal arrow keys, a ligature that does not allow for an insertion point between its constituting characters is treated as one character. Combining the Option key with a horizontal arrow key simulates clicks at word boundaries instead of character boundaries. Combining the Command key with a horizontal arrow key in or against the line direction simulates clicks at the trailing edge or leading edge of the last line or first line intersecting with the selection, respectively.

When the user presses the Up Arrow key, the caret moves up one line, even in lines of text containing fonts of different sizes. When the caret is positioned on the first line of a **text object**, and the user presses the Up Arrow key, MLTE moves the caret to the beginning of the text on that line. This position corresponds to the visible right end of a line when the primary line direction is right to left and to the left end of the line when the primary line direction is left to right.

Similarly, when the user presses the Down Arrow key, the caret moves down one line. When the caret is positioned on the last line of a text object, and the user presses the Down Arrow key, MLTE moves the caret to the end of the text on that line. That is, the caret moves to the visible left end of a line when the primary line direction is right to left and to the right end of a line when the primary line direction is left to right.

Combining the Option key with an Up Arrow or Down Arrow key simulates a click at the corresponding edge of the portion of the view shown in the window, paging the view first if the active selection was already at that edge. The starting point for a selection is determined at the beginning of an uninterrupted sequence of Up Arrow and Down Arrow keys.

The Selection Range, the Insertion Point, and Highlighting in MLTE

A user can select a range of text to be edited or your application can set the **selection range** programmatically. A user can define a selection by creating a new one or modifying the current one. A new selection is defined by the Select All command, by mouse actions (single-, double-, or triple-clicking or dragging), or by using the arrow keys along with the Shift key (which could be combined with the Command or Option keys). A selection can be modified by pressing the Shift key and performing a mouse-based or arrow key–based selection action. MLTE interprets user actions to modify a selection according to the fixed-point model described in the *Apple Human Interface Guidelines*.

The **anchor point**is the position in the text at which the user positions the pointer and presses the mouse button. When visual feedback shows the desired range, the user releases the mouse button. The point at which the button is released is called the **active end**of the range. The user can expand or shrink the active selection by performing a modifying action such as pressing and holding the Shift key while pressing an arrow key. If necessary, the text scrolls to make a selection visible in the view rectangle. See *Apple Human Interface Guidelines* for more details, including illustrations, on selection behavior.

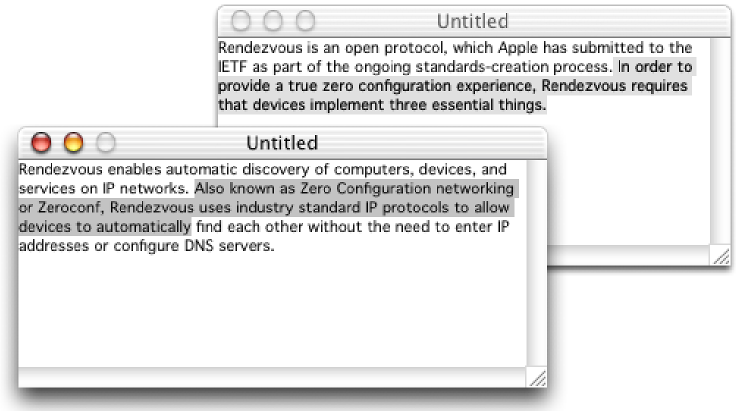
Single-clicking defines an insertion point. Double-clicking selects a word as defined by the Script Manager or ATSUI. Triple-clicking selects a visual line from the beginning of the line to the beginning of the next line. Quadruple-clicking selects the paragraph. If the user performs a double, triple, or quadruple click, then drags, the selection grows by words, visual lines, or paragraphs respectively. A click in an empty space is mapped to some location that has text.

Regardless of how text is selected, the selected text becomes the current selection range. MLTE uses a **byte offset**to identify the position of a character in the text object, and a text object includes fields that specify the byte offsets of the characters that correspond to the beginning and the end of the current selection range in the displayed text. A graphics, movie, or sound object embedded in a text object is treated as a single character.

When the byte offset values for the beginning and the end of the selection range are the same, the selection range is an insertion point. MLTE displays an insertion point as a blinking caret in the form of a vertical bar (|). You can turn caret display on or off by setting the appropriate frame option when you create a new text object in MLTE.

MLTE highlights a selection range. Highlight regions for nonempty selections are drawn in the system highlight color, while carets are drawn in black. MLTE modifies the highlighting as shown in Figure 2-7 for selected text in an inactive window, as required by the Drag Manager.

Figure 2-7 Highlighted text selection in background window

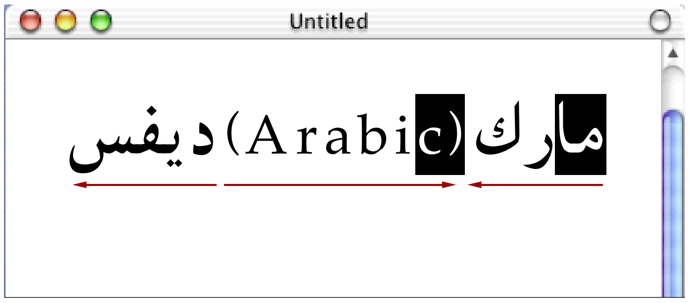


Rendezvous is an open protocol, which Apple has submitted to the IETF as part of the ongoing standards-creation process. In order to provide a true zero configuration experience, Rendezvous requires that devices implement three essential things.

Rendezvous enables automatic discovery of computers, devices, and services on IP networks. Also known as Zero Configuration networking or Zeroconf, Rendezvous uses industry standard IP protocols to allow devices to automatically find each other without the need to enter IP addresses or configure DNS servers.

Because MLTE supports bidirectional text, the selection range can appear as **discontinuously highlighted** as shown in Figure 2-8. Displayed text is highlighted according to the storage order of the characters. When multiple script systems with different line directions are installed, a continuous sequence of characters in memory may appear as a discontinuous selection when displayed.

Figure 2-8 Discontinuous highlighting



Drag and Drop

MLTE provides the drag and drop user experience as specified in the *Apple Human Interface Guidelines*. MLTE highlights selections in inactive views, so that users can drag between active and inactive views. If the cursor is over the highlighted region in an active view, the cursor changes to an arrow. MLTE automatically distinguishes between selection operations and drag-and-drop operations, and MLTE provides user feedback. If the mouse-down event occurs within the highlighted region of the current selection and the Drag Manager is available, then MLTE waits to see whether the mouse is dragged. If the mouse is dragged, MLTE initiates a drag–and–drop operation. Otherwise, MLTE interprets the mouse event as a selection operation.

Because MLTE has no contextual information, it cannot provide your application feedback about the destination of the dragged text. However, it highlights the insertion point where dropped text gets inserted, performs the actual move, and selects the dragged text in its new location. By default, MLTE recognizes dragging as a move operation. The user can drag a copy by pressing the Option key.

Line Breaking

MLTE renders text into a single rectangular frame. You can choose to have your application create arbitrarily wide lines or break lines at a certain width. If you choose to break lines, MLTE uses ATSUI line-breaking algorithms to control where a line breaks. Text is flowed into a visual line as long as it fits, then a new line is started with the first unbreakable unit (such as a word) that did not fit completely in the line.

If your application turns off ATSUI (this is not recommended), MLTE uses QuickDraw Text line–breaking algorithms. In this case, in scripts that use space characters to separate words, one (and only one) space character at the logical end of the text flowed into a visual line is consumed by the line break—it is ignored for measurements and not displayed.

ATSUI Font Features and Variations

Your application can pass ATSUI features and variations to MLTE functions and have them applied to a selection. This, like building the Font menu, requires your application to be aware that it is running on a system that has ATSUI, and further requires you to use some of the moderately complicated ATSUI functions.

MLTE does not provide a human interface that allows a user to view and select font variations and features on a per font basis. However, your application can create a user interface that displays font variation and font features. Then you can use MLTE functions to get and set font features and font variations in response to items your user changes in the interface your application creates. See "Accessing and Displaying Advanced Typographical Features" (page 49) for information on how your application could provide this capability.

Keyboard and Font Synchronization

Keyboard and font synchronization is a process by which the operating system compares the current keyboard script to the script of the font at the current insertion point. If the two do not match, one or the other is changed so the two scripts are the same.

Keyboard to Font Synchronization

In a multiscript environment, the operating system should display text in a font that supports the character set in which the text is written. In the **WorldScript** environment, the operating system typically monitors the current keyboard script and compares it to the script of the font at the current insertion point. If the two scripts do not match and the user starts typing, the operating system automatically replaces the font with one that belongs to the keyboard script.

This behavior is not always appropriate, as there is not a one-to-one correspondence between fonts and keyboards. Typically, non-Roman keyboard layouts support only the characters that are specific to that script, not the ASCII characters that are supported by all fonts designed for the WorldScript environment.

Despite these drawbacks, MLTE by default attempts to synchronize the font to the keyboard when the user changes the keyboard. To find the appropriate font, MLTE first searches backwards in the document for an appropriate font, then forward. If it does not find an appropriate font, the **application font** or the **system font** for the keyboard script is used. Font synchronization does not interrupt typing commands.

Font to Keyboard Synchronization

Some text editors also support synchronization in the opposite direction: They automatically switch the keyboard script to the script of the font being used at the current selection. For example, this could happen when the user changes the selection. The assumption is that the user is most likely to type additional text in the script already being used for the current selection. Also, the location of the caret in bidirectional text may depend on the direction of the keyboard script. In this context, it is important that the direction of the keyboard script matches the direction of text in which the user clicks.

In double-byte scripts, the issue of caret placement does not exist, so input methods often allow users to enter ASCII characters in a **pass-through mode**. Switching the keyboard is not necessary. Users of single-byte scripts can enter ASCII characters only by switching to a **Roman keyboard script**.

MLTE supports font to keyboard synchronization by default. You can turn off keyboard synchronization by using the `kTXNNoKeyboardSyncMask`constant when you set the `iFrameOptions` parameter in the function `TXNNewObject`.

Overriding Font Synchronization

A user can override the behavior by pressing and holding the Control key while changing the font. In this case, the text changes to the selected font no matter what characters are selected. Each character that is not supported by the new font will be represented by a **missing character glyph**(such as an empty rectangle). To represent missing glyphs with a character that resembles the missing one more closely, you can turn on "Font Substitution" (page 23).

Font Substitution

There may be situations in which MLTE cannot draw a character with the assigned font because the font is not installed on the user's system. If you set font substitution options, MLTE will use the transient font matching function supplied by ATSUI. The transient font matching function scans all valid fonts on the user's system until it finds a suitable substitute font. If you do not set this bit, missing character glyphs are used to represent unavailable characters.

You can use the font fallback mask `kTXNUseFontFallBackMask` to set fallback options for static text display. If you want to use font substitution for a text object, you can use the font substitution tag `kTXNDoFontSubstitution`. See *Multilingual Text Engine Reference* for more information on these constants.

MLTE and Carbon Events

Prior to the release of Mac OS X version 10.1, MLTE used the Apple Event Manager to handle text input. Beginning with Mac OS X version 10.1, MLTE uses the Carbon Event Manager instead. Because MLTE uses Carbon events, you have less code to write, as MLTE handles most Carbon events without your intervention. If you prefer to handle some of the Carbon events MLTE now handles, you can. You can install Carbon event handlers on top of MLTE's handlers, as long as you are careful to call the Carbon Event Manager function `CallNextEventHandler` or return `eventNotHandledErr` if the event should be passed to MLTE for handling. For further details, see "Customizing MLTE Support for Carbon Events" (page 54).

MLTE Tasks

This chapter provides instructions and code samples for the most common tasks you can accomplish with Multilingual Text Engine (MLTE), such as displaying static text, working with document-wide settings, and handling File, Edit, and Font menu commands.

The section on advanced topics includes working with embedded objects (graphics, sound, movies), displaying chemical equations, and displaying advanced typographical features (such as ligatures).

You should read the section "Migrating an Application from TextEdit to MLTE" (page 61) if you have an existing application that uses TextEdit, and you want to determine what you need to do to rewrite the code so your application uses MLTE instead.

The code samples assume you are developing your application on Mac OS using CarbonLib. All code samples in this chapter are in C.

Displaying Static Text

MLTE provides an easy way for your application to display static text whether or not it uses other MLTE features to implement editing services. You do not need to initialize MLTE to display static text. You can use a static text box for such purposes as displaying text information associated with a control.

Note: If you want to display a document that is read-only, do not use a static text box. Instead, call the function `TXNNewObject` to create a text object with the frame options parameter (`iFrameOptions`) set to read only (`kTXNReadOnlyMask`).

The `TXNDrawUnicodeTextBox` and `TXNDrawCFStringTextBox` functions display text that a user cannot edit. You use the `TXNDrawUnicodeTextBox` function when you want to display a Unicode string and the `TXNDrawCFStringTextBox` function when you want to display a `CFString`object. Each function draws the text in a rectangle whose size you specify in the local coordinates of the current graphics port.

MLTE uses the ATSUI style you specify to display the text or creates an ATSUI style based on the style associated with the current graphics port. You can specify a number of other options, such as text orientation (horizontal or vertical) and text alignment (right, left, centered, or fully justified).

Listing 3-1 shows how to use the `TXNDrawCFStringTextBox` function to display a static string. The `TXNDrawCFStringTextBox` function draws into the current graphics port. An explanation for each numbered line of code appears following the listing.

Listing 3-1 Displaying static text in a text box

```
static void MyDrawStaticText (CFStringRef stringToDisplay,
                              WindowRef theWindow)
{
    Rect            bounds;
    GrafPtr         myOldPort;

    GetPort (&myOldPort);                                           // 1
    SetPortWindowPort (theWindow);                                  // 2
    EraseRect (GetWindowPortBounds (theWindow, &bounds));           // 3
    TXNDrawCFStringTextBox (stringToDisplay, &bounds, NULL, NULL);  // 4
    SetPort (myOldPort);                                            // 5
}
```

Here's what the code does:

1. Call the QuickDraw function `GetPort` to save the current graphics port. You'll need to restore this later.

> **Note:** QuickDraw uses the default CGContext. It is also possible for you set up and use your own CGContext. See *Quartz 2D Programming Guide* for more information.

2. Calls the QuickDraw function `SetPortWindowPort` to set the graphics port to the window port.

3. Calls the QuickDraw function `EraseRect` to make sure the rectangle in which you will draw is empty.

4. Calls the MLTE function `TXNDrawCFStringTextBox` to draw the `CFString` passed to the function.

5. Restores the graphics port by calling the function `SetPort`.

Initializing MLTE

You need to initialize MLTE before you can use any MLTE function except the two functions that display static text—`TXNDrawUnicodeTextBox` and `TXNDrawCFStringTextBox`. You should initialize MLTE at the same time you call other initialization functions for your application.

At the very least, to use MLTE your application must call the MLTE initialization function `TXNInitTextension`. On a more practical level, most applications need to provide users with a Font menu or Fonts window. So your application should also set up the menu bar and the Font user interface in addition to calling the MLTE initialization function `TXNInitTextension`. To set up your application to use MLTE functions and provide users with a Font menu you need to perform the tasks described in this section.

Setting Up Font Descriptions

When you call the MLTE initialization function `TXNInitTextension`, you pass an array of **font descriptions**, which are structures of type `TXNMacOSPreferredFontDescription`. Each font description specifies the **font family** ID, point size, style, and encoding. The array can be `NULL` or can have an entry for any encoding for which you would like to designate a default font. You can use the MLTE constants `kTXNDefaultFontName`, `kTXNDefaultFontStyle`, and `kTXNDefaultFontSize` as the font family ID, point size, and style values for a font description. You can supply an encoding of type `TextEncoding`, created using the function `CreateTextEncoding`. See Listing 3-2 (page 27) for an example of assigning font default values to a single font description.

Assigning Initialization Options

You can specify whether MLTE should support data types other than text, such as graphics, movies, and sound, in your application. You can specify other data types by using the initialization option masks described in *Inside Mac OS X: MLTE Reference*. See `Listing 3-2` (page 27) for an example of assigning initialization options using the option masks supplied by MLTE.

Calling the MLTE Initialization Function

You initialize MLTE by calling the `TXNInitTextension`function. You need to call this function only once. Calling it more than once returns the result code `kTXNAlreadyInitializedErr` and has no effect. If for some reason you want to initialize MLTE again while your application is running, you must first call the `TXNTerminateTextension`function.

Listing 3-2 shows how you can initialize MLTE using a `MyInitializeMLTE`function. You call the `MyInitializeMLTE` function from your application's one-time-only initialization function. An detailed explanation for each numbered line of code appears following the listing.

Listing 3-2 Initializing MLTE

```
void MyInitializeMLTE (TextEncodingBase myTextEncodingBase)
{
    OSErr status;
    TXNInitOptions options;
    TXNMacOSPreferredFontDescription  defaults;                  // 1

    status = ATSUFindFontFromName ("Times Roman",
                        strlen("Times Roman"),
                        kFontFullName, kFontNoPlatform,
                        kFontNoScript, kFontNoLanguage,
                        &theFontID);
    defaults.fontID = theFontID;                                 // 2
    defaults.pointSize = kTXNDefaultFontSize;                    // 3
    defaults.fontStyle = kTXNDefaultFontStyle;                   // 4
    defaults.encoding  = CreateTextEncoding (myTextEncodingBase,
                            kTextEncodingDefaultVariant,
                            kTextEncodingDefaultFormat);         // 5

    options = kTXNWantMoviesMask | kTXNWantSoundMask |
                                  kTXNWantGraphicsMask;          // 6

    status = TXNInitTextension (&defaults, 1, options);          // 7
    if (status != noErr)
        MyAlertUser (eNoInitialization);                         // 8
}
```

Here's what the code does:

1. Declares a data structure to hold the default font values.

2. Sets Times Roman as the default font by calling the ATSUI function `ATSUFindFontFromName` to obtain the font ID associated with the font name. If you don't need to assign a particular font you can assign the default system font using the following line of code:

```
    defaults.fontID     = kTXNDefaultFontName;
```

3. Assigns the default font size.

4. Assigns the default font style.

5. Assigns the default text encoding. The encoding must be a `TextEncoding` data type, created by calling the Text Encoding Manger function `CreateTextEncoding`. The sample function `MyInitializeMLTE` uses the `TextEncodingBase` passed to the function. You would either need to determine the current text encoding base or provide one of the Base Text Encoding constants defined in the Text Encoding Conversion Manager.

6. Assigns initialization options. The sample code sets up options to support movies, sound, and graphics embedded in text data.

7. Initializes MLTE by calling the function `TXNInitTextension`. You need to pass an array of font descriptions. In this case, there is only one description in the array. You also need to pass the initialization options.

8. Checks for an error condition, and if there is one, calls your function to handle the error. You can't use MLTE unless it initializes without error. See "Posting an Alert" (page 44) for information on writing an alert function.

Setting Up the Menu Bar

Your application needs to set up menus as part of its initialization function. Once you've set up the menu bar, you can use MLTE functions to create a Font menu and handle user interaction with the Font menu.

Creating a Font Menu Object

The `TXNFontMenuObject`structure is an opaque structure that MLTE uses to handle user interaction with the Font menu. You create a Font menu object by calling the `TXNNewFontMenuObject`function. You should create a font menu object at the same time you are preparing to display your menu bar.

When you call the `TXNNewFontMenuObject` function, you must provide the Font menu reference, the menu ID, and the menu ID at which hierarchical menus begin. By default, MLTE creates hierarchical menus similar to what is shown in Figure 2-2 (page 14).

Listing 3-3 shows how your application can create a Font menu object. If you want to display a check mark next to the active font in the Font menu the first time a user opens the menu, you must call the function `TXNPrepareFontMenu`. However, you call that function right after you create a text object, as shown in Listing 3-6 (page 33).

Listing 3-3 Creating a Font menu object

```
 void MySetUpFontMenu (MenuRef myMenuRef, SInt16 myMenuID)
{
    TXNFontMenuObject myFontMenuObject;                                   // 1
    OSStatus  status;

    status = TXNNewFontMenuObject (myMenuRef,
                              myMenuID,
                              kStartHierMenuID,
                              &myFontMenuObject);                  // 2
    if (status != noErr)
            MyAlertUser (eNoFontMenuObject);                       // 3
    DrawMenuBar();                                                 // 4
   }
```

Here's what the code does:

1. Declares a `TXNFontMenuObject` data type.

2. Creates a Font menu object by calling the MLTE function `TXNNewFontMenuObject`. You must supply a value greater than 160 that specifies the menu ID at which hierarchical menus begin. The sample code uses a an application-defined constant `kStartHierMenuID`. Note that MLTE creates hierarchical menus automatically on systems that use ATSUI. On output, `&myFontMenuObject` points to a new Font menu object.

3. Checks for an error. If there is one, calls your function to notify the user. See "Posting an Alert" (page 44) for information on writing an alert function.

4. Displays the menu bar by calling the Menu Manager function `DrawMenuBar`.

Note: If you want to provide a Fonts window in your application instead of a Font menu, see *Apple Type Services for Fonts Programming Guide*.

Terminating MLTE

You need to call the `TXNTerminateTextension`function when you terminate your application. Listing 3-4 shows how you can terminate MLTE when your application quits. You should first check to make sure all the document windows are closed and the Font menu object is disposed of before you terminate MLTE so that your application quits gracefully. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-4 Terminating MLTE in your application's termination function

```
void MyTerminate (TXNFontMenuObject myFontMenuObject)
{
    WindowPtr       theWindow;
    Boolean         closed;

    closed = true;
    do {
        theWindow = FrontWindow ();                                // 1
        if (theWindow != NULL)                                     // 2
            closed = MyDoCloseWindow (theWindow);
    }
    while (closed && (theWindow != NULL));                         // 3
    if (closed)
    {
        if (myFontMenuObject != NULL)                              // 4
        {
            OSStatus        status;
            status = TXNDisposeFontMenuObject (myFontMenuObject);             // 5
            // If there is an error
            if (status != noErr)
                MyAlertUser (eNoDisposeFontMenuObject);                       // 6
            myFontMenuObject = NULL;                                          // 7
        }
    }
    TXNTerminateTextension ();                                                // 8
}
```

Here's what the code does:

1. Gets the front window by calling the Window Manager function `FrontWindow`.
2. Checks to see if there is a window. If so, calls your function to close the window.
3. Iterates through each open window, closing each one.
4. When all the windows are closed, check for a Font menu object. If your application uses a Fonts window instead of a Font menu, you do not need to check for or dispose of a Font menu object.
5. Disposes of the Font menu object by calling the MLTE function `TXNDisposeFontMenuObject`.
6. Checks for an error. If there is one, calls your function to notify the user. See "Posting an Alert" (page 44) for information on writing an alert function.
7. Sets the Font menu object to `NULL`. You need to do this even if there is an error.
8. Terminates MLTE by calling the function `TXNTerminateTextension`.

Working With Text Objects

Text objects (`TXNObject`) are the fundamental structures in MLTE; most MLTE functions operate on them. A text object contains text along with character attribute information. Text objects also contain document-wide formatting and privileges information and the private variables and functions necessary to handle text formatting at the document level. (For an overview of text objects see "`Text Objects (TXNObject)`" (page 11).)

To work with text objects, your application must perform the tasks described in this section.

Creating a Window

Creating the text object is not of much use to your users unless you attach the text object to a window and make sure the window is visible. Your application can either create a text object and attach it to a window using the function `TXNAttachObjectToWindow`, or it can create a window and then create a text object with a reference to the window. Listing 3-6 (page 33) shows how to attach a new text object to a window.

Because of how MLTE uses Carbon events internally, the window that the document will be displayed in must have the standard event handlers installed. This can be easily done by doing one of the following:

- If you create the window by calling the Window Manager functions `CreateNewWindow` or `CreateCustomWindow` you should include the attribute `kWindowStandardHandlerAttribute`.

  Note: If you have an existing window, you can call the function `ChangeWindowAttributes` to add `kWindowStandardHandlerAttribute` to the window.

  For more information see *Handling Carbon Windows and Controls*.

- If you create the window from an Interface Builder nib file, you can call the Carbon Event Manager function `InstallStandardEventHandler` to install the standard event handlers on the window's target, as shown in Listing 3-5. For more information on the standard event handler see *Carbon Event Manager Programming Guide*.

A detailed explanation for each numbered line of code appears following Listing 3-5.

Listing 3-5 A function that creates a window from a nib file

```
WindowRef MyMakeNewWindow (CFStringRef inName)
{
    IBNibRef        windowNib;
    OSStatus        status;
    WindowRef       theWindow;

    EventTypeSpec   windowEventSpec[] =  {                              // 1
                {kEventClassWindow,  kEventWindowFocusRelinquish},
                {kEventClassWindow,  kEventWindowFocusAcquired},
                {kEventClassWindow,  kEventWindowCursorChange},
                {kEventClassCommand, kEventCommandProcess}};

    status = CreateNibReference (CFSTR ("window"), &windowNib);         // 2
    require_noerr (status, CantGetNibRef);                              // 3

    status = CreateWindowFromNib(windowNib, CFSTR ("Window"), &theWindow);  // 4
    require_noerr (status, CantCreateWindow);                           // 5

    DisposeNibReference (windowNib);                                    // 6
    status =  InstallWindowEventHandler (theWindow,
                        NewEventHandlerUPP (MyWindowEventHandler),
                        4, windowEventSpec,
                        (void *) theWindow, NULL);                      // 7
    status =  SetWindowTitleWithCFString (theWindow, inName);           // 8
    return theWindow;

    CantCreateWindow:
    CantGetNibRef:
        return NULL;
}
```

Here’s what the code does:

1. Declares an event specification. Your application would declare only those events it wants to handle. In this example, the following four events are specified:

- Window focus relinquished. You need to handle this event only if your application uses a Fonts window, as you will need to update the Fonts window display accordingly. See *Inside Mac OS X: Managing Fonts* for information on handling focus relinquished and focus acquired events.
- Window focus acquired. You need to handle this event only if your application uses a Fonts window, as you will need to update the Fonts window to display the current font selection in the window acquiring focus.
- Window cursor changed. When this event occurs, you can call your function to update the menus. See "Updating the File and Edit Menus" (page 38) for more information on updating menus.
- Command process. When this event occurs, you can call your function to process the command chosen by the user. See "Calling the Appropriate MLTE Function" (page 16) for more information on processing commands.

2. Creates a nib reference for the window by calling the Interface Builder Services function `CreateNibReference`. You must supply two parameters:
   - a `CFString` that represents the name of the nib file that defines the window user interface but without the `.nib` extension
   - a pointer to an `IBNibRef` data type. On return, this points to a reference to the nib file.
3. Checks for errors by calling the macro `require_noerr`. It the nib reference can't be created, the function terminates, as it should.
4. Creates a window from the nib reference by calling the Interface Builder Services function `CreateWindowFromNib`. On return, `theWindow` is a window reference to the newly-created window.
5. Checks for errors by calling the macro `require_noerr`. It the window reference can't be created, the function terminates, as it should.
6. Disposes of the nib reference.
7. Installs a window event handler on the window. This example installs a window event handler (`MyWindowEventHandler`) created by the application to handle the four events discussed previously. Everything except these four events will be handled by the standard window event handler. If you want to install only the standard event handler, you would use this code:

   ```
   status = InstallStandardEventHander (theWindow);
   ```
8. Set the title of the newly created window to the name passed to the function `MyMakeNewWindow`.

Setting Options for the Text Object's Frame

Before you create a text object your application must specify options for the text object's **frame** (that is, the view rectangle). Frame options determine whether the window in which the text object is displayed has scroll bars, a size box, or a number of other options. Listing 3-6 (page 33) shows how an application uses the MLTE frame option masks described in *Inside Mac OS X: MLTE Reference* to specify a frame that has horizontal and vertical scroll bars and a size box.

Creating a Text Object

You create a text object using the `TXNNewObject` function. Listing 3-6 (page 33) shows a sample function—`MyCreateTextObject`—that creates a text object and attaches it to a window. Error handling code has been omitted in the sample function so that you can more easily read the sequence of functions calls. A detailed explanation for each numbered line of code appears following the listing.

Once the text object is attached to the window, the `MyCreateTextObject` function sets the state of the scroll bars using the `TXNActivate`function and then prepares the Font menu for display. The function `TXNPrepareFontMenu` displays a check mark next to the active font in the Font menu the first time the user opens the Font menu associated with the text object. If you don't call the `TXNPrepareFontMenu` function, the check mark is not displayed until the second time the user opens the Font menu associated with the text object.

Listing 3-6 Creating a text object

```
OSStatus MyCreateTextObject (const FSSpec *fileSpecPtr,
          WindowRef  theWindow)
{
    TXNObject   textObject = NULL;              // text object
    TXNFrameID  frameID    = 0;                 // ID for text frame
    TXNFrameOptions frameOptions;

    frameOptions = kTXNShowWindowMask | kTXNWantVScrollBarMask |
                    kTXNWantHScrollBarMask |kTXNDrawGrowIconMask;             // 1
    status = TXNNewObject (fileSpecPtr,
                theWindow,
                NULL,
                frameOptions,
                kTXNTextEditStyleFrameType,
                kTXNTextensionFile,
                kTXNSystemDefaultEncoding,
                &textObject,
                &frameID,
                0);                                                           // 2
    SetWindowProperty (theWindow, kPropertyTag, kObjectTag,
                    sizeof (TXNObject), &textObject);                         // 3
    SetWindowProperty (theWindow, kPropertyTag, kFrameTag,
                    sizeof (TXNFrameID),&frameID);                            // 4
    status = TXNPrepareFontMenu (object, gTXNFontMenuObject);                 // 5
    return status;
    }
```

Here's what the code does:

1. Sets frame options for the text object's frame. The frame options shown here specify to display a window when the text object is created, and that the frame should have horizontal and vertical scroll bars and a size box.

2. Creates a text object. If `fileSpecPtr` is `NULL` the object is empty. Otherwise the object has the contents of the file to which `fileSpecPtr` points. The remaining parameters specify the following:

    - `theWindow`, is a reference to the window in which you want the document displayed
    - `NULL` specifies to use the window's port rectangle as the frame

- `frameOptions` are the options specified previously (See item 1.)
- `kTXNTextEditStyleFrameType` specifies to use a Text Edit-style frame
- `kTXNTextensionFile` specifies MLTE file format as the file type of the text object
- `kTXNSystemDefaultEncoding` specifies to use the default text encoding
- `textObject` is the newly created text object you obtain from the function. You need this later when you call other MLTE functions to operate on the text object.
- `frameID` is the text frame ID you obtain from the function. You need this later when you call other MLTE functions to operate on the text object.
- `0` specifies you have no private data. This is where you can specify a reference constant for your by your application.

Note: When you want to allow text input to this text object, you must call the function `TXNFocus`.

3. Calls the Window Manager function `SetWindowProperty` to save the text object as a property of the window. This allows your application to retrieve the text object later.

4. Calls the Window Manager function `SetWindowProperty` to save the frame ID as a property of the window. This allows your application to retrieve the frame ID later.

5. Call the function `TXNPrepareFontMenu` to prepare the Font menu for display. When you call this function, MLTE displays a check mark next to the active font. You can call this function only if you have already created a valid Font menu object using the function `TXNNewFontMenuObject`.

Note: If you are using a Fonts window instead of a Font menu in your application, then you do not call the MLTE function `TXNPrepareFontMenu`. See *Inside Mac OS X: Managing Fonts* for an example of what you would need to do to support the Fonts window.

Disposing of a Text Object

When you close a window associated with a text object, you should call the `TXNDeleteObject`function to release the text object and all associated data structures from memory.

Listing 3-7 (page 34) shows a sample function—`MyDisposeObject`—that first checks the `TXNGetChangeCount` function to see if the text object has been modified since it was created or saved last. If it has been modified, you can give the user an opportunity to save the changes before the object is deleted and the window is disposed of. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-7 Disposing of a text object

```
Boolean MyDisposeObject (WindowPtr theWindow)
{
    TXNObject   textObject = NULL;
    Boolean     okToClose = true;
    MyGetTextObject (theWindow, &textObject);                    // 1
    if (TXNGetChangeCount (textObject))                          // 2
```

```
        okToClose = MyDoSaveDialog (window, textObject);
    if (okToClose)                                                    // 3
    {
        TXNDeleteObject (textObject);
        DisposeWindow (theWindow);
    }
    return okToClose;
}
```

Here's what the code does:

1. Calls your function to obtain the text object associated with the window. If you saved the text object as a property of the window using `SetWindowProperty`, you can retrieve it by calling the Window Manager function `GetWindowProperty`.
2. Checks to see if the text object has changed. If so, calls your function to open the Save dialog and gives the user an opportunity to save the text object to a file.
3. If it is okay to close the window, calls the function `TXNDeleteObject` to dispose of the text object and calls the Window Manager function `DisposeWindow` to dispose of the window.

Handling File and Edit Menu Commands

This section shows how your application can handle commands from the File and Edit menus. MLTE provides a variety of functions to handle these menu commands, such as the functions `TXNCut` and `TXNSelectAll`. Regardless of the command, your application must call the appropriate MLTE function to handle the command and update the menu items.

Calling the Appropriate MLTE Function

If you've installed a window event handler to process menu commands, the Carbon event manager will call your handler whenever the user chooses a command from the menu. Your handler can then call the appropriate MLTE function to process the command.

Listing 3-8 shows how your application can handle window-related menu events from within a `MyWindowEventHandler` function. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-8 Handling menu commands from the File and Edit menus

```
pascal OSStatus MyWindowEventHandler ( EventHandlerCallRef myHandler,
                            EventRef event,
                            void * userData)                          // 1
{

     OSStatus   result,
                status = eventNotHandledErr;
    UInt32      eventClass = GetEventClass (event);                   // 2
    HICommand   command;
    TXNObject   textObject;

    switch (eventClass)
    {
        case kEventClassWindow:                                             // 3
        {
            // Your code to handle any window events
            // such as window-focus and window-relinquish events.
         }
        case kEventClassCommand:                                            // 4
        {
            GetEventParameter (event, kEventParamDirectObject,
                                typeHICommand, NULL,
                                sizeof (HICommand),
                                NULL, &command);                            // 5
            MyGetTextObject (theWindow, &textObject);                       // 6
            switch (command.commandID)                                      // 7
            {
                case kHICommandUndo:
                    TXNUndo (textObject);                                   // 8
                break;
                case kHICommandRedo:
                    TXNRedo (textObject);                                   // 9
                break;
                case kHICommandCut:
                    status = TXNCut (textObject);                           // 10
                break;
                case kHICommandCopy:
                    status = TXNCopy (textObject);                          // 11
                break;
                case kHICommandPaste:
                    status = TXNPaste (textObject);                         // 12
                break;
                case kHICommandClear:
                    status = TXNClear (textObject);                         // 13
                break;
                case kHICommandSelectAll:
                    TXNSelectAll (textObject);                              // 14
                break;
                case kHICommandPageSetup:
                    TXNPageSetup (textObject);                              // 15
                break;
                case kHICommandPrint:
                    TXNPrint (textObject);                                  // 16
                break;
                case kHICommandSave:
                    MyDoSave (textObject);                                  // 17
                break;
                case kHICommandSaveAs:
                    MyDoSaveAs (textObject);                                // 18
                break;
                case kHICommandClose:
                    MyDoCloseDoc (textObject);                              // 19
                break;
            } // switch command.commandID
            break;
         }//  case kEventClassCommand
    }//  switch eventClass
    MyUpdateMenus();                                                    // 20
    return status;
}
```

Here’s what the code does:

1. Declares parameters for the window event handler. When the Carbon Event Manager invokes your handler it passes an event reference from which you can determine the event. It also passes a reference constant. In this case, `userData` is a reference to the window on which the handler is installed.
2. Gets the event class from the event reference by calling the Carbon Event Manager function `GetEventClass`.
3. Checks for a window class event. You need this only if you want to handle any window events. If your application implements a Fonts window instead of a Font menu, you need to handle window focus and window relinquish events.
4. Checks for a command class event. This includes any command issued from any menu in the menu bar.
5. Obtains the parameters associated with a command event by calling the Carbon Event Manager function `GetEventParameter`. In this case, gets the HI command that triggered the event.
6. Calls your function to obtain the text object associated with the window. If you saved the text object as a property of the window using `SetWindowProperty`, you can retrieve it by calling the Window Manager function `GetWindowProperty`.
7. Checks for one of the standard command ID constants defined in the Carbon Event Manager.
8. Calls the function `TXNUndo` to undo the last user action. The undo stack is 32 levels deep
9. Calls the function `TXNRedo` to redo the last user action.
10. Calls the function `TXNCut` to delete the current selection and copy it to the private MLTE scrap.
11. Calls the function `TXNCopy` to copy the current selection to the private MLTE scrap.
12. Calls the function `TXNPaste` to paste the contents of the Clipboard into the text object. Before you call this function, you can call the function `TXNIsScrapPastable` to make sure the Clipboard contains data supported by the MLTE.
13. Calls the function `TXNClear` to delete the current selection. This function does not add the deleted selection to the private MLTE scrap.
14. Calls the function `TXNSelectAll` to select all text in the frame of a text object. You can check whether your application should enable the Select All menu item by calling the function `TXNDataSize` to see if the text object contains any data.
15. Calls the MLTE function `TXNPageSetup` to display the Page Setup dialog. This function handles all changes in response to page layout settings your user makes.
16. Calls the MTLE function `TXNPrint` to display the Print dialog. This function handles all changes in response to print settings your user makes, then prints the text associated with the text object.
17. Closes the window by calling your function to close the window and dispose of the text object.

18. Calls your function to save the text object. Your `MyDoSave` function should call the MLTE function `TXNSave` to save the text object and perform any clean-up tasks that are necessary.

19. Calls your function to save a copy of the text object. Your `MyDoSaveAs` function should call the MLTE function `TXNSave` to save the text object and perform any clean-up tasks that are necessary.

20. Calls your function to update the menu items so the items are enabled or disabled appropriated. See "Updating the File and Edit Menus" (page 38) for more information.

If your application uses a Font menu instead of a Fonts window, you also need to to check for the appropriate case (hierarchical or non hierarchical) and then execute the following code:

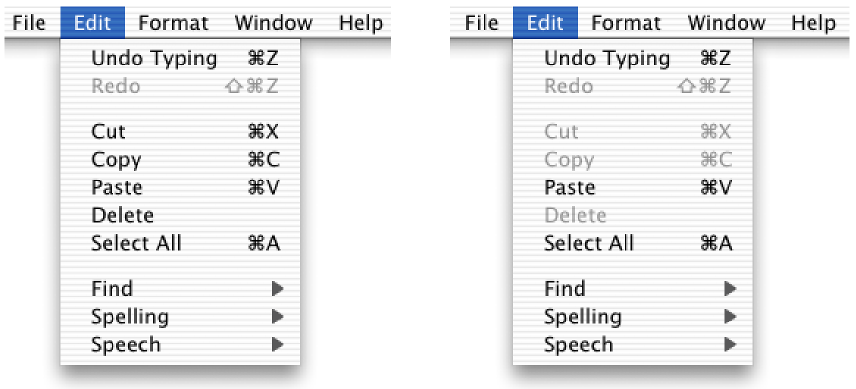
```
if (gTXNFontMenuObject!= NULL)
            status = TXNDoFontMenuSelection (textObject,
                             gTXNFontMenuObject,
                             menuID, menuItem);
```

Updating the File and Edit Menus

MLTE provides functions that your application can use to determine whether File and Edit menu items should be enabled or disabled. (When a menu item is enabled, a user can select it from the menu. When a menu item is disabled, it appears dimmed to the user.) Figure 3-1 shows two Edit menus, the first with more items enabled than the second.

Note: If your application uses an MLTE Font menu object, you do not need to update the Font menu. When your application handles the Font command, MLTE updates the menu automatically.

Figure 3-1 Edit menus with a variety of items enabled and disabled



The Cut, Copy, and Clear items from the Edit menu operate on selected text. Before you enable these items in the Edit menu, your application should check whether the user has selected text by calling the `TXNIsSelectionEmpty`function. If `TXNIsSelectionEmpty` returns `false` (meaning text is selected) then your application should enable the Cut, Copy, and Clear menu items.

The Paste item in the Edit menu should be enabled if the current scrap contains data supported by MLTE. You can test the current scrap by calling the `TXNIsScrapPastable`function. If `TXNIsScrapPastable` returns `true`, then your application should enable the Paste menu item.

The Select All menu item should be enabled if a text object contains any data at all. The `TXNDataSize`function returns the size of a text object. If the returned value is not zero, your application should enable the Select All menu item.

The Save menu item should be enabled if any changes were made to the text object since the text object was created or saved last. You can check for changes by calling the `TXNGetChangeCount`function. If the function returns a value greater than 0, then changes have been made.

The Undo and Redo command should be enabled if the previous action by the user is undoable or redoable. You can use the functions `TXNCanUndo` and `TXNCanRedo` to test whether these menu items should be enabled. These functions also return a value that indicates the action than can be undone or redone. You can use this information to customize the Edit menu. For example, `Figure 3-1` (page 38) shows an Undo Typing menu item instead of simple Undo menu item. See "Customizing MLTE Support for Carbon Events" (page 54) for information on how to provide a callback that customizes the Undo menu item.

Listing 3-9 shows a function that updates the File and Edit menu items. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-9 A function that updates the File and Edit menu items

```
OSStatus MyUpdateMenus ()
{
    TXNObject       textObject = NULL;
    TXNActionKey    actionKey;
    OSStatus        status = noErr;
    WindowRef       theWindow;

    if (theWindow = FrontWindow())
            MyGetTextObject (theWindow, &textObject);                    // 1
    if (TXNGetChangeCount (textObject) > 0 )                             // 2
    {
            EnableMenuCommand (NULL, kHICommandSave);
            EnableMenuCommand (NULL, kHICommandSaveAs);
    }
    else
    {
            DisableMenuCommand (NULL, kHICommandSave);
            DisableMenuCommand (NULL, kHICommandSaveAs);
    }
    if (theWindow != NULL)                                               // 3
    {
            EnableMenuCommand (NULL,  kHICommandPageSetup);
            EnableMenuCommand (NULL,  kHICommandPrint);
            EnableMenuCommand (NULL,  kHICommandClose);
    }
    else
    {
            DisableMenuCommand (NULL,  kHICommandPageSetup);
            DisableMenuCommand (NULL,  kHICommandPrint);
            DisableMenuCommand (NULL,  kHICommandClose);
    }

    if (!TXNIsSelectionEmpty (textObject))                               // 4
    {
            EnableMenuCommand (NULL,  kHICommandCut);
            EnableMenuCommand (NULL,  kHICommandCopy);
            EnableMenuCommand (NULL,  kHICommandClear);
    }
    else
    {
            DisableMenuCommand (NULL,  kHICommandCut);
            DisableMenuCommand (NULL,  kHICommandCopy);
            DisableMenuCommand (NULL,  kHICommandClear);
    }
    if (TXNIsScrapPastable () && (textObject != NULL))              // 5
                    EnableMenuCommand (NULL,  kHICommandPaste);
    else
                    DisableMenuCommand (NULL,  kHICommandPaste);
    if (TXNDataSize (textObject) > 0)                               // 6
                    EnableMenuCommand (NULL,  kHICommandSelectAll);
    else
                    DisableMenuCommand (NULL,  kHICommandSelectAll);
    if (TXNCanUndo (textObject, &actionKey))                        // 7
                    EnableMenuCommand (NULL,  kHICommandUndo);
    else
                    DisableMenuCommand (NULL,  kHICommandUndo);
    if (TXNCanRedo (textObject, &actionKey))                        // 8
                    EnableMenuCommand (NULL,  kHICommandRedo);
    else
                    DisableMenuCommand (NULL,  kHICommandRedo);
    return status;
}
```

Here's what the code does:

1. Calls the Window Manger function `FrontWindow` to obtain a reference to the window. If the window is not `NULL`, calls your function to obtain the text object associated with the window. If you saved the text object as a property of the window using `SetWindowProperty`, you can retrieve it by calling the Window Manager function `GetWindowProperty`.

2. Calls the function `TXNGetChangeCount` to get the number of times the text object has changed since the last time it was saved. If there have been changes, enables the Save and Save As items in the File menu. If not, these menu items are disabled. The Menu Manager functions `EnableMenuCommand` and `DisableMenuCommand` take two parameters, a menu reference and a command ID. If you pass `NULL` instead of a menu reference, the Menu Manager starts the search for the command at the root menu.

3. Makes sure the window is not `NULL`. If it is not `NULL`, you can assume the window is open and you should enable the Print, Page Setup, and Close items. Otherwise, disable these items.

4. Calls the function `TXNIsSelectionEmpty` to see whether the current selection is empty. If the selection is not empty, enables the Cut, Copy, and Clear items in the Edit menu. If the selection is empty, disables the menu items.

5. Calls the function `TXNIsScrapPastable` to see whether the Clipboard contains data supported by MLTE. If there is data that can be pasted, enables the Paste item in the Edit menu. If the data is not supported by MLTE, or there is no data available to paste, disables the menu item.

6. Calls the function `TXNDataSize` and checks to see if the size of the data in the text object is greater than 0. Any value other than zero indicates that the text object is not empty, so the code enables the Select All item in the Edit menu. If the text object is empty, disables the menu item.

7. Calls the function `TXNCanUndo` to see if the last user action can be undone. If the action can be undone, enables the Undo item in the Edit menu. Otherwise it disables the menu item. On output, this function provides a `TXNActionKey` value that identifies the action that can be undone. You can use this information to customize the Undo menu item with the name of the item than can be undone. See "Writing an Action Key Mapping Callback Function" (page 59) for more information.

8. Calls the function `TXNCanRedo` to see if the last user action can be redone. If the action can be redone, enables the Redo item in the Edit menu. Otherwise it disables the menu item.

Setting Font Size and Style

You can set font size and style for the current selection by calling the `TXNSetTypeAttributes`function. The current selection can be a range of text selected by the user or specified by your application. You specify the current selection with the `iStartOffset` and `iEndOffset` parameters of the `TXNSetTypeAttributes` function.

MLTE specifies a type attribute using a **triple**. A triple is an attribute tag, the size of the attribute, and a value for the attribute. Attribute tags are constants. The attributes you can specify are described in *Inside Mac OS X: MLTE Reference* and include the following attribute tags for font size and style:

- `kTXNQDFontSizeAttribute`
- `kTXNQDFontStyleAttribute`

The following constants specify attribute sizes for font size and style:

- `kTXNFontSizeAttributeSize`
- `kTXNQDFontStyleAttributeSize`

MLTE stores the triple that specifies a type attribute in a `TXNTypeAttributes`structure. This structure contains the `TXNAttributeData`union, and it is that union you use to pass the triple that specifies a type attribute. The `tag` field of the `TXNTypeAttributes` structure defines the type of data in the `TXNAttributeData` union, and the `size` field of the `TXNTypeAttributes` structure defines the size of the data in the `TXNAttributeData` union.

Typically your application would have a function that handles size and style selections whether these selections occur in Size and Style menus or in the Size and Style columns of a Fonts window. If a user selects a new font size or style, you change the appropriate values in the `TXNTypeAttributes` structure, then you call the `TXNSetTypeAttributes` function to change the attributes for the current selection.

The code fragment in Listing 3-10 shows how your application can change size and style changes for the current text selection. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-10 Changing size and style attributes for selected text

```
OSStatus ChangeSizeAndStyleAttributes (TXNObject myTextObject,
                            Fixed myNewSize, Style myNewStyle);
{
    OSStatus status = noErr;
    TXNTypeAttributes typeAttr[2];
    typeAttr[0].tag = kTXNQDFontSizeAttribute;                                  // 1
    typeAttr[0].size = kTXNFontSizeAttributeSize;
    typeAttr[0].data.dataValue = myNewSize;
    typeAttr[1].tag = kTXNQDFontStyleAttribute;                                 // 2
    typeAttr[1].size = kTXNQDFontStyleAttributeSize;
    typeAttr[1].data.dataValue = myNewStyle;
    status = TXNSetTypeAttributes(myTextObject, 2, typeAttr,
                            kTXNUseCurrentSelection,
                            kTXNUseCurrentSelection);                           // 3
    return status;
}
```

Here's what the code does:

1. Assigns a triple (attribute tag, size of the attribute, and attribute value) to specify the font size.
2. Assigns a triple (attribute tag, size of the attribute, and attribute value) to specify the font style.
3. Calls the function `TXNSetTypeAttributes` to set the size and style attribute for the current selection.

Handling Multiple Text Objects

When your application manages multiple text objects, it must make sure the appropriate text object has user focus. The **user focus** is the part of your application's user interface toward which keyboard input is directed. You can bring the appropriate text object into user focus by calling the `TXNFocus`function.

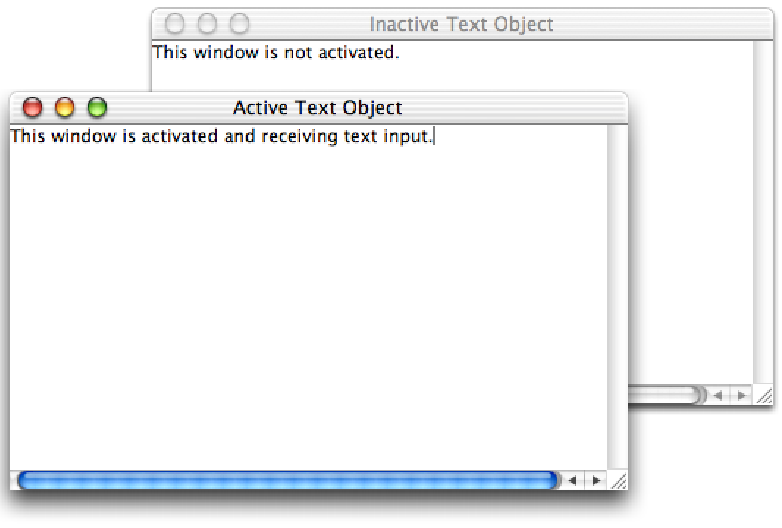
When you have multiple text objects you may need to handle the activation state of the text objects as well as the user focus. The activation state is independent of the user focus and can be turned on or off by calling the function `TXNActivate`.

Keep in mind that the `TXNActivate` function does not change the user focus. You typically call the `TXNActivate` function when you have multiple text objects in a window and you want all of them to be scrollable even though only one at a time can have user focus. This lets application users scroll the inactive text without changing the focus from another text area.

Note that if your application sets the `iActiveState` parameter of the `TXNActivate` function to `kScrollBarsAlwaysActive`, the scroll bars are active even when the text object does not have user focus. Before you call `TXNActivate`, you should call the function `TXNFocus` to focus the scroll bars and insertion point so they become active or inactive, depending on whether you want the text object to obtain user focus.

When your application brings a text object into user focus by calling the `TXNFocus`function, the function removes any visual indication of its inactive state and then sets the state of the scroll bars so they are drawn correctly in response to update events. Figure 3-2 (page 43) shows examples of text objects that are activated (and have user focus) and deactivated (without user focus).

Figure 3-2 An activated and a deactivated text object, with and without user focus


Inactive Text Object
This window is not activated.
Active Text Object
This window is activated and receiving text input.

Listing 3-11 shows a `MyDoUserFocus` function that handles user focus. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-11 Handling an activation event

```
void MyDoUserFocus (WindowPtr theWindow, Boolean  becomingActive)
{
    TXNObject           textObject;
    TXNFrameID          frameID     =   0;
    OSStatus            status      =   noErr;

    MyGetTextObject (theWindow, &textObject);                              // 1
    MyGetFrameID (theWindow, &frameID);                                    // 2
    TXNFocus (textObject, becomingActive);                                 // 3

    if (becomingActive)                                                    // 4
        {
            TXNActivate (textObject, frameID, kScrollBarsAlwaysActive);
            MyUpdateMenus ();
        }
    else                                                                   // 5
            TXNActivate (textObject, frameID, kScrollBarsSyncWithFocus);
    }
```

Here's what the code does:

1. Calls your function to obtain the text object associated with the window. If you saved the text object as a property of the window using `SetWindowProperty`, you can retrieve it by calling the Window Manager function `GetWindowProperty`.
2. Calls your own function to get the frame ID of the text object associated with the window.
3. Calls the function `TXNFocus` to change the user focus of the scroll bars and insertion caret to the state specified by the `becomingActive` parameter.

4. If the text object obtains user focus, calls the function `TXNActivate` to set the state of the scroll bars so they are drawn correctly. Then, calls your function to adjust the menus so menu items are enabled and disabled appropriately.

5. If the text object loses user focus, calls the function `TXNActivate` to synchronize the activity state of the scroll bars with the focus state of the frame. In this case, only when the frame has user focus does the frame have active scroll bars.

Posting an Alert

When an alert is posted, the Dialog Manager starts a sub-event loop. Keyboard input is instead returned to the Dialog Manager, which in most cases ignores the input. Keyboard input is not sent to your application and does not appear in your application's document.

MLTE uses event handlers to receive input directly from the Unicode input method. Keyboard events don't flow through the event loop and are not returned as event records. MLTE receives keyboard input and places the input into the active MLTE text object (`TXNObject`).

When your application deactivates an MLTE text object, the keyboard input handlers are removed. Typically, your application should deactivate an MLTE text object when it receives a deactivate event for the window that contains the object. When the user switches between two document windows, the application receives the deactivate event in its main event loop, deactivates the old MLTE text object, activates the new MLTE text object, and keyboard input goes to the newly active document window.

When you post an alert, however, the Dialog Manager's sub-event loop receives the deactivate event for the document window. Your application doesn't see the event so it doesn't deactivate the MLTE text object associated with the document window. When a user presses keys, the input is sent to the MLTE text object instead of the Dialog Manager's sub-event loop. Because the MLTE text object is active, it still has its event handlers installed.

To avoid this situation, you must provide a modal event filter callback to the Dialog Manger Alert function. The Dialog Manager passes the deactivate event for your application's window to the modal event filter callback, and the callback deactivates the `TXNObject`. Once the MLTE text object is deactivated by the callback, the document window no longer receives keystrokes while the Alert dialog is open.

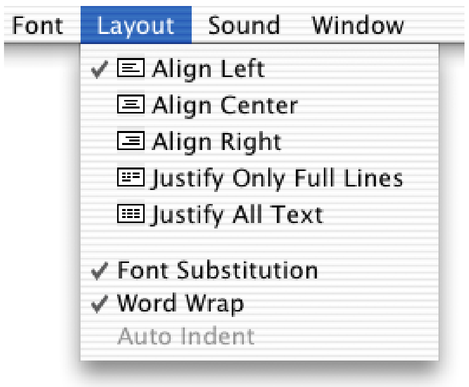
Working With Document-Wide Settings

MLTE has a variety of settings that apply to an entire text object, such as line direction, tab values, read-only privileges, keyboard synchronization, automatic indentation, word wrap, caret display, font substitution, and type of input for input methods. You can get and set global settings with the `TXNGetTXNObjectControls` and `TXNSetTXNObjectControls`functions. This section shows you how to implement code for changing two document-wide settings—word wrap and line justification. You can take the same approach shown here to implement code that changes other document-wide settings.

Implementing Word Wrap

Your application can provide a Layout menu that allows the user to enable and disable automatic word wrap. Once you create a Layout menu that contains a word-wrap item, you can indicate to the user whether automatic word wrap is enabled or disabled by displaying a check mark or other visual indicator next to the item if it is enabled. Figure 3-3 (page 45) shows a Layout menu that has the Word Wrap item checked.

Figure 3-3 Using check marks to show enabled Layout settings



When a user changes the word-wrap setting, your application can use the `TXNGetTXNObjectControls` and `TXNSetTXNObjectControls` functions to get the current setting and then to toggle the setting appropriately. These functions are used to get and set a number of document-wide settings, so you need to use the `iControlTags` parameter to specify that automatic word wrap is the setting you want to change. MLTE provides a constant called `kTXNWordWrapStateTag` that you can use for the `iControlTags` parameter.

You use the `iControlData` parameter of the `TXNSetTXNObjectControls` function to specify the new value of the word-wrap setting. MLTE provides two constants called `kTXNAutoWrap` and `kTXNNoAutoWrap` that you can use to specify the word-wrap state.

Listing 3-12 shows a `MyDoWordWrap` function that your application can call from a menu-handling function to enable or disable word wrap in response to what your user selects from a Layout menu you create. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-12 Toggling the word-wrap setting

```
OSStatus MyToggleWordWrap (WindowRef theWindow)
{
    TXNObject   textObject = NULL;
    OSStatus    status = noErr;
    TXNControlTag               controlTag[1];
    TXNControlData              controlData[1];

    MyGetTextObject (theWindow, &textObject);                       // 1
    controlTag[0] = kTXNWordWrapStateTag;                           // 2
    status = TXNGetTXNObjectControls (textObject, 1,
                        controlTag, controlData);                   // 3
    if (controlData[0].uValue == kTXNAutoWrap)                      // 4
            controlData[0].uValue = kTXNNoAutoWrap;
    else                                                            // 5
```

```
            controlData[0].uValue = kTXNAutoWrap;
    status = TXNSetTXNObjectControls (textObject, false, 1,
                                controlTag, controlData);                    // 6
    return status;
}
```

Here's what the code does:

1. Calls your function to obtain the text object associated with the window. If you saved the text object as a property of the window using `SetWindowProperty`, you can retrieve it by calling the Window Manager function `GetWindowProperty`.

2. Assigns the word-wrap state as the formatting data to obtain.

3. Calls the function `TXNGetTXNObjectControls` to obtain the current value of the word-wrap state for the text object associated with the specified window.

4. If the current state is automatic word wrap, set the value to no automatic word wrap.

5. Otherwise, set the value to enable automatic word wrap.

6. Calls the function `TXNSetTXNObjectControls` to set the value of the word-wrap state for the text object. The second value passed to this function (`false`) indicates that none of the other formatting and privileges attributes should be reset to their default value.

Implementing Line Justification

Your application can provide a Layout menu that allows the user to select a line justification setting. Once you create a menu that contains line justification items, you can indicate to the user which justification setting is active by displaying a check mark or other visual indicator next to the item if it is enabled. Figure 3-3 (page 45) shows a Layout menu that has the Justify Full item checked.

When a user changes the line justification setting, your application can use the `TXNGetTXNObjectControls` and `TXNSetTXNObjectControls` functions to get the current setting, then change the setting appropriately. These functions are used to get and set a number of document-wide settings, so you need to use the `iControlTags` parameter to specify that line justification is the setting you want to change. MLTE provides a constant called `kTXNJustificationTag` that you can use for the value of the `iControlTags` parameter.

You use the `iControlData` parameter of the `TXNSetTXNObjectControls` function to specify the new value of the line justification setting. MLTE provides six constants that you can use to specify the value—`kTXNFlushDefault`, `kTXNFlushLeft`, `kTXNFlushRight`, `kTXNCenter`, `kTXNFullJust`, and `kTXNForceFullJust`. The `kTXNFlushDefault` constant indicates text should be flush according to the line direction. The constant `kTXNForceFullJust` indicates that every line of text, including the last line, should be flush left and right.

Listing 3-13 shows a `MyDoJustification` function that your application can call from a menu-handling function to implement the type of justification your user selects from a Layout menu you create. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-13 A function that sets line justification

```
OSStatus MyDoJustification (WindowRef theWindow,
                            SInt32 myNewJustification)
```

```
{
    TXNObject       textObject = NULL;
    OSStatus        status = noErr;
    TXNControlTag   controlTag[1];
    TXNControlData  controlData[1];

    MyGetTextObject (theWindow, &textObject);                       // 1
    controlTag[0] = kTXNJustificationTag;                           // 2
    status = TXNGetTXNObjectControls (textObject, 1,
                            controlTag, controlData);               // 3
    if (controlData[0].sValue != myNewJustification)
    {
        controlData[0].sValue = myNewJustification;                 // 4
        status = TXNSetTXNObjectControls (textObject, false, 1,
                            controlTag, controlData);               // 5
    }
    return status;
}
```

Here's what the code does:

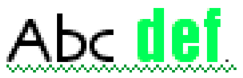
1. Calls your function to obtain the text object associated with the window. If you saved the text object as a property of the window using `SetWindowProperty`, you can retrieve it by calling the Window Manager function `GetWindowProperty`.
2. Assigns justification as the formatting data to get.
3. Calls the `TXNGetTXNObjectControls` to obtain the current justification value for the text object.
4. If the justification value is not equal to that passed to the function, assigns the new value.
5. Calls the function `TXNSetTXNObjectControls` to set the new justification value.

Advanced Topics

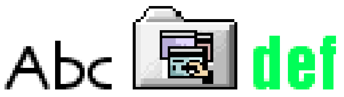
Working With Embedded Objects

Embedded objects are graphics, movie, or sound data embedded in text data. An embedded object is represented by one character offset in a text object. The offset does not represent the size of the embedded object, it is merely a placeholder for the object. This means you cannot use the number of offsets to calculate the size of the text object if your text object contains embedded data. If you need to calculate the size, you must write your own function to do so.

The `TXNGetData`function does not copy data that cross a data type boundary. But it may be used to copy data that cross a text-run boundary. For example, suppose your document consists of this:



If you call the `TXNGetData` function with a starting offset of 0 and an ending offset of 6, the returned data would contain the characters "abc def" even though the offsets cross a style-run boundary.

However suppose your document consists of this: 

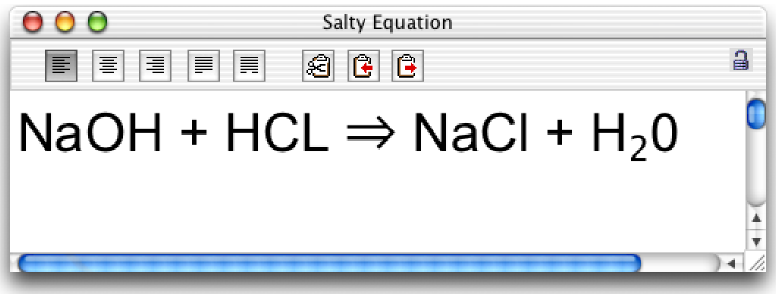
If you call the `TXNGetData` function with a starting offset of 0 and an ending offset of 6, the function returns the result code `kTXNIllegalToCrossDataBoundariesErr`.

If you are not sure whether a text object contains embedded data, you can use the `TXNCountRunsInRange`function to determine how many runs are in the text object. If there is more than one run in the range, you can then use the `TXNGetIndexedRunInfoFromRange`function to determine if the runs contain different types of data. The `oRunDataType` parameter returns the data type of each run. Once you know the run type, you can use the function `TXNGetData` to copy the run data.

Displaying Chemical Equations

Chemical equations, such as the one displayed in Figure 3-4, use subscripts. MLTE treats a subscript as a font attribute, so you use the function `TXNSetTypeAttributes` to set the attribute that controls subscripts.

Figure 3-4 Chemical equation that uses a subscript



$NaOH + HCL \Rightarrow NaCl + H_20$

To subscript the 2 shown in Figure 3-4 you would do the following

- assign values to the `TXNTypeAttributes` structure to specify the attribute tag, size, and data
- call the MLTE function `TXNSetTypeAttributes`

You can use the ATSUI constant `kATSUCrossStreamShiftTag` as the attribute tag. This constant specifies a **cross-stream shift**—a shift in a character's position in the direction against the reading direction. In the case of Figure 3-4, this is a vertical shift.

The data value associated with the cross-stream shift tag is the amount by which the position should be changed with respect to the **base line**. Values can range from -1.0 to 1.0, with negative values indicating that a character should be drawn lower than the base line. (See *Inside Mac OS X: Rendering Unicode Text With ATSUI* for more information on stream shift attribute tags and values.)

Listing 3-14 shows how to set text attributes to display a character as a subscript. If you want to display the equation subscript with a smaller font size than the size used for the chemical symbol to which the subscript is associated, you need to change the size attribute for the subscript to an appropriate value. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-14 Setting attributes for a subscript

```
#define myFloatToFixed (a) ((Fixed) ((float) (a) * fixed1))

OSErr   status;
TXNTypeAttributes typeAttr[1];
typeAttr[0].tag = kATSUCrossStreamShiftTag;                          // 1
typeAttr[0].size = sizeof(Fixed);                                    // 2
typeAttr[0].data.dataValue = myFloatToFixed(-0.35);                  // 3
status = TXNSetTypeAttributes (myTextObject, 1, typeAttr,
                kTXNUseCurrentSelection, kTXNUseCurrentSelection);   // 4
```

Here's what the code does:

1. Assigns cross-stream shift as the tag.
2. Assigns an attribute size. The size of the value associated with the cross-stream shift tag is 4-byte fixed.
3. The value of the cross-stream shift must be between -1. and 1.0. A subscript should be negative. You may need to convert the value type to a fixed value, as shown here.
4. Calls the function `TXNSetTypeAttributes` to set the attribute. You can use the constant `kTXNUseCurrentSelection` to specify the current selection if you are applying the attribute to user-selected text. Otherwise, you should specify starting and ending offset values for the character you want to subscript.

Accessing and Displaying Advanced Typographical Features

Your application can use MLTE to display advanced typographical features (such as ligatures, diacritical marks, and diphthongs). However, you need to be familiar with font features described in *Inside Mac OS X: Rendering Unicode Text With ATSUI*.

To have your application provide your users with the ability to apply advanced typographical features to text they select, your application needs to do the following:

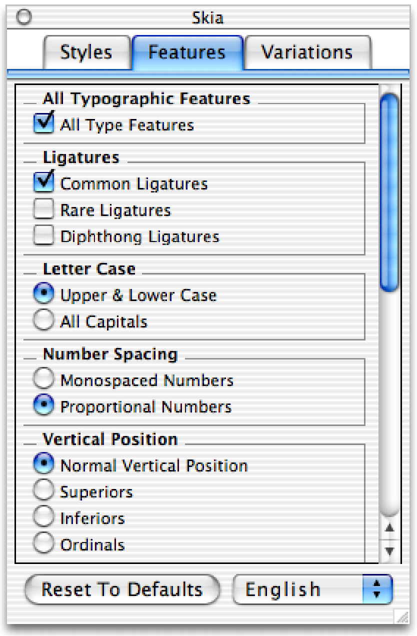
- provide an interface for users to select advanced features
- respond to user-selected events for advanced features
- translate user selections into the appropriate font feature type and font feature selector constants
- assign the constants to the `TXNTypeAttributes` structure
- call the MLTE function `TXNSetTypeAttributes`
- update the display

Note: Not all features are available for a font. The set of available features are determined by the font designer. You can provide users access only to those features the font designer has included with the font.

Providing an Interface for Users to Select Advanced Features

Your application can provide an interface that lets users view and select advanced font styles, font features, and font variations. One approach is to create a typography dialog that has three panes—Styles, Features, and Variations. This would allow users control over all the advanced ATSUI typography that can be set using the MLTE function `TXNSetTypeAttributes`. Figure 3-5 shows a sample features dialog. The dialog lets users access a few of the ATSUI features for the Skia font that can be set using the MLTE function `TXNSetTypeAttributes`. Depending on your users' needs, your application could create a feature dialog that lists all ATSUI features that can be set for the current font, or you could limit user-selectable font features to those most important to your users.

Figure 3-5 A Features dialog for the Skia font


Skia
Styles | Features | Variations

All Typographic Features
[✓] All Type Features

Ligatures
[✓] Common Ligatures
[ ] Rare Ligatures
[ ] Diphthong Ligatures

Letter Case
(●) Upper & Lower Case
( ) All Capitals

Number Spacing
( ) Monospaced Numbers
(●) Proportional Numbers

Vertical Position
(●) Normal Vertical Position
( ) Superiors
( ) Inferiors
( ) Ordinals

Reset To Defaults | English

Responding to User Selected Events for Advanced Features

Your application then responds to user selections from the features dialog. Depending on how you set up the user interface, your application can respond on an item-by-item basis, or can apply, at the same time, all changes specified in the dialog.

Translating the User's Selections to Constants

Translating the user's selections into the appropriate font feature type and font feature selector constants requires some research as you write your application. You need to be familiar with the font features and font variations constants that are described in *ATSUI Programming Guide*.

For example, imagine your user has just made changes to the Ligatures section of the Features pane shown in Figure 3-5 (page 50). Your application should represent the ligatures feature by using the `kLigaturesType` constant.

Note: Although there are constants defined for most font features, this does not mean a font has that feature. You must query a font to see whether or not a specific feature is available.

To represent the specific ligature selections made by the user, you need constants that indicate that rare, common, and diphthong ligatures are enabled, while logo and rebus ligatures are disabled. So your application would use the following font feature type constants: `kRareLigaturesOnSelector`, `kCommonLigaturesOnSelector`, `kDiphthongLigaturesOnSelector`, `kLogosOffSelector`, and `kRebusPicturesOffSelector`.

You would take a similar approach for any font feature or font variation. First, look up the constant that represents the feature or variation category. Then look up the constants that represent the state of each feature or variation in that category.

Assigning Constants to the Type Attributes Data Structure

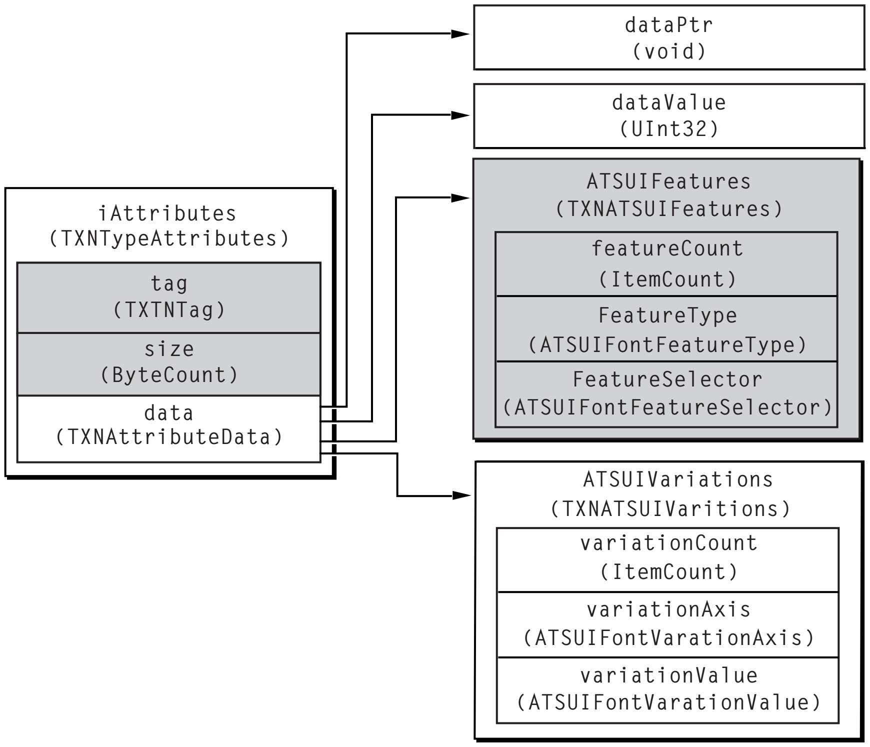
You can use the MLTE function `TXNSetTypeAttributes` to set a variety of features from simple to the most complex ATSUI features. The `iAttributes` parameter of the `TXNSetTypeAttributes` function is an array of `TXNTypeAttributes` structures that you use to indicate what features you want to set and the values to which the features should be set.

A `TXNTypeAttributes` structure has a `TXNAttributeData` union as one of its fields. The kind of attribute your application needs to set determines the data contained in the union. For ATSUI features, your application should supply `atsuFeatures` data, as defined by the `TXNATSUIFeatures` data structure. The `TXNATSUIFeatures` structure contains information about the number of features in the structure, along with pointers to the feature type and selector information (that is, whether a feature is enabled or not).

You can also use the MLTE `TXNSetTypeAttributes` function to set font variations data. In this case, your application would supply `atsuVariations` data in the `TXNAttributeData` union.

Figure 3-6 shows the `TXNTypeAttributes` structure and its fields. The shaded areas show the fields for which your application needs to provide data in order to change the ATSUI font features for a selection.

Figure 3-6 The fields for which your application needs to supply data in order to set ATSUI font features (shaded areas)



Once your application identifies the kind of data (feature or variation) it needs to set, then it needs to assign the appropriate values to the `tag` and `size` fields of the `TXNTypeAttributes` structure. The `tag` field determines the kind of data in the `TXNAttributeData` structure and the `size` field indicates the size of the attribute data. If your application needs to set ATSUI feature data, it would use the constant `kTXNATSUIFontFeaturesAttribute` for the `tag` field. It would use `sizeof(TXNATSUIFeatures)` for the `size` field. See *Inside Mac OS X: MLTE Reference* for a description of the font run attribute constants you can use for the `tag` field and for a description of the font run attribute size constants you can use for the `size` field.

Finally, your application needs to supply ATSUI feature data. You need only take the constants you identified based on the user's selections (see "Translating the User's Selections to Constants" (page 50)) and assign them to the appropriate fields of the `TXNATSUIFeatures` structure.

Listing 3-15 shows how to assign parameter values that set up diphthongs. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-15 Assigning parameter values for diphthongs

```
TXNTypeAttributes        typeAttr[1];
TXNATSUIFeatures         myFeatures;
TXNObject                textObject;
ATSUFontFeatureType      myFeatureType[1];
ATSUFontFeatureSelector myFeatureSelector[1];

myFeatureType[0] = kLigaturesType;                                    // 1
myFeatureSelector[0] = kDiphthongLigaturesOnSelector;                 // 2
```

```
myFeatures.featureCount     = 1;                                               // 3
myFeatures.featureTypes = myFeatureType;                                       // 4
myFeatures.featureSelectors = myFeatureSelector;                               // 5

typeAttr[0].tag = kTXNATSUIFontFeaturesAttribute;                              // 6
typeAttr[0].size = sizeof (TXNATSUIFeatures);                                  // 7
typeAttr[0].data.atsuFeatures   = (TXNATSUIFeatures *)&myFeatures;             // 8
```

Here's what the code does:

1. Assigns the ligatures feature type.
2. Assigns the diphthong ligatures on selector. Constants that represent font feature types and selectors are declared in the header file SFNTLayoutTypes.h, and are fully described in *ATSUI Programming Guide*.
3. Assigns the number of features in `myFeatures` to be 1; `myFeatures` is a `TXNATSUIFeatures` data structure.
4. Assigns the feature type.
5. Assigns the feature selector.
6. Assign the attribute tag value to be an ATSUI font feature attribute.
7. Assigns the attribute size to be the size of the `TXNATSUIFeatures` structure
8. Sets the value of the data field associated with the ligatures feature.

Calling the MLTE Set Type Attributes Function

Once you have determined which constants to use and to which fields of the `TXNTypeAttributes` structure the constants should be assigned, the call to the MLTE function `TXNSetTypeAttributes` is straightforward. In addition to the `TXNTypeAttributes` structure, you need to pass the current text object and the starting and ending offsets of the current selection.

Listing 3-16 Calling the MLTE function to set type attributes

```
OSStatus status;
status = TXNSetTypeAttributes (textObject, 1, typeAttr,
                kTXNUseCurrentSelection, kTXNUseCurrentSelection);
```

Updating the Display

Your application needs to update the display to reflect your user's selections. You need to check or uncheck items in the features dialog to reflect whether the item is now enabled or disabled.

Supporting Monostyled Text

This section describes how you can use MLTE to create custom, editable controls that use text in a single style. Beginning with Mac OS X version 10.2, MLTE supports monostyled text when typing, copying and pasting, and dragging and dropping styled text into a control. You can specify that MLTE uses monostyled text by using the `TXNFrameOptions` flag `kTXNMonostyledTextMask`.

When you set this option for an object such as a control, the text in the object has a single style no matter what changes the users makes to the text.

To set monostyled text, follow these steps:

1. Call the function `TXNNewObject` with the `kTXNMonostyledTextMask` option.

   If you obtain data from a file at the same time you create the text object, the style information in the file is ignored. It is preferable that you first create the text object, set its style, and then set data into the text object by calling the function `TXNSetDataFromFile`.

   Alternatively, you can create the text object without attaching it to a window, set the data, then set the style, and finally attach the text object to a window.

   Note: Font substitution is enabled by default when the `kTXNMonostyledTextMask` option is set.

2. Set the style of the monostyled text by calling the function `TXNSetTypeAttributes`. MLTE ignores any starting and ending offsets you supply to this function as the style of the monostyled text is set for all the text associated with the text object.

Customizing MLTE Support for Carbon Events

In Mac OS X version 10.1 or later, MLTE automatically sets up Carbon event handlers for text input and window events. See "What's Installed On a Text Object" (page 55) for details. However, there are a few situations in which you might want to configure Carbon events in MLTE:

- Your application runs on a release that's earlier than Mac OS X version 10.1.

  In this case, you need to specify that MLTE use Carbon events instead of Apple events to handle text input and window events, because MTLE does not automatically set up such support in versions 10.1 and earlier. Read the sections "Building a Dictionary" (page 55), "Instantiating the Carbon Events Structure" (page 57), and "Calling the function TXNSetTXNObjectControls" (page 58).

- You need MLTE to support other Carbon events, namely command and Font menu events.

  In this case, you use the data structure `TXNCarbonEventInfo` to specify that MLTE use Carbon events to handle these simple event-driven tasks that you would otherwise need to route to MLTE through function calls to the MLTE API. Read the sections "Building a Dictionary" (page 55), "Instantiating the Carbon Events Structure" (page 57), and "Calling the function TXNSetTXNObjectControls" (page 58).

- You want to use your own action key mapping callback function to customize the Undo and Redo menus.

In this case, you pass your callback function to MTLE through the `TXNCarbonEventInfo` data structure. For information on writing the callback, see "Writing an Action Key Mapping Callback Function" (page 59). Then read the sections "Building a Dictionary" (page 55), "Instantiating the Carbon Events Structure" (page 57), and "Calling the function TXNSetTXNObjectControls" (page 58).

- You don't want to use Carbon events or you want to handle all Carbon events in your application.

  You should let MLTE support Carbon events, but in the rare case you don't want MLTE to use its Carbon event handlers, you can turn off MLTE automatic support for them. Read the section "Turning Off MLTE Support for Carbon Events" (page 59).

What's Installed On a Text Object

MLTE installs Carbon event handlers each time you create an MLTE text object (`TXNObject`). The handlers that are set up for a text object (`TXNObject`) depend on the parameters you pass to the function `TXNNewObject`.

If you call the function `TXNNewObject` with the parameter `iFrame` set to `NULL`, handlers are set up for all text input events and the following window events:

- `kEventWindowActivated`
- `kEventWindowDeactivated`
- `kEventWindowDrawContent`
- `kEventWindowClickContentRgn`
- `kEventWindowResizeCompleted`

If you call the function `TXNNewObject` with the parameter `iWindow` set to `NULL`, the Carbon event handlers aren't installed until you call the function `TXNAttachObjectToWindow` with a valid window pointer. The Carbon event handlers are automatically removed if you call the function `TXNAttachObjectToWindow` with a `NULL` window pointer.

If you call the function `TXNNewObject` with the parameter `iFrame` set to a valid `Rect`, or if you call the functions `TXNSetViewRect`, `TXNSetFrameBounds`, or `TXNSetRectBounds`, then the default Carbon event handlers for window events are only the following:

- `kEventWindowDrawContent`
- `kEventWindowClickContentRgn`

The reason for this is that the text object is now association with a frame, not the entire window.

Building a Dictionary

If you want MLTE to support client-specified Carbon events, you need to build a dictionary whose keys are strings that represent events (such as "CommandUpdate" or "WindowResize") and whose values are event target references associated with the events. A **dictionary** is a set of key-value pairs. (An **event target** is the interface object (window, menu, and so forth) in which an event occurs.) See *Inside Mac OS X: MLE Reference* for a list of the predefined keys you can use to build the dictionary.

There are two additional items that dictionary can contain:

- a font menu object (`TXNFontMenuObject`)
- a universal procedure pointer (`TXNActionKeyMapperUPP`) to a callback function you provide to handle action key mapping events

You use an action key mapping callback to dynamically change the edit menu to reflect what actions can be undone and redone (for example, Undo Paste instead of Undo). You'll see how to create an action key mapping callback in the section "Writing an Action Key Mapping Callback Function" (page 59).

If dictionary values are event target references or a font menu object, how is it possible to provide a universal procedure pointer (UPP) or a `TXNFontMenuObject` as a dictionary value? These data types use the same amount of space as an event target reference uses. So as long as you cast either of them to be of type `EventTargetRef`, the compiler won't complain. Although casting is generally not recommended, in this case you can be confident that once MLTE processes the dictionary, the UPP or `TXNFontMenuObject` is cast back to the appropriate type.

Listing 3-17 shows a `MyBuildTargetsDictionary` function that builds a dictionary for of Carbon events that MLTE should handle. When you create your own function to build a dictionary, you can add as many of the Carbon events supported by MLTE as you'd like. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-17 Building a dictionary

```
static CFDictionaryRef MyBuildTargetsDictionary(
        WindowRef               targetWindow
        MenuRef                 editMenu,
        TXNFontMenuObject       fontMenuObj)
{
        CFStringRef     keys[] = {  kTXNCommandTargetKey,                     // 1
                                    kTXNCommandUpdateKey,
                                    kTXNActionKeyMapperKey,
                                    kTXNFontMenuObjectKey };

        EventTargetRef  values[4];
        OSStatus        status;
        MenuRef         theFontMenuRef;

    values[0] = GetWindowEventTarget (targetWindow);                          // 2
    values[1] = GetMenuEventTarget (editMenu);                                // 3

    if (gTXNActionKeyMapperUPP == NULL)                                       // 4
        gTXNActionKeyMapperUPP =                       NewTXNActionKeyMapperUPP
 (ActionKeyMappingProc);
    values[2] = (EventTargetRef) gTXNActionKeyMapperUPP;                      // 5
    values[3] = theFontMenuObj;                                               // 6

    return CFDictionaryCreate (kCFAllocatorDefault,
                        (const void **)&keys,
                        (const void **)&values,
                        4,
                        &kCFCopyStringDictionaryKeyCallBacks,
                        NULL);                                                // 7
}
```

Here's what the code does:

1. Defines the keys that will be used in the dictionary. The keys you can use are defined in *Inside Mac OS X: MLTE Reference*. Each key must be a `CFStringRef`, which these predefined keys are.
2. Defines the value associated with the first key (`kTXNCommandTargetKey`). Each entry in the `values` array must specify the event target associated with the key of the same index. The target for an edit command event is the `WindowRef` to which the text object is attached.
3. Defines the value associated with the second key (`kTXNCommandUpdateKey`). The target for an edit command update event is the Edit menu.
4. Defines a universal procedure pointer (`TXNActionKeyMapperUPP`) to your callback function. This is a global UPP which is created the first time the function `NewTXNActionKeyMapperUPP` is called. See "Writing an Action Key Mapping Callback Function" (page 59) for more information on this callback.
5. Defines the value associated with the third key (`kTXNActionKeyMapperKey`). This is the UPP, but you must cast it to an `EventTargetRef`.
6. Defines the value (`TXNNewFontMenuObject`) associated with the fourth key (`kTXNFontMenuObjectKey`).
7. Calls the Core Foundation Collection Services function `CFDictionaryCreate` to create the dictionary and return a reference to it. The caller of the `MyBuildTargetsDictionary` function must release the dictionary after it has been passed to MLTE.

Instantiating the Carbon Events Structure

So far you've built a dictionary that contains information about the specific Carbon events and event targets you want MLTE to handle. You need to pass this dictionary to a `TXNObject` you've created, along with additional information needed to set the state of the `TXNObject` to use Carbon events. You use the MLTE function `TXNSetTXNObjectControls` to change the state of the `TXNObject`. Before you call this function, you need to instantiate the data structure `TXNCarbonEventInfo` with the information needed to set up Carbon events.

The code fragment shown in Listing 3-18 shows you how to do this. The next section will show you how to pass this structure to MLTE. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-18 Instantiating MLTE's Carbon events data structure

```
TXNCarbonEventInfo      carbonEventInfo;

carbonEventInfo.useCarbonEvents = true;                                  // 1
carbonEventInfo.filler = 0;                                              // 2
carbonEventInfo.flags = 0;                                               // 3
carbonEventInfo.fDictionary = MyBuildTargetsDictionary (aWindow,
                                                theEditMenu,
                                                aTXNFontObject);         // 4
```

Here's what the code does:

1. Turns on support for Carbon events by setting `useCarbonEvents` to `true` in the `TXNCarbonEventInfo` data structure.

2. Sets the filler value to `0`. The filler is just that, a value that's not used for anything.
3. Sets the flags value to `0`, as there is currently no other value you should supply here.
4. Calls the dictionary-building function `MyBuildTargetsDictionary` created in "Building a Dictionary" (page 55).

Calling the function TXNSetTXNObjectControls

The function `TXNSetTXNObjectControls` is what you call to request a change to the state of a `TXNObject`. You must supply tags that identify the data you are passing to the function. To set up Carbon events, you must provide a tag to specify you are passing Carbon events data.

Listing 3-19 shows the code you need to call the function `TXNSetTXNObjectControls`. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-19 Changing the state of a text object

```
TXNControlTag           iControlTags[] = { kTXNUseCarbonEvents };               // 1
TXNControlData          iControlData[1];                                        // 2
iControlData[0].uValue = (UInt32) &carbonEventInfo;                             // 3


status = TXNSetTXNObjectControls(
                        MyTextObject,                                           // 4
                        false,                                                  // 5
                        1,                                                      // 6
                        iControlTags,
                        iControlData);
CFRelease (carbonEventInfo.fDictionary);                                        // 7
```

Here what the code does:

1. Sets the control tag to the constant `kTXNUseCarbonEvents`. This constant specifies that the data in the `iControlData` parameter is a `TXNCarbonEventInfo` structure.
2. Declares a control data array. `TXNControlData` is a union that contains a field that you set to point to the `TXNCarbonEventInfo` structure.
3. Assigns a pointer to the `TXNCarbonEventInfo` structure to the `iControlData` variable. You filled this structure in the section "Instantiating the Carbon Events Structure" (page 57).
4. Passes `MyTextObject` which is a `TXNObject` created by your application. You call the function `TXNNewObject` to create a `TXNObject`.
5. Sets this parameter to `false` to indicate whether or not MLTE should clear all tags associated with the `TXNObject`. Make sure you pass `false`. If you set this to `true`, all formatting and privileges attributes are reset to their default value.
6. Specifies the count of the number of items in the `iControlTags` array.
7. Calls the Core Foundation Base Services function `CFRelease` to release the memory associated with the dictionary you built.

As mentioned in "Customizing MLTE Support for Carbon Events" (page 54), MLTE supports Carbon events on a per object basis rather than on a per application basis. In other words, every time you allocate a new `TXNObject` with a call to the function `TXNNewObject`, you also need to call the function `TXNSetTXNObjectControls` to have MLTE support client-specified Carbon events for that text object.

Turning Off MLTE Support for Carbon Events

There is no need to turn off MLTE's support for Carbon events unless you don't want to use Carbon events or you want to handle all Carbon events in your application. Listing 3-20 shows the code you can use should you encounter such a rare situation. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-20 Turning off MLTE support for Carbon event handling

```
carbonEventInfo.useCarbonEvents = false;                                // 1
carbonEventInfo.filler = 0;
carbonEventInfo.flags = 0;                                              // 2
carbonEventInfo.fDictionary = NULL;

iControlData[0].uValue = (UInt32) &carbonEventInfo;
status = TXNSetTXNObjectControls(
                                    MyTextObject,
                                    false,
                                    1,
                                    iControlTags,
                                    iControlData
                                    );                                  // 3
```

Here's what the code does:

1. Sets `useCarbonEvents` to `false` to indicate to MLTE that Carbon events should be turned off for the `TXNObject` you specify.

2. Sets the flags value to `0`, which is only value you should currently provide.

3. Calls the function `TXNSetTXNObjectControls` to apply the control values to the specified text object.

Writing an Action Key Mapping Callback Function

You need to write an action key mapping callback function if you want to customize the Redo and Undo menu items with the specific action that can be redone or undone. For example, if the user just typed some text, you can change the Redo menu item to Redo Typing.

You provide your callback function to MLTE by passing a universal procedure pointer to the callback as an entry in the `TXNCarbonEventInfo` data structure. See "Customizing MLTE Support for Carbon Events" (page 54) for details.

The function shown in Listing 3-21 (page 60) maps `TXNActionKey` to the localized strings you want displayed to the user. The function takes two parameters: a `TXNActionKey` and a command ID. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-21 A function that maps action keys to localized strings

```
static CFStringRef MyActionKeyMappingProc (TXNActionKey theActionKey,
                                            UInt32   theCommandID )
{
    CFStringRef  theActionString = CFSTR("");                              // 1

    switch (theActionKey)                                                  // 2
    {
        case kTXNTypingAction                                              // 3
            if (theCommandID == kHICommandUndo)
                theActionString = CFCopyLocalizedString(
                    CFSTR("Undo Typing"), "Localized undo typing string.");
            else
                theActionString = CFCopyLocalizedString(
                    CFSTR("Redo Typing"), "Localized redo typing string.");
            break;
        case kTXNCutAction:                                                // 4
            if (theCommandID == kHICommandUndo)
                theActionString = CFCopyLocalizedString(
                    CFSTR("Undo Cut"), "Localized undo cut string.");
            else
                theActionString = CFCopyLocalizedString(
                    CFSTR("Redo Cut"), "Localized redo cut string.");
            break;
        case kTXNPasteAction:                                              // 5
                theActionString = CFCopyLocalizedString(
                    CFSTR("Undo Paste"), "Localized undo paste string.");
            else
                theActionString = CFCopyLocalizedString(
                    CFSTR("Redo Paste"), "Localized redo paste string.");
            break;
        default:                                                           // 6
            if (theCommandID == kHICommandUndo)
                theActionString = CFCopyLocalizedString(
                    CFSTR("Undo"), "Localized plain undo string.");
            else
                theActionString = CFCopyLocalizedString(
                    CFSTR("Redo"), "Localized plain redo string.");
            break;
    }
    return theActionString;                                                // 7
}
```

Here's what the code does:

1. Declares a `CFStringRef` and set its value to an empty string. A Core Foundation string (`CFString`) represents an array of Unicode characters (`UniChar`) along with a count of the number of characters. You'll use `CFString` objects a lot in Carbon because Unicode-based strings in Core Foundation provide a solid foundation for internationalizing software and they are stored more efficiently than arrays of Unicode characters.

2. Checks for an action key. The switch statement in this sample function has three cases, plus a default case. When you write your own function, you can have as many cases as there are `TXNActionKey` values, plus a default case. See *Inside Mac OS X: MLTE Reference* for a complete list of action types.

3. Checks for a typing action, then checks the command ID to see if the action is an undoable one. If the action is undoable, the code sets the action string to "Undo Typing." If the action is redoable, the code sets the action string to "Redo Typing." You must call the Core Foundation String Services macro `CFCopyLocalizedString` to get the string (such as "Undo Typing") you want displayed in the Edit menu. `CFCopyLocalizedString` takes two parameters, a `CFString` that is the key for the string you want to retrieve, and a comment to provide localizers with contextual information necessary for translation. The key is usually in the development language. `CFCopyLocalizedString` searches the default strings file (Localizable.strings) for the localized string associated with the specified key. For information on creating Localizable.strings files, see *Inside Mac OS X: System Overview*.

4. Checks for a cut action, then checks the command ID to see if the action is an undoable one. If the action is undoable, the code sets the action string to "Undo Cut." If the action is redoable, the code sets the action string to "Redo Cut."

5. Checks for a paste action, then checks the command ID to see if the action is an undoable one. If the action is undoable, the code sets the action string to "Undo Paste." If the action is redoable, the code sets the action string to "Redo Paste."

6. Returns the localized version of the plain Undo or Redo string.

7. Returns the string associated with the action. The caller of the function `MyActionKeyMappingProc` must release the string by calling the Core Foundation Base Services function `CFRelease`.

Migrating an Application from TextEdit to MLTE

If you have an application that uses TextEdit, and you want to modify the application so it uses MLTE instead of TextEdit, you should get the Multilingual Text Engine Software Developer's Kit (SDK). The MLTE SDK is available from the Apple Developer Connection website.

The TEtoMLTESample folder in the MLTE SDK contains a project file named TEtoMLTE.proj. When you open that project, you will find the folders TESources and MLTE Sources. You should compare the C++ code in one folder with the comparable code in the other folder. For example, compare TESample.cp with MLTESample.cp. The comments in the file that uses MLTE describe the changes necessary to use MLTE instead of TextEdit.

Document Revision History

This table describes the changes to *Handling Unicode Text Editing With MLTE*.

Date	Notes
2008-10-15	Removed statement "When word wrap is turned off, MLTE uses left alignment for all text."
2006-09-05	Made minor content correction.
	Corrected typographical errors. Revised the fifth bullet point in "Typing and Inline Input."
2005-07-07	Fixed typographical errors.
2003-04-01	Updated code to use Carbon events; removed references to unsupported functions.
	Added information, where appropriate, on the Fonts window interface.
	Includes minor bug fixes, updated formatting, and information on features added for Mac OS X version 10.2
	This revision incorporates the information formerly published in *Inside Carbon: Setting Up MLTE to Use Carbon Events*.
2002-07-02	Released for Mac OS X version 10.1. This document supplements the MLTE documentation on the Carbon Developer Documentation website.

MLTE Glossary

active end The point at which the user releases the mouse button when selecting a range of text or other items. Compareanchor point (page 65).

alignment The horizontal placement of lines of text with respect to the left and right edges of the text area. Alignment can be left, right, centered, or justified (flush on both left and right edges.)

anchor point The point at which the user presses the mouse button to begin selecting a range of text or other items by dragging through them. The anchor point is at one corner of the range of objects. Compare active end (page 65).

application font The default font for use by applications. The application font is defined by each script system.

ATSUI (Apple Type Services for Unicode Imaging) A technology that enables the rendering of Unicode-encoded text with advanced typographic features. ATSUI automatically handles many of the complexities inherent in text layout, including how to correctly render text in bidirectional and vertical script systems.

ATSUI style mask A byte-length mask with one bit set for each ATSUI-supported style to be applied.

auto-key event An event indicating the user has held a key down for a certain amount of time.

background The part of a glyph bitmap that surrounds the pixels that constitute the glyph itself.

base line An imaginary horizontal line that coincides with the bottom of each character in a font, excluding descenders (tails on letters such as p).

bidirectional script system A script system in which text is generally right-aligned with most characters written from right to left, but with some left-to-right text as well. Arabic and Hebrew are bidirectional script systems.

bidirectional text The combination of text with both left-to-right and right-to-left directions within a single line of text.

bottomline input A type of input method in which the user enters text in a small window, called a floating input window, that appears near the bottom of the screen. Compare inline input (page 67).

byte offset The indexed position of a 2-byte Unicode character in a text buffer, starting at zero for the first character. Sequential values for character offset correspond to the storage order of the characters.

caret A vertical or slanted blinking bar, appearing at the caret position in the display text, that marks the point at which text is to be insert or deleted. See also split caret (page 68).

caret position A location onscreen, typically between glyphs, that relates directly to the offset (in memory) of the current text insertion point in the source text. At the boundary between a right-to-left and left-to-right direction run on a line, one character offset may correspond to two caret positions, and one caret position may correspond to two offsets.

character An atomic unit of content for text data. A character is an abstract entity without any particular appearance; characters include letters, digits, punctuation, and symbols. See alsocharacter code (page 65) ; glyph (page 67).

character code In MLTE and ATSUI, a 16-bit value representing a Unicode text character. Text is stored in memory as character codes. Each script system's keyboard-layout ('KCHR') resource converts the virtual key codes generated by the keyboard or numeric keypad into character codes; each script system's fonts convert the character codes into glyphs for display or printing.

character encoding A conversion table for interpreting a specific character set. See also text encoding (page 69).

character rendering The process of preparing characters for display, taking into account line direction, contextual rules, and character reordering. For example, the formation of ligatures and diphthongs occurs during the display of text.

CFString An object that represents an array of Unicode characters (`UniChar`) along with a count of the number of characters. Unicode-based strings in Core Foundation provide a solid basis for internationalizing the software you develop. Unicode makes it possible to develop and localize a single version of an application for users who speak most of the world's written languages, including Russian (Cyrillic), Arabic, Chinese, and Japanese. Although conceptually CFString objects store strings as arrays of Unicode characters, in practice they often store them more efficiently. The memory a CFString object requires is typically about the same or even less than that required by a simple `UniChar` array.

continuous style In MLTE, a style value that is constant over an entire selection range.

cross-stream shift Refers to a shift in a character's position in the direction against the reading direction (that is, vertical for horizontal text and horizontal for vertical text).

data run See run (page 68); style run (page 68); text run (page 69).

destination rectangle The rectangle defining the area in which text is drawn.

diphthong A complex vowel sound that can be phonetically represented by 2 characters. The characters represent the initial and final sounds of the diphthong.

direction boundary A point between offsets in memory or glyphs on a display, at which the direction of the stored or displayed text changes.

direction run A contiguous (in memory) sequence of characters having the same right-to-left or left-to-right line direction.

discontinuous highlighting A highlighting effect that can occur when a selection range crosses one or more direction boundaries.

display order The order in which glyphs are drawn on a screen. Glyphs are always drawn in left-to-right order. Because not all text is read left-to-right, the display order of glyphs may be different from the storage order of their corresponding character codes in memory.

embedded objects Graphics, sound, or movie data that is in a text object along with text data.

encoding See text encoding (page 69).

floating input window A window used for text entry by an input method. See also floating window (page 66).

floating window A window that is similar to a standard Window Manager window except that is occupies a special layer so that it always remains in front of any application windows.

font A collection of glyphs that usually have some element of design consistency such as the shapes of the counters, the design of the stem, the stroke thickness, or the use of serifs.

font attributes A group of flags that modify the behavior or identity of a font.

font description A table that contains data that fully describes a font.

font family A group of fonts that share certain characteristics and a common family name.

font feature The set of typographic and layout characteristics that create a specific appearance for a glyph.

font run A contiguous (in memory) sequence of characters having the same font.

font variation An algorithmic way to produce a range of typestyles along a particular variation axis.

frame The viewable area of a text object; the view rectangle. Compare destination rectangle (page 66).

glyph The distinct visual representation of a character in a form that a screen or printer can display. A glyph may represent one character (the lowercase a), more than one character (the fi ligature), part of a character (the dot over an i), or a nonprinting character (the space character). See also character (page 65).

imaging system The system used to render text or graphics.

inline input An input method that allows the user to enter text directly into a document. In inline input, entry and conversion of characters take place at the current line position—where the converted text is intended to appear—rather than in a separate window. Inline input is the principal example of the kind of text service supported by the Text Services Manager. Compare bottomline input (page 65).

input method A software module for 2-byte script systems that converts phonetic or syllabic characters, entered from a keyboard, into ideographic or other complex representation of text. Because 2-byte script systems have too many characters to be entered directly from a keyboard, the input method uses a conversion technique, such as translating sequences of phonetic characters that are typed into a special input window. For example, the Japanese script system provides software for transcribing Kana (phonetic Japanese) into ideographic Kanji.

insertion point The point in the source text at which text is to be inserted or deleted. An insertion point is specified by a single caret position. Compare caret (page 65).

justification A type of alignment that involves the spreading or compressing of printed text to fit into a given line width so that it is flush on both left and right edges of the text area.

keyboard and font synchronization A process by which the current keyboard script is compared to the script of the font at the current insertion point. If the two don't match, one or the other is changed so the two scripts are the same. In most cases, when the user starts typing, the font is automatically replaced with one belonging to the keyboard script, although it is possible to synchronize in the other direction.

key-down event An event indicating the user pressed a key.

key-up event An event indicating the user released a key.

keyboard script The script system for keyboard input. It determines the character input method and the mapping of keystrokes to character codes. The keyboard script may be different from the script used to display text.

ligature A glyph that is created when two or more characters are combined to create a new character.

line direction The direction in which text in a particular language is written and read. The English language has a left-to-right line direction; Arabic and Hebrew have a (primarily) right-to-left line direction.

localize See localization (page 67).

localization The adaptation of system software or applications to a particular language or region. Localization involves translating strings and providing proper conventions for sorting, date and time formats, currency and measurement units, calendars, numbers, and other culturally specific items such as icons.

missing character glyph The glyph in a font that is drawn when no glyph is defined for a character code in a font.

modifier key A key that when pressed at the same time as another key, modifies the behavior of the other key.

pass-through mode A mode that does not modify data. With respect to keyboard-entry, pass-through mode allows users to enter ASCII characters in the context of a 2-byte script, without changing the keyboard script.

primary line direction The dominant line direction (right-to-left or left-to-right) of the current text. The primary line direction is typically specified by the value of the global system direction variable. See also line direction (page 67).

private MLTE scrap Scrap used exclusively by MLTE.

Roman character set A set of characters used for the Roman writing system. Roman character sets include the Standard Roman character set and the ASCII character set.

Roman keyboard script A keyboard script that uses the Roman character set.

run A sequence of glyphs that are contiguous in memory and share a set of common attributes. See also font run (page 66); style run (page 68).

script A method for depicting words visually. Some examples of scripts are Latin, Greek, Hiragana, Katakana, and Han.

script system A collection of software utilities that provides for the representation of a specific writing system. It consists of a set of keyboard resources, a set of international resources, and one or more fonts. Script systems include Roman, Japanese, Arabic, traditional Chinese, Simplified Chinese, Hebrew, Greek, Thai, and Korean.

selection range The contiguous sequence of characters in the source text that mark where the next editing operation is to occur. The glyphs corresponding to those characters are commonly highlighted onscreen.

Shift JIS (Japanese Industrial Standard) A character encoding based on two JIS standards: JIS X 0201 and JIS X 0208. Shift JIS consists of codes from the JIS X 0208 standard that are shifted to make room for older Hankakukana codes from the JIS X 0201 standard.

single caret In unidirectional text, the standard text-insertion caret. In mixed-directional text, one caret that appears at the place where the user will insert the next character, given the current keyboard script. At a boundary between two direction runs, the single caret can correspond to either the primary line direction or the secondary line direction. Because changing the keyboard script in that situation changes the caret location, the single caret is also called a moving caret or jumping caret.

split caret A type of caret that, at the boundary between text of opposite directions, divides into two parts: a high caret and a low caret, each measuring half the line's height. The two separate half-carets merge into one in unidirectional text. Compare single caret (page 68).

Standard Roman character set The 256 characters and character codes that are supplied with the Macintosh Roman script system. The Standard Roman character set consists of the Macintosh character set plus additional defined characters with character codes between $D9 and $FF.

storage order The order in which character codes are stored in memory. Storage order may be different from display order.

style A visual attribute, other than size, applied as a systematic variation to the plain (unstyled) characteristics of a font glyph. For example bold, italic, underline, outline, shadow, condense, and extend.

style run A sequence of text that is contiguous in memory and in which all the characters are in the same style. Compare text run (page 69).

synchronization See keyboard and font synchronization (page 67).

system direction The horizontal placement of interface elements, including the default line direction (left-to-right or right-to-left) for text in the system script. System direction is specified by the global system direction variable.

system font The font used to display text in menus, dialog boxes, alert boxes, and so forth in a given script system. For example, in the Roman script system, the system font is Chicago on Mac OS 9 and earlier versions.

system script The primary script system used by the operating system, such as in dialogs and menu bars. The system script affects system defaults, such as the system font, line direction, and text-formatting rules. All other scripts are secondary to the system script.

text A set of specific symbols that, when displayed in a meaningful order, conveys information.

text area The space on the display device within which the text should fit.

text direction The direction in which reading proceeds. Roman text has a left-to-right direction; Hebrew and Arabic have a (predominantly) right-to-left direction; Chinese and Japanese can have a vertical direction.

text encoding The coded character set or character encoding scheme used to represent a particular piece of text.

Text Encoding Conversion (TEC) Manager A pair of shared library extensions—namely, the Text Encoding Converter and the Unicode Utilities—that facilitate text encoding conversion on Mac OS–based computers.

Text Encoding Converter A shared library extension that provides the services for general and algorithmic encoding conversions or multi-encoding streams. The Text Encoding Converter sometimes uses Unicode Utilities.

text manipulation System-level procedures used to order and compare characters, determine line breaks, determine text directionality, and keep track of character properties, such as case.

text object An opaque structure that the Multilingual Text Engine uses to handle text formatting at a document level.

text run A sequence of text that is contiguous in memory and in which all characters are in the same font. Compare style run (page 68).

text segment For text layout, the portion of a style run that falls on a single text line. (It may be the entire style run.) Most text measuring and drawing routines work on a single text segment at a time.

Text Services Manager (TSM) The Mac OS technology that provides text services such as input methods. TSM handles communication between client applications that request text services and the software modules, known as text service components, that provide them.

tick 1/60 second.

Unicode Unicode is an ISO standard for 16-bit universal worldwide character encoding developed by a consortium that includes Apple. Unicode has enough capacity to handle unique encodings for all characters available in all scripts, including the 2-byte script systems such as Chinese, Japanese, and Korean.

Unicode Utilities A shared library extension that provides table-based conversion between Unicode and other encodings.

unidirectional text A sequence of text that has a single line direction. Compare bidirectional text.

UTF-16 (Unicode Transformation Format) A form of Unicode in which 16-bits are used to encode a character.

variation axis A range of values used to produce different type styles for a font. For example, a font that has a weighting axis could be displayed with weights that range from 0.7 point (light) to 1.3 points (bold). It is possible to combine variations. For example, font width variations can be combined with weighting variations to produce font variations ranging from light, narrow to bold, wide.

view rectangle In MLTE, the rectangle defining the portion of the window within which text is actually displayed. Text drawn in the destination rectangle is made visible to the application user in the view rectangle.

WorldScript A group of Mac OS managers, extensions, and resources that facilitate multilingual text processing.

writing system A set of characters and the basic rules for their use in creating a visual depiction of language. Writing systems may differ in the direction in which their characters and lines run, the size of the character set used, and the context sensitivity of character selection. Writing systems include Roman, Japanese, Arabic, and Hebrew. Compare script system (page 68).