TextEdit

TextEdit is a collection of routines and data structures that give your application basic text formatting and editing capabilities, including text display in multiple scripts. TextEdit manages fundamental text processing tasks on text limited to 32 KB. You can use the TextEdit routines in many kinds of applications, such as spreadsheets, online (data-entry) forms, online advertising programs, simple programming-language or text-file text editors, electronic mail programs, drawing and painting programs with simple text-editing features, and electronic note cards. However, TextEdit was not designed to be used to implement word-processing applications with complex support that manipulate lengthy documents.

To use TextEdit and the information provided in this chapter, you should be familiar with the basic concepts and structures behind QuickDraw and how it handles text—particularly points, rectangles, graphics ports, fonts, and character style—the Event Manager, the Window Manager—particularly update and activate events—the Font Manager, the Script Manager, and Text Utilities.

For information on non-text features of QuickDraw, see *Inside Macintosh: Imaging*. For information on the Event Manager and the Window Manager, see *Inside Macintosh: Macintosh Toolbox Essentials*.

This book includes chapters that cover the Font Manager, Text Utilities, the Script Manager, and QuickDraw Text. Although these chapters pertain to TextEdit, the only chapter in this book that you need to read as a prerequisite to TextEdit is "Introduction to Text on the Macintosh."

This chapter describes how to use TextEdit to perform a range of editing and formatting capabilities including

- inserting new text
- selecting and highlighting ranges of text
- deleting selected text and possibly inserting it elsewhere, or copying text without deleting it
- replacing selected text
- translating mouse activity into text selection
- scrolling text within a window, including automatically scrolling text that is not visible but is affected by the editing activity
- changing the characteristics of text, including font family, style, and size
- customizing some TextEdit behavior

About TextEdit

TextEdit was originally designed to handle editable text items in dialog boxes and other parts of the system software. Although TextEdit has been enhanced to provide more text-handling support since its inception, especially in its handling of multi-script text, it retains some of its original limitations. TextEdit was not originally intended to manipulate lengthy documents or text requiring more than rudimentary formatting. For example, TextEdit does not handle tabs. (Your application can provide support for tabs to supplement TextEdit.)

However, TextEdit handles some of the cumbersome tasks that a text processor needs to perform, and provides you with an alternative to writing your own text processor. For example, when you use TextEdit routines to edit text, your application does not need to allocate memory for blocks of text that change dynamically during the editing session because TextEdit takes care of this for you. When the user selects a range of displayed text of a TextEdit edit record, TextEdit recognizes this and responds by highlighting the text.

TextEdit relies on the Script Manager, QuickDraw, and Text Utilities to handle text correctly, and eliminates the need for your application to call these routines directly. Because TextEdit supports text from more than one script system and manages scripts having different primary line directions, you can use its routines and features to develop applications that support multiple languages.

TextEdit uses Text Utilities routines: the `FindWordBreaks` procedure for determining word breaks and the `StyledLineBreak` function for determining line breaks. TextEdit also allows you to customize how word boundaries and line breaks are defined.

TextEdit and Standard Macintosh Features

Because TextEdit routines follow the Macintosh user interface guidelines, using them ensures the presentation of a consistent user interface in your application. Your application can rely on TextEdit to support these standard features instead of having to implement them directly:

- selecting text by clicking and dragging with the mouse
- double-clicking to select words, which are defined according to the rules of the script system in which they are written
- line breaking, which prevents a word from being split inappropriately between lines when text is drawn

- extending or shortening a selection range by Shift-clicking
- highlighting of the current text selection, or display of a blinking vertical bar at an insertion point
- cutting, copying, and pasting within and between applications
- the use of more than one font, size, color, and stylistic variation from character to character within a single block of text
- display of text in more than one language on a single line

Multistyled and Monostyled Text

Text is rendered in a certain font, style, size, and color. These aspects of text are collectively referred to as **character attributes.** TextEdit supports the display of text in various character attributes (different fonts, styles, sizes, and colors) within the context of a single edit record.

Text that uses a variety of fonts, styles, sizes, or colors is referred to in this chapter as *multistyled text* to distinguish it from text that uses a single font, style, size, and color, which is referred to as *monostyled text*.

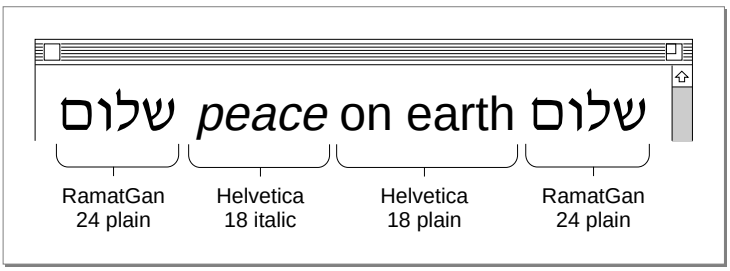
TextEdit lets you boldface, italicize, underline, outline, condense, extend, and shadow text. Using TextEdit routines, you can change the font family and type size of the entire text of an edit record (or a selected range of text that the user has chosen or the application has set). You can even increase the type size incrementally across a range of text containing various sizes, for example, so that all 10 point text is changed to 12 point and all 12 point text is changed to 14 point. If your application uses multistyled TextEdit and allows users to select fonts, TextEdit displays text correctly in all scripts. Apart from the TextEdit routines that deal with multistyled text exclusively, you can use all of the TextEdit routines to simplify and manage your application's text editing tasks for both multistyled and monostyled text.

Note
In the original *Inside Macintosh* documentation that describes TextEdit, the term *face* is used to refer to the following text style attributes: bold, italic, underline, outline, condense, extend, and shadow. The term *style* is now used instead of face to refer to these attributes. ◆

TextEdit organizes multistyled text into style runs. The characters comprising a **style run** are contiguous in memory and are all displayed in the same font, size, color, and script as well as style. TextEdit tracks style runs in the data structures that are allocated for a multistyled edit record and uses this information to correctly display multistyled text.

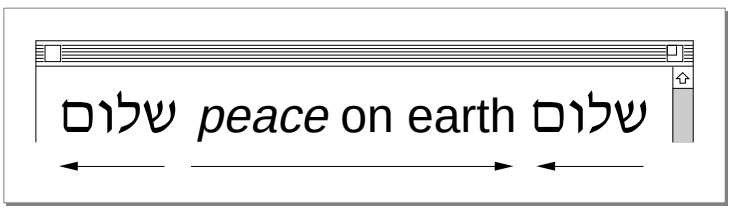
Figure 2-1 shows four style runs in a line of text.

Figure 2-1 Style runs in a line of text



TextEdit supports **mixed-directional text:** the combination of scripts with left-to-right and right-to-left directional text within a single line. Figure 2-2 shows an example of Hebrew and Roman text on the same line. The two runs of Hebrew text have a right-to-left direction, and the Roman text direction is left to right.

Figure 2-2 Mixed-directional text display



Font and Keyboard Script Synchronization

TextEdit handles synchronization of the **font script,** the script system that corresponds to the font of the current graphics port, and the **keyboard script,** the script system used for keyboard input, for multistyled and monostyled text.

For monostyled text, the primary script system determines whether or not TextEdit synchronizes the font script and the keyboard script, based on the value of a flag in the script system's international bundle resource (`'itlb'`). TextEdit uses this flag, without requiring any action on the part of your application.

For multistyled text, TextEdit always synchronizes the font script and the keyboard script. (If the font script at the selection range or insertion point is the same as the keyboard script, then this font is used.) The following sections explain the conditions that determine whether TextEdit matches the keyboard script to the font script or vice versa. TextEdit synchronizes *the keyboard script with the font script* under the following conditions:

- When your application calls a TextEdit routine to change the font of a text selection or to process a mouse-down event in text as either an insertion point or a selection. This means, for example, that if a user types Arabic text followed by Roman text and clicks in the Arabic text, the keyboard adjusts and changes to Arabic without the user's needing to change the keyboard manually. Similarly, if a user clicks in the Roman text, the keyboard changes to Roman without the user's altering the keyboard.
- If the selection range encompasses text—if it is not an insertion point—then TextEdit uses the font corresponding to the first character of the selected text to determine the keyboard script. When an insertion point falls on a script boundary, the keyboard is synchronized to the font of the character preceding the boundary (in storage order). (A **selection range** is a series of characters, selected by the user or the application, where the next editing operation is to occur. Although the character representations are contiguous in memory, they can be discontinuous on the display screen when the text is bidirectional. For more information, see "The Selection Range, the Insertion Point, and Highlighting in TextEdit" on page 2-10.)

TextEdit synchronizes *the font script with the keyboard script* under the following condition:

- When your application calls a TextEdit routine to input a character and if the keyboard script is different from the font script at the selection range (or insertion point). If a font was selected and never used, thus remaining in the scrap that TextEdit uses for character attributes (null scrap) and if the font script coincides with the keyboard script, then this font is used. Otherwise, TextEdit searches through the preceding fonts in the style run table until it locates a font that corresponds to the keyboard. If one does not exist, then it uses the application font. For more information about the null scrap, see "The TextEdit Private, Null, and Style Scraps" on page 2-15.

Cutting, Copying, and Pasting Text

TextEdit provides routines that let you cut, copy, and paste text

- within a single edit record
- between edit records within an application
- between an application and a desk accessory
- across applications

You use the same routines to cut and copy monostyled and multistyled text. There are, however, separate routines for pasting monostyled and multistyled text. For multistyled text, the TextEdit routines preserve any stylistic variation along with the cut or copied text in order to restore it when you paste the text.

The TextEdit User Interface

This section describes the TextEdit user interface, that is, how TextEdit displays text on the screen and the methods it uses to communicate information about that text to an application user. It explains some of the processes that TextEdit performs automatically for your application, including how TextEdit uses highlighting or a caret to identify where the next editing operation is to occur, how TextEdit handles line measurement for your application, and how TextEdit uses buffering to handle 2-byte characters.
This section also covers some aspects of the user interface that your application can control through TextEdit routines, such as the kind of text alignment and the use of buffering to enhance performance.

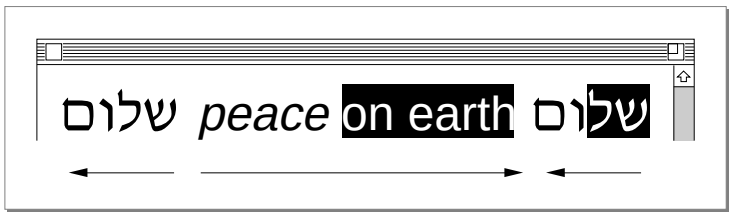
The Selection Range, the Insertion Point, and Highlighting in TextEdit

Depending on the purpose of an application, a user might select a range of text to be edited or the application might set the selection range. In either case, the selected text becomes the current selection range. TextEdit uses a byte offset to identify the position of a character in the text buffer of an edit record, and an edit record includes fields that specify the byte offsets of the characters in the text buffer that correspond to the beginning and the end of the current selection range in the displayed text. (See "An Overview of the Edit Record" on page 2-16 for more about edit records.)

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.** TextEdit marks an insertion point with a blinking **caret** in the form of a vertical bar (|).

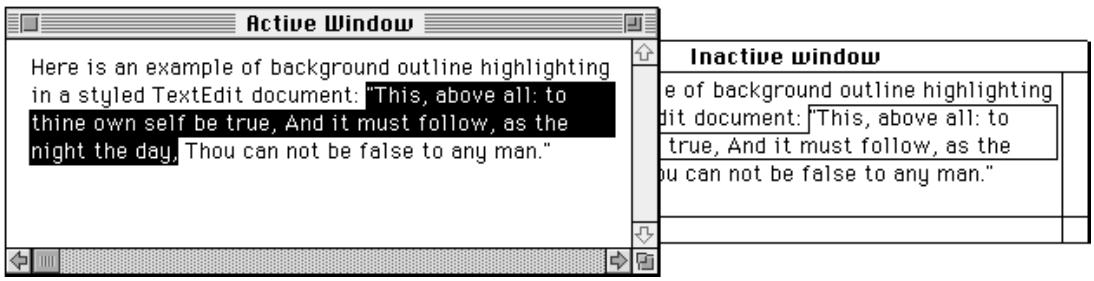
TextEdit uses highlighting to display a selection range. Because TextEdit supports mixed-directional text, the selection range can appear as discontinuous text. Displayed text is highlighted according to the storage order of the characters. When multiple script systems having different line directions are installed, a continuous sequence of characters in memory may appear as a discontinuous selection when displayed.
Figure 2-3 shows how TextEdit highlights a range of text whose displayed glyphs are not contiguous, although their corresponding byte offsets are contiguous in memory. In this example, the primary line direction is left to right.

Figure 2-3 Discontinuous highlighting display



TextEdit provides a function that lets you to turn **outline highlighting,** the framing of text in a selection range, in an inactive window, on or off. See Figure 2-4. (For more information about outline highlighting, see "TEFeatureFlag" on page 2-109.)

Figure 2-4 Outline highlighted text selection in background window



Caret Position and Movement

This section describes how TextEdit displays and moves a caret. For more information, see the discussion of caret handling in the chapter "Introduction to Text on the Macintosh" in this book.

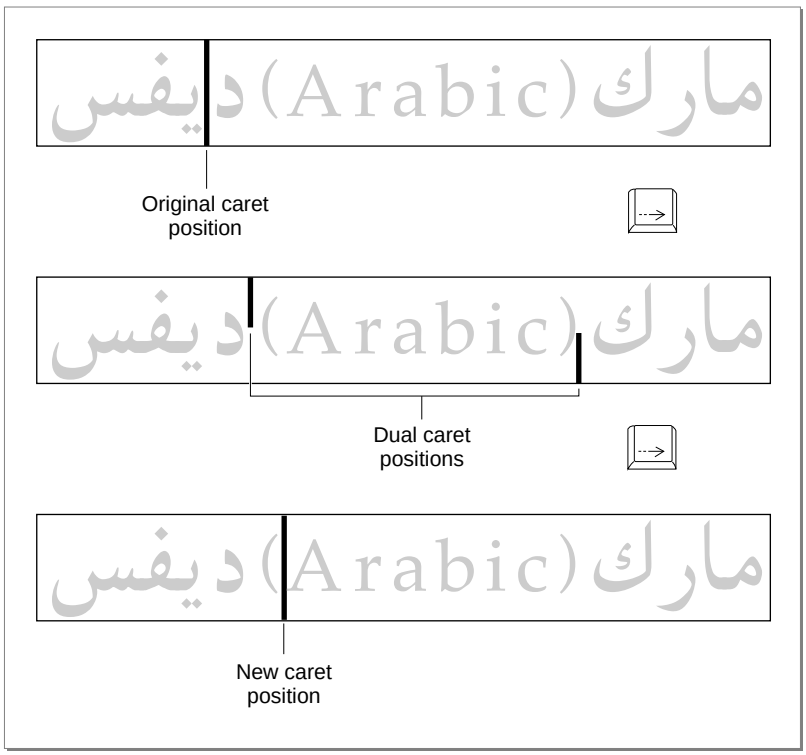
TextEdit marks the position in the displayed text where the next editing operation is to occur with a caret. When TextEdit pastes text into a record, it positions a caret after the newly pasted text on the screen. TextEdit uses a single caret for text that does not include mixed directions. When TextEdit displays a single caret in unidirectional text and the user presses an arrow key to move the caret left or right across the text, TextEdit moves the caret in the direction of the arrow key.

When the text includes mixed directions, TextEdit uses either a moving caret or a dual caret, depending on the value of a Script Manager flag. For example, if this flag specifies a moving caret, TextEdit displays the caret at the screen location where the next glyph is to appear, based on the text direction of the keyboard script.

If this flag specifies a dual caret, TextEdit displays a high caret and a low caret, each measuring half the line's height. The high caret is displayed at the screen location associated with the glyph that has the same direction as the primary line direction, and the low caret is displayed at the screen location associated with the glyph that has a different direction from the primary line direction.

When TextEdit displays a dual caret on a direction boundary, only the primary caret moves in the direction of the arrow. Figure 2-5 shows a sequence of two Right Arrow keypresses and their impact on caret display and movement in a line containing mixed-directional text. In this example, the primary line direction is right to left.

Figure 2-5 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 splits into a dual caret. When the user presses the Right Arrow key again, TextEdit displays a single, full caret after the parenthesis in the Roman text. Because the caret position is again in the middle of a style run, TextEdit no longer uses the dual caret.

Note
TextEdit currently deviates from this model for caret movement in monostyled left-to-right text (displayed in a non-Roman font) on any primary right-to-left script system. On the Arabic script system, for example, it is possible to display the low-ASCII Roman characters from an Arabic font. If a user presses the arrow keys to move through these characters, the caret moves in the opposite direction of the arrow. ◆

Vertical movement of the caret is less complex. When the user presses the Up Arrow key, the caret moves up by one line, even in lines of text containing fonts of different sizes. When the caret is positioned on the first line of an edit record, and the user presses the Up Arrow key, TextEdit moves the caret to the beginning of the text on that line, at primary caret position 0. (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 an edit record, and the user presses the Down Arrow key, TextEdit moves the caret to the end of the text on that line (that is, 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).

Note
TextEdit does not support the use of modifier keys, such as the Shift key or the Option key, in conjunction with the arrow keys. ◆

If spaces at the end of a text line extend beyond the view rectangle, TextEdit draws the caret at the edge of the view rectangle, not beyond it. Whether TextEdit displays a caret at the beginning or end of a line when a mouse-down event occurs at a line's end depends on the current caret position and the value in a field (`clikStuff`) of the edit record. TextEdit sets this field to reflect whether the most recent mouse-down event occurred on the leading or trailing edge of a glyph.

For example, if the mouse-down event occurs on the leading edge of a glyph, TextEdit displays the caret at the caret position corresponding to the leading edge of that glyph. If the mouse-down event is on the trailing edge of a glyph, TextEdit displays the caret at the beginning of the next line. For more information about determining a caret position, see the sections that discuss caret handling in the chapters "Introduction to Text on the Macintosh" and "QuickDraw Text" in this book.

Text Alignment

TextEdit 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 TextEdit supports accommodate script systems that are read from right to left, as well as those that are read from left to right. The types of alignment supported are

- default alignment (positions the text according to the line direction of the system script. It 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 rightward, or left-to-right, line direction. Arabic and Hebrew have a [primarily] leftward, or right-to-left, line direction.)
- center alignment (centers each line of text between the left and right edges of the destination rectangle)
- right alignment (positions the text along the right edge of the destination rectangle)
- left alignment (lines up the text with the left edge of the destination rectangle)

If your application requires justified alignment, you can use the QuickDraw routines that support full justification; TextEdit does not support justified alignment. See the chapter "QuickDraw Text" in this book for more information.

Line Measurement

TextEdit measures a line of text appropriately for all script systems by removing any trailing white space from the end of it, taking the line direction into account. It uses the QuickDraw `VisibleLength` function to exclude trailing white space, based on the script system, the text direction, and the primary line direction. For more information about the behavior of `VisibleLength` for various script systems, see the chapter "QuickDraw Text" in this book.

An anomaly exists, however, in the way TextEdit draws at the end of a line. When the primary line direction of a script system is right to left (for instance, on a Hebrew system), when the alignment is left or center, and when spaces are entered in a right-to-left font, TextEdit measures spaces at the end of the line and therefore may draw the text beyond the edge of the view rectangle. The caret, however, remains in view and is pinned to the left edge of the view rectangle.

This anomaly also exists when the primary line direction of a script system is left to right and the alignment is center. In this instance, TextEdit measures spaces at the end of the line, and as more spaces are added (and, therefore, measured), the visible text in the line is drawn out of view beyond the left edge of the view rectangle. The caret, however, remains in view and is pinned to the right edge of the view rectangle.

Text Buffering

TextEdit uses two methods of text buffering; one method, which is automatic, is used to handle 2-byte characters properly. The other method, which you can enable or disable, improves performance in relation to how TextEdit handles input of 2-byte characters.

For the first method, which is automatic, TextEdit relies on the Script Manager. The Script Manager handles 2-byte characters properly, and TextEdit takes advantage of this. If a 2-byte character, such as a Kanji character, is typed, TextEdit buffers the first byte until it processes the second byte, at which time it displays the character. The internal buffer that TextEdit uses for a 2-byte character is unique to each edit record. For example, TextEdit can buffer the first byte of a 2-byte character in a record, then the application can call the TextEdit `TEKey` procedure for another edit record. While `TEKey` processes the character for the second edit record, the first byte of the 2-byte character remains in the first edit record's buffer until TextEdit processes the second byte of that 2-byte character, and then displays the character.

The second method of text buffering enhances performance, and you can turn it on or off through the TextEdit function, `TEFeatureFlag`. In this case, TextEdit uses a global buffer—it differs from the `TEKey` procedure's internal 2-byte buffer—that is used across all active edit records. These records may be in a single application or in multiple applications. Because of this, you should exercise care when you enable the text-buffering capability in more than one active record; otherwise, the bytes that are buffered from one edit record may appear in another edit record.

- Ensure that buffering is not turned off in the middle of processing a 2-byte character. To guarantee the integrity of your record, it is important that you wait for an idle event before you disable buffering or enable buffering in a second edit record.

- When text buffering is enabled, ensure that `TEIdle` is called before any pause of more than a few ticks—for example, before `WaitNextEvent`. A possibility of a long delay before characters appear on the screen exists—especially in non-Roman systems. If you do not call `TEIdle`, the characters may end up in the edit record of another application.

If you enable text buffering for performance enhancement on a non-Roman script system and the keyboard has changed, TextEdit flushes the text of the current script from the buffer before buffering characters in the new script.

The TextEdit Private, Null, and Style Scraps

There are three scrap areas that TextEdit uses exclusively: the TextEdit private scrap, the TextEdit null scrap, and the TextEdit style scrap. The TextEdit routines use all of these scraps to hold transient information.

TextEdit uses the **private scrap** for all cut, copy, and paste activity whether the text is multistyled or monostyled. The private scrap belongs to the application. When the text is multistyled, TextEdit also copies the text to the Scrap Manager's desk scrap.

TextEdit uses the **null scrap** to store character attribute information associated with a null selection (an insertion point) or text that is deleted when the user backspaces over it. The null scrap belongs to the multistyled edit record. Character attribute information stored in the null scrap is retained until it is used, for example, when applied to newly inserted text, or until some other editing action renders it unnecessary, such as when TextEdit sets a new selection range. A number of routines that deal with multistyled text check the null scrap for character attribute information and, if there is any, apply it to newly inserted text when character attributes for that text are not available.

When you cut or copy multistyled text, memory is allocated dynamically for the **style scrap** and the character attribute information is copied to it. Your application can also use the style scrap for other purposes. For example, to save and restore multistyled text both the text and the associated character attribute information must be preserved; you can save character attributes associated with a range of text in the style scrap. Also, you can create a style scrap record and store character attribute information in it to be applied to inserted text. Your application can create as many style scraps as it needs. For more information, see the discussion of the style scrap record under "Data Structures" on page 2-65.

As part of TextEdit initialization, `TEInit` creates the private scrap and allocates a handle to it. TextEdit creates and initializes a null scrap for a multistyled edit record when an application calls `TEStyleNew` to create the edit record. (The null scrap remains throughout the life of the edit record: it is disposed of when the application calls `TEDispose` to destroy the edit record and release the memory allocated for it.) TextEdit allocates memory used for the style scrap dynamically when your application calls a routine that uses it.

Note
Because these scraps are in RAM, they are volatile, and a power failure can cause the data in a scrap to be lost. ◆

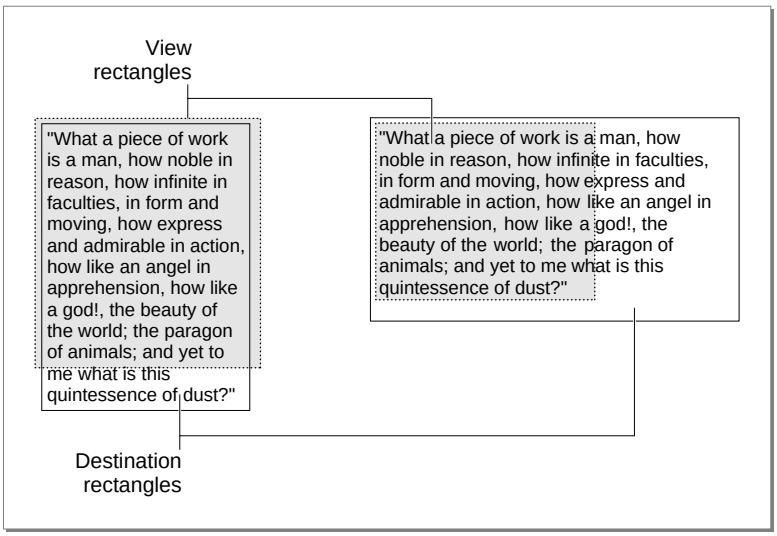
An Overview of the TextEdit Data Structures

To edit text on the screen, TextEdit maintains information about where the text is stored, where to display it, and the text style. This information is contained in a record that defines the complete editing environment. You can allocate a **monostyled edit record** to contain text that is set in a single font, size, and style, or you can allocate a **multistyled edit record** to contain text with attributes that can vary from character to character.

An Overview of the Edit Record

An edit record, which is the primary data structure that TextEdit uses, carries text storage, display, and editing information. When you allocate an edit record, you specify where the text is to be drawn and where it is to be made visible. The **destination rectangle** is the area in which the text is drawn, and the **view rectangle** is that portion of the window within which the text is actually displayed. (For a complete discussion of destination and view rectangles, see the QuickDraw chapters in *Inside Macintosh: Imaging*.) Figure 2-6 shows two sets of destination and view rectangles. The view rectangles 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 view rectangle.

Figure 2-6 Destination and view rectangles



The edit record includes fields that point to these rectangles. In addition to the two rectangles, the edit record also contains

- a handle to the text to be edited
- the current selection range that determines exactly which characters are to be affected by the next editing operation
- the alignment of the text, as left, right, or center
- for multistyled edit records, a handle to a subsidiary record, the style record, containing the character attributes used to portray the text. This style record, itself, contains subsidiary data structures.

Related Data Structures

Stemming from the main TextEdit edit record, relationships exist among the rest of the TextEdit data structures.

When TextEdit creates an edit record, the record contains a field that stores the handle to the dispatch record. The dispatch record is an internal data structure whose fields, referred to as hook fields or hooks, contain the addresses of routines that TextEdit uses internally, for example, to measure and draw text, or to determine a character's position on a line. These routines, called hook routines, determine the way TextEdit behaves. You can use a TextEdit customization routine to replace the address of a default hook routine with the address of your own customized routine. For example, you can provide a routine to be used for word selection that defines word boundaries more precisely for any script system.

When you allocate a monostyled edit record, the edit record, a handle to the text, and a single subsidiary internal data structure, the dispatch record, are created. However, when you allocate a multistyled edit record, a number of additional subsidiary data structures are created to support the text styling capabilities and the display of text in multiple languages.

For a multistyled edit record, the edit record contains a handle to the style record. The style record stores the character attribute information for the text, and contains a handle to the style table, which has one entry for each distinct set of character attributes. Each entry in the style table is a style element record. The style record also contains a style run table, which is an array that gives the start of each style run, and an index into the style table. The style run table array identifies the byte offset of the starting character to which the character attributes, stored in the style table, apply.

The style record contains two other handles: a handle to the line-height table and a handle to the null style record. The line-height table provides vertical spacing and line ascent information for the text to be edited with one element for each line of an edit record. A line number is a direct index into this array. The null style record consists of a reserved field and a handle to the style scrap record.

The style scrap record, which is part of the null scrap, stores character attribute information associated with a null selection to be applied to inserted text. It also holds character attribute information associated with a selected range of multistyled text when the character attributes are to be copied, or the text and its attributes are to be cut or copied.

Part of the style scrap record is the scrap style table which has a separate element for each style run in the style scrap record. The character attribute information for each of these elements is stored in a scrap style element record.

Several TextEdit routines use a text style record to pass character attribute information between the application and the routine.

Figure 2-7 shows the two data structures that TextEdit creates for monostyled text. Figure 2-8 shows the data structures that TextEdit creates for multistyled text and how they are related; these data structures consist of the two records that TextEdit also creates for monostyled text plus additional structures needed to store character attribute information. See Figure 2-15 on page 2-67 for a version of the data structures including fields.

Figure 2-7 Relationship between the TextEdit data structures for monostyled text

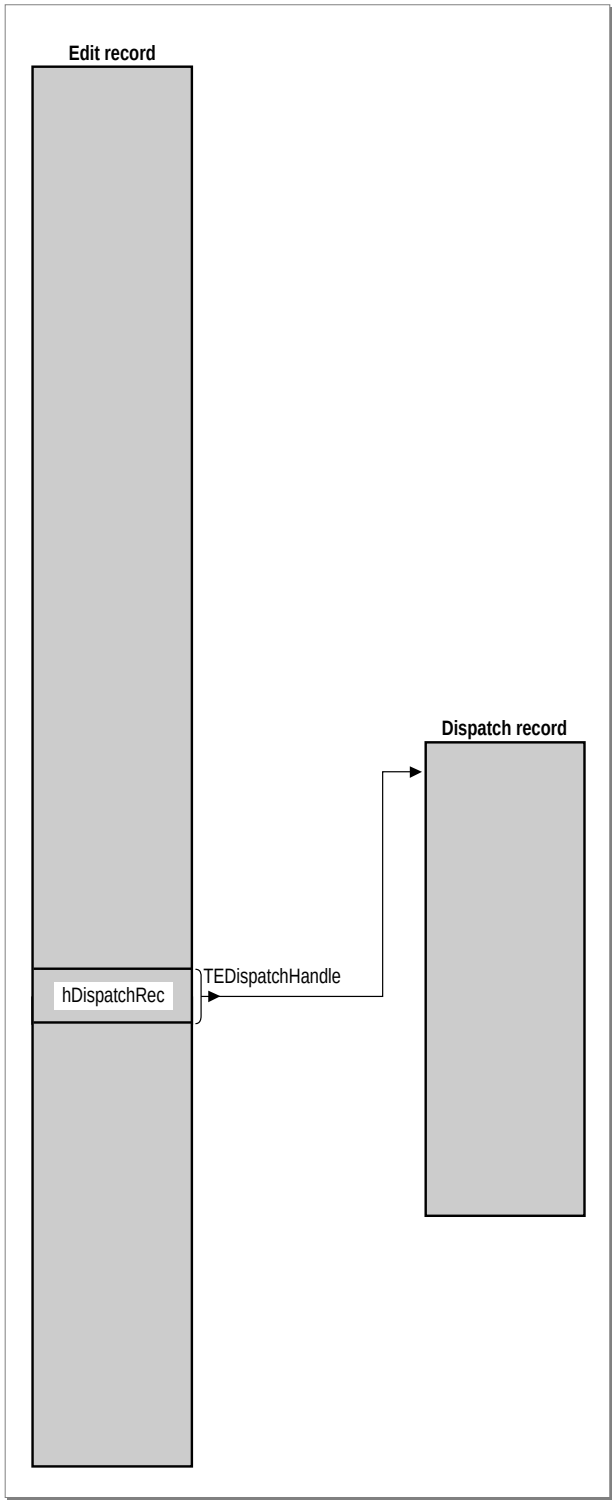
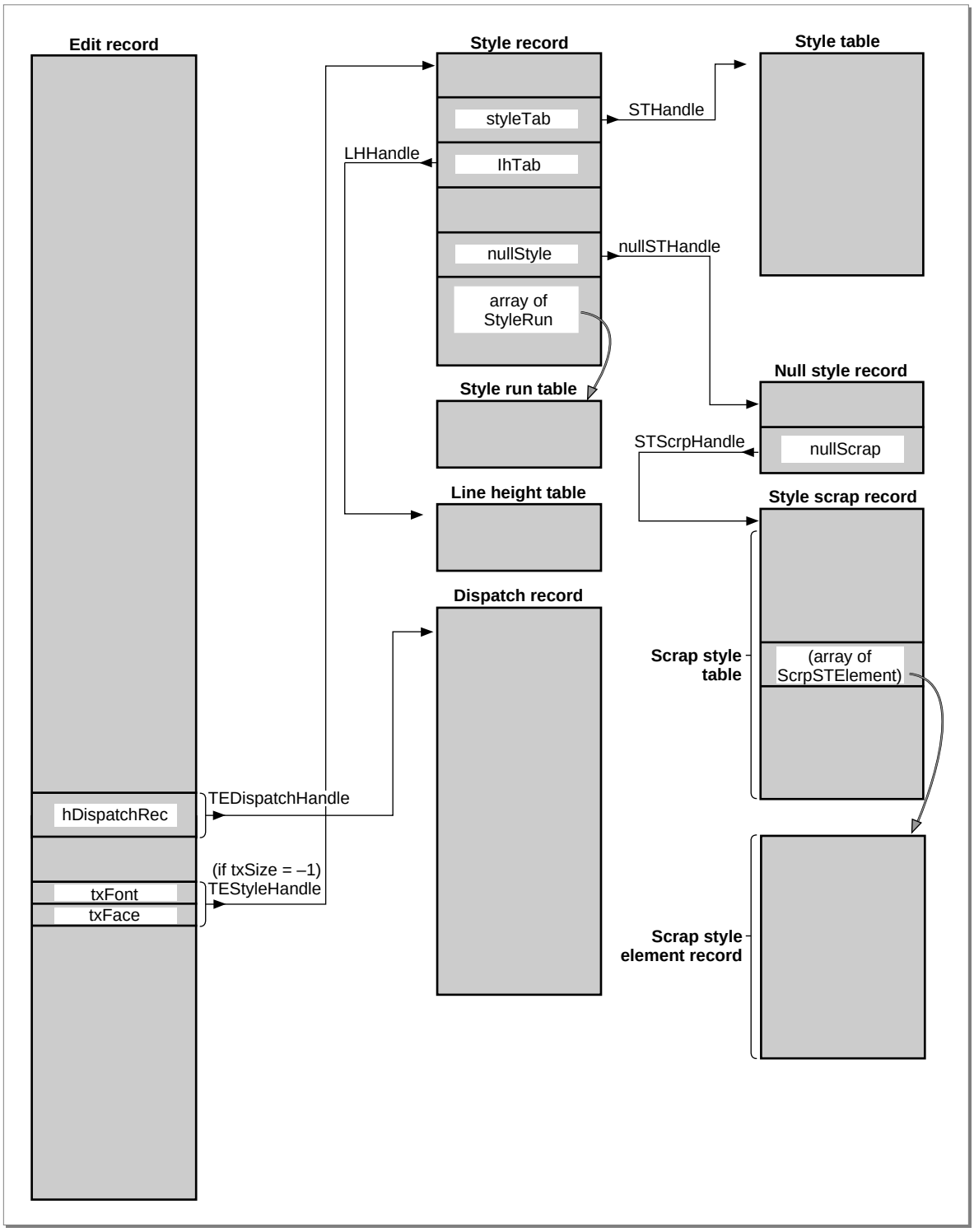


Figure 2-8 Relationships among the TextEdit data structures for multistyled text


Edit record
Style record
Style table
styleTab
STHandle
LHHandle
lhTab
nullStyle
nullSTHandle
array of StyleRun
Style run table
Null style record
nullScrap
STScrpHandle
Line height table
Style scrap record
Dispatch record
Scrap style table
(array of ScrpSTElement)
hDispatchRec
TEDispatchHandle
(if txSize = –1)
TEStyleHandle
txFont
txFace
Scrap style element record

TE32_Fig.8

Using TextEdit

This section describes how to initialize TextEdit and use the TextEdit routines and data structures to display text and implement editing features in an application. It also describes how to customize the behavior of TextEdit, for example, to better suit the requirements of your application and the script systems it supports.

- "Getting Started With TextEdit" describes how to display static text in a box, create an edit record for modifiable text, set the text of an edit record and scroll it, set its insertion point, and dispose of the edit record.
- "Responding to Events Using TextEdit" describes how to handle mouse-down, key-down, and idle events.
- "Moving Text In and Out of Edit Records" describes how to cut, copy, and paste text and its character attributes within or across applications, or between an application and a desk accessory.
- "Text Attributes" describes how your application can check the current attributes of a range of text to determine which ones are consistent across the text. It also describes how you can manipulate the font, style, size, and color of a range of text.
- "Saving and Restoring a TextEdit Document, and Implementing Undo" describes how to save to disk the contents of a document created using TextEdit, and restore it when the user opens the document.
- "Customizing TextEdit" describes how to replace the default end-of-line, drawing, width-measuring, and hit test hook routines, use the multi-purpose low-memory global variable `TEDOText` hook routine, customize word selection and automatic scrolling, and determine the length of a line of text.

This section includes sample application-defined routines and code fragments that show some of the ways you can use TextEdit. These examples are provided for illustrative purposes only; they are not meant to be used in applications you write.

Note

For both monostyled and multistyled edit records, the text is limited to 32 KB. Whenever you insert or paste text, you need to ensure that adding the new text does not exceed the 32 KB limit. Your application can check for this limit before you insert or paste text. ◆

Getting Started With TextEdit

You can use TextEdit to display static text, for example, in a dialog box; the TextEdit procedure that you use to do this creates its own edit record. You can use TextEdit to display and manipulate modifiable text, for which purpose you must first create an edit record. This section discusses these two uses of TextEdit. It describes how you create an edit record and bring existing text into its text buffer, then set the text selection range or insertion point, scroll the text, and, finally, release the memory allocated for the edit record when you are finished with it. The topics are described in the following order:

- preparing to use TextEdit
- displaying static text
- creating an edit record
- setting the text of an edit record
- setting the selection range or the insertion point
- scrolling text
- disposing of an edit record

Preparing to Use TextEdit

This section describes two basic tasks that your application needs to perform before using TextEdit. It must

- determine the installed version of TextEdit
- initialize other managers and TextEdit

To determine the installed version of TextEdit, you use the Gestalt Manager, which is fully documented in the chapter "The Gestalt Manager" in *Inside Macintosh: Operating System Utilities*.

You can get information about the current version of TextEdit using the `Gestalt` function with the `Gestalt` selector `gestaltTextEditVersion`, which returns one of the values listed and described below. In this list, a new feature is shown only when it is first introduced in the software, although it is part of TextEdit in succeeding versions. For system software version 6.0.4, different patches were made to TextEdit for different hardware platforms. In these cases, unique values are returned that also identify the hardware.

Returned value	New features	System software/hardware
`gestaltUndefSelectorErr`	Multistyled TextEdit	Systems before 6.0.4/all hardware
`gestaltTE1`		System 6.0.4 Roman script system/IIci-family hardware
`gestaltTE2`	New width measurement hook Script Manager compatible	System 6.0.4 non-Roman script system/IIci-family hardware
`gestaltTE3`		System 6.0.4 non-Roman script system/all non-IIci family hardware
`gestaltTE4`	`TEFeatureFlag`	System 6.0.5/all hardware
`gestaltTE5`	Text width measurement hook	System 7.0/all hardware

You need to initialize other managers and TextEdit before your application calls any TextEdit routines, including `TEInit`. First, you initialize QuickDraw, the Font Manager, and the Window Manager, and then TextEdit, in that order. To do this, call the following routines from an initialization procedure that is called from your application's main routine.

```
BEGIN
   InitGraf(@thePort);
   InitFonts;
   InitWindows;
   InitMenus;
   TEInit;
   ....
```

In addition to initializing miscellaneous global variables, such as `TEDoText` and `TERecal`, the `TEInit` procedure sets up the private scrap and allocates a handle to it.

Note
You should call `TEInit` even if your application doesn't use TextEdit so that desk accessories and dialog and alert boxes, which use TextEdit routines, work correctly. ◆

Displaying Static Text

TextEdit provides an easy way for your application to display static text whether or not it uses other TextEdit features to implement editing services. The `TETextBox` procedure displays unchanging text that you cannot edit. You don't create an edit record because the `TETextBox` procedure creates its own edit record, which it deletes when it's finished with it.

The `TETextBox` procedure draws the text in a rectangle whose size you specify in the local coordinates of the current graphics port. You can also specify how text is aligned in the box. Text can be right aligned, left aligned, or centered.

You can use any of the following constants to specify how text is aligned in the box that `TETextBox` creates.

Constant	Description
`teFlushDefault`	Default alignment according to the primary line direction
`teCenter`	Center for all scripts
`teFlushRight`	Right for all scripts
`teFlushLeft`	Left for all scripts

Listing 2-1 shows how to use `TETextBox`. The first parameter is a pointer to the text to be drawn, which is a Pascal string. Because Pascal strings start with a length byte, you need to advance the pointer one position past the beginning of the string to point to the start of the text.

Listing 2-1 Using `TETextBox` to draw static text

```
str   := 'String in a box';
SetRect(r,100,100,200,200);
TETextBox(POINTER(ORD(@str)+1),LENGTH(str),r,teCenter);
FrameRect(r);
```

Creating an Edit Record

To use all other TextEdit routines in your application except the `TETextBox` procedure, first you need to create an edit record. This section discusses how to create an edit record. It also describes

- which type of edit record to use, monostyled or multistyled, and why
- some ways to store the edit record handle that the function returns when you create an edit record
- what to consider when you specify values for the destination and view rectangles when you create an edit record
- how TextEdit initializes those edit record fields that are used differently for monostyled and multistyled edit records, and those that are used the same

The TEStyleNew function allocates a multistyled edit record which contains text with character attribute information that can vary from character to character. The TENew function allocates a monostyled edit record which contains text in a single font, face, and size. (Before your application calls either of these functions, the window must be the current graphics port.)

If your application supports only monostyled text, use TENew to avoid the unnecessary allocation of additional data structures used to store character attribute information for multistyled edit records. You can use TEStyleNew in this case also, although it is not recommended.

Both TENew and TEStyleNew return a handle to the newly created record. Most TextEdit routines require you to pass this handle as a parameter, so your application needs to store it using any of the following methods:

- You can store the edit record handle in a private data structure whose handle is stored in your application window's refcon field.
- You can create a record in which to store information about the window, and include a field to store the edit record handle. Listing 2-2 provides an example of this method.
- You can define a variable in your application for each edit record handle, and then use the variable to store the handle.

Listing 2-2 shows a sample document record declaration for an application that handles text files. The document record is an application-specific data structure that contains the handle to the edit record, and any controls for scroll bars.

Listing 2-2 A sample document record

```
TYPE
   MyDocRecHnd =  ^MyDocRecPtr;
   MyDocRecPtr =  ^MyDocRec;
   MyDocRec    =
   RECORD
      editRec:       TEHandle;      {handle to TextEdit record}
      vScrollBar:    ControlHandle; {vertical scroll bar}
      hScrollBar:    ControlHandle; {horizontal scroll bar}
   END;
```

To associate an application-defined document record with a particular window, you can set a handle to that record as the reference constant of the window by using the Window Manager procedure SetWRefCon. This technique is described further in the chapter "Introduction to File Management" in *Inside Macintosh: Files*.

When you create an edit record, you specify the area in which the text is drawn as the destination rectangle, and the portion of the window in which the text is actually displayed as the view rectangle.

To ensure that the first and last characters in each line are legible in a document window, you can inset the destination rectangle at least four pixels from the left and right edges of the graphics port (20 pixels from the right edge if the window contains a scroll bar or size box).

The destination rectangle must always be at least as wide as the first character drawn. The view rectangle must not be empty; for example, if you do not want any text visible, specify a rectangle off the screen—don't make its trailing edge less than its leading edge.

Editing operations may lengthen or shorten the text. The bottom of the destination rectangle can extend to accommodate the end of the text. In other words, you can think of the destination rectangle as bottomless. The sides of the destination rectangle determine the beginning and the end of each line of text, and its top determines the position of the first line.

Your program should not have a destination rectangle that is wider than the view rectangle if you are displaying mixed-directional text. For example, the Dialog Manager makes the destination rectangle extend twice as far on the right as the view rectangle, so that horizontal scrolling can be used in normal dialog boxes. When the Arabic script system is installed, this extension is disabled, because the text may be right aligned, and therefore out of view. Your application can include the following code to check that the destination and view rectangles have the same width.

```
IF scriptsInstalled > 1 THEN
   IF GetEnvirons (smBidirect)<>0 THEN
      BEGIN
      {make the rectangles the same width}
      END;
```

When you create an edit record, TextEdit initializes the record's fields, based on values in the current graphics port record and the kind of edit record you create. Although most edit record fields are initialized similarly for both monostyled and multistyled edit records, there are some fields that are used differently, and their initial values depend on how they are used.

For a monostyled edit record that you create by calling `TENew`, the `txSize`, `lineHeight`, and `fontAscent` fields of the edit record hold actual values reflecting the text size, the line height, and the font ascent. Because the text is monostyled, these values apply to all of the text of the edit record.

- The `txSize` field is set to the value of the current graphics port's text size (`txSize`) field, which indicates that all text is set in a single font, size, and face.
- The value of the `lineHeight` field specifies the fixed vertical distance from the ascent line of one line of text down to the ascent line of the next. The line height corresponds to the ascent plus descent for the font and leading to create single-spacing for the lines in the new edit record.

- The value of the `fontAscent` field specifies how far above the base line the pen is positioned to draw the caret or to highlight the text. For single-spaced text, this is the ascent of the text in pixels (the height of the tallest characters in the font from the base line). The font ascent corresponds to the ascent of the font indicated by the `txFont` and `txSize` fields of the current graphics port.

Note
To adjust the spacing for a monostyled edit record, you can alter the values in the `fontAscent` and `lineHeight` fields of the edit record. ◆

For more information, see the discussion of font measurements in the chapter "Font Manager" in this book.

For a multistyled edit record, `TEStyleNew` initializes the `txSize`, `lineHeight`, and `fontAscent` fields of the edit record to –1. A value of –1 in each of these fields means:

- `txSize`

  The edit record contains associated character attribute information and the `txFont` and `txFace` fields combine to contain the text style record handle for the character attribute information.

- `lineHeight`

  The vertical distance from the ascent line of one line of text down to the ascent line of the next is calculated independently for each line, based on the maximum value for any individual character attribute on that line. These values are stored in the line height table (`LHTable`).

- `fontAscent`

  The font ascent is calculated independently for each line, based on the maximum value for any individual character attribute on that line. These values are stored in the line height table (`LHTable`).

For both multistyled and monostyled records, the following fields are initially set to the same values:

- The record initially contains no text. The text handle (`hText`) points to a zero-length block in the heap, and the text length field (`teLength`) of the edit record is set to 0. To furnish text to be edited, you use the `TESetText` procedure if you are incorporating existing text and the `TEKey` procedure if the user is entering text.
- The value of the `just` field determines the alignment of text in the edit record. The default value is `teFlushDefault`, indicating that the alignment is to follow the primary line direction. For languages that are read from left to right, the default value is left; for languages that are read from right to left, the default value is right. To change the alignment of text in the record, you use the `TESetAlignment` procedure.
- The `selStart` and `selEnd` fields are initially set to 0; this places the insertion point at the beginning of the text.
- The edit record uses the drawing environment of the graphics port specified by the `destRect` and `viewRect` parameters. These parameters contain the local coordinates of rectangles within the current graphics port, which becomes the graphics port for the new edit record. The text in the new edit record is to have the characteristics of the current graphics port.

Listing 2-3 shows the `MyAddTE` function, which is a sample application-defined function that creates a new multistyled edit record for an existing window. The `TEStyleNew` function call returns a handle to the edit record that it creates. The code stores the handle in the `docTE` variable. The `TEAutoView` procedure call turns on automatic scrolling for the newly created edit record. For a complete discussion of scrolling, see the chapter "Control Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Listing 2-3 Creating a multistyled edit record

```
FUNCTION MyAddTE (myWindow: WindowPtr): TEHandle;
VAR
   destRect, viewRect:  Rect;
   docTE:               TEHandle;
   CONST
   kMaxDocWidth = 576;
BEGIN
   MyGetTERect(myWindow, viewRect); {get TextEdit rectangle}
   destRect := viewRect;
   destRect.right := destRect.left + kMaxDocWidth;
   docTE := TEStyleNew(destRect, viewRect);
   IF docTE <> NIL THEN
      BEGIN
         TEAutoView(TRUE, docTE);
         docTE^^.clikLoop := @AsmClikLoop;
      END;
   MyAddTE := docTE;
END;
```

Specifying the Destination and View Rectangles

When you create an edit record, whether monostyled or multistyled, you specify the area in which the text is drawn as the destination rectangle, and the portion of the window in which the text is actually displayed as the view rectangle.

To ensure that the first and last glyphs in each line are legible in a document window, you can inset the destination rectangle at least four pixels from the left and right edges of the graphics port (20 pixels from the right edge if the window contains a scroll bar or size box).

The destination rectangle must always be at least as wide as the first glyph drawn. The view rectangle must not be empty; for example, if you do not want any text visible, specify a rectangle off the screen—don't make its trailing edge less than its leading edge.

Editing operations may lengthen or shorten the text. The bottom of the destination rectangle can extend to accommodate the end of the text. In other words, you can think of the destination rectangle as bottomless. The sides of the destination rectangle determine the beginning and the end of each line of text, and its top determines the position of the first line.

Your program should not have a destination rectangle that is wider than the view rectangle if you are displaying mixed-directional text. For example, the Dialog Manager makes the destination rectangle extend twice as far on the right as the view rectangle, so that horizontal scrolling can be used in normal dialog boxes. When the Arabic script system is installed, this extension is disabled, because the text may be right aligned, and therefore out of view. Your application can include the following code to check that the destination and view rectangles have the same width.

```
IF scriptsInstalled > 1 THEN
   IF GetEnvirons (smBidirect)<>0 THEN
      BEGIN
      {make the rectangles the same width}
      END;
```

Setting the Text of an Edit Record

When you create an edit record, it doesn't contain any text until either the user enters text through the keyboard or opens an existing document. This section describes how to specify *existing* text to be edited. "Accepting Text Input Through Key-Down Events" on page 2-37 discusses how to insert text that the user enters through the keyboard.

When a user opens a document, your application can bring the document's text into the text buffer of an edit record by calling `TESetText`. If the text has associated character attribute information, your application also needs to manage it.

There are two ways to specify existing text to be edited. The easier method is to use `TESetText`, which creates a copy of the text and stores the copy in the existing handle of the edit record's `hText` field. One of the parameters that you pass to `TESetText` specifies the length of the text. The `TESetText` procedure resets the `teLength` field of the edit record with this value and uses it to determine the end of the text; it sets the `selStart` and `selEnd` fields to the last byte offset of the text so that the insertion point is positioned at the end of the displayed text. The `TESetText` procedure calculates line breaks, eliminating the need for your application to do this.

You can use the second method to save space if you have a lot of text. Using this method, you can bring text into an edit record by directly changing the `hText` field of the edit record, replacing the existing handle with the handle of the new text. When you do this for a monostyled edit record, you need to modify the `teLength` field to specify the length of the new text, and then call `TECalText` to recalculate the `lineStarts` array and `nLines` values to match the new text.

Using the second method is somewhat more complicated for multistyled text because `TECalText` does not update the style run table (`StyleRun`) properly. To compensate for this, your application needs to perform the following tasks:

- Before changing the edit record's `hText` field, reduce the style run table to one entry. Do this by setting the edit record's `selStart` field to 0 and its `selEnd` field to 32767, then call `TESetStyle`.
- Before calling `TECalText`, set the start character (`startChar`) field of the style run table to the length of the new text plus one, that is:

```
TEStyleRec.runs[1] to length(hText)+1
```

Using the same edit record for different pieces of text

Rather than allocate a new edit record for each piece of text you want to edit, you can use the same record to edit different pieces of text. For example, you can create an edit record and either accept user input or call `TESetText` to incorporate existing text. If you know that you'll want to edit the text again whose handle is currently stored in the `hText` field, first you need to save the text before you call `TESetText`, because `TESetText` uses the same handle, resizing it for the new text, if necessary. ◆

The `TESetText` procedure doesn't affect the text drawn in the destination rectangle, so call the Window Manager's `InvalRect` procedure afterward, if necessary. For more information about the `InvalRect` procedure, see the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Setting the Selection Range or the Insertion Point

You can use the `TESetSelect` procedure to specify the selection range or the position of the insertion point as determined by the application. For example, you can use `TESetSelect` to highlight an initial default value in an application such as an online data-entry form, or to position the caret at the start of the field where you want the user to enter a value. You can also use it to implement a Select All menu command.

You can set the selection range (or insertion point) to any character positions within the text of the edit record corresponding to byte offsets 0 through 32767. To select a range of text, you pass `TESetSelect` the handle to the edit record along with the byte offsets corresponding to the beginning and the ending characters of the text to be highlighted. The `TESetSelect` procedure modifies the `selStart` and `selEnd` fields of the edit record.

To display a caret at an insertion point, specify the same value for both the `selStart` and `selEnd` parameters. To encompass the edit record's entire text block as the selection range, specify 0 as the value of `selStart` and 32767 as the value of `selEnd`. You can implement a Select All menu command by specifying the edit record's entire range of text, as shown in the following code fragment, by using the `teLength` field.

```
iSelectAll:
   TESetSelect(0, myTERec^^.teLength, myTERec);
```

Scrolling Text

Using TextEdit routines, your application can allow the user to control text scrolling through the scroll bars; in this case, you scroll the text by calling a TextEdit procedure. It can also automatically scroll the text of an edit record into view when the user clicks in the view rectangle, and then drags the mouse outside of it, if you enable automatic scrolling through another TextEdit procedure.

To scroll the text when a mouse-down event occurs in a scroll bar, your application needs to determine how far to scroll the text. For example, to vertically scroll the text of a monostyled edit record, you can use the `lineHeight` field of the edit record to calculate the number of pixels to scroll; you multiply every click in the scroll bar by the number of pixels in the `lineHeight` field and by the number of lines displayed in the view rectangle. For multistyled text, you need to use the value of the `lhHeight` field of the line height table for each line in the view rectangle because line height can vary from line to line.

To scroll the text, you call either `TEScroll` or `TEPinScroll` specifying the number of pixels to scroll. The only difference between `TEScroll` and `TEPinScroll` is that `TEPinScroll` stops scrolling when the last line is scrolled into the view rectangle.

When the user clicks in the scroll arrow pointing down, you scroll the text up. When the user clicks in the scroll arrow pointing up, you scroll the text down. Passing a positive value to either routine moves the text right and down, passing a negative value moves the text left and up. The destination rectangle is offset by the amount you scroll. For example, the following call scrolls the text of a monostyled edit record up one line.

```
TEScroll(0, -hTE^^.lineHeight, hTE)
```

There are two ways to enable or disable automatic scrolling for an edit record. You can use the `TEAutoView` procedure or the `teFAutoScroll` feature of the `TEFeatureFlag` function. However, neither of these routines actually scrolls the text. To ensure that the selection range is always visible, your application should call `TESelView`. When automatic scrolling is turned on, `TESelView` scrolls the selection range into view, if necessary.

Listing 2-3 on page 2-29 creates a multistyled edit record and turns on automatic scrolling for it. It saves the address of the default click loop procedure installed in the edit record's `clikLoop` field, then replaces it with the address of its own customized click loop routine.

The `clikLoop` field of the edit record contains the address of a click loop procedure that is called continuously as long as the mouse button is held down. When automatic scrolling is turned on, the default click loop routine determines if the mouse has been dragged out of the view rectangle; if it has, the default click routine scrolls the text using `TEPinScroll`. For example, if the user clicks in the text and drags the mouse outside of it to the right, the text is automatically scrolled left.

How much the text is scrolled vertically is determined by the `lineHeight` field of the edit record for a monostyled edit record and by the `lhHeight` field of the line height table for a multistyled edit record.

Scroll bars are not scrolled automatically with the text if the default click loop routine is used. However, you can replace the default click loop routine with a routine that updates scroll bars. For more information about customizing scrolling, see "Customizing Automatic Scrolling" on page 2-62. For a complete discussion of scrolling, see the chapter "Control Manager" in *Inside Macintosh: Macintosh Toolbox Essentials.*

Disposing of an Edit Record

When your application is completely finished with an edit record, you should release any memory allocated for it by calling `TEDispose`. To continue to refer to the text once you've destroyed the edit record, use the Operating System Utilities `HandToHand` function before you call `TEDispose`. It copies the text (whose handle is stored in the edit record's `hText` field), and returns a new handle to it. (See *Inside Macintosh: Operating System Utilities* for more information.) For a multistyled edit record, you also need to save the character attribute information. If your program retains the original handle to the text stored in the `hText` field after you call `TEDispose`, the handle becomes invalid because the text is removed—the memory used for it is deallocated.

Responding to Events Using TextEdit

This section discusses some of the TextEdit routines that your application can call in response to event notification. You can use TextEdit routines to

- handle idle processing in response to null events (`TEIdle`)
- identify the active edit record in response to an activate event (`TEActivate` and `TEDeactivate`)
- handle mouse-down events (`TEClick`)
- update the destination rectangle in response to an update event (`TEUpdate`)
- handle key-down events (`TEKey`)

Handling a Null Event

Your program needs to call `TEIdle` whenever it receives a null event. If there is more than one edit record associated with an active window, make sure you pass `TEIdle` the handle to the currently active edit record. (See "Activating an Edit Record" in the following section for more information.)

If you have turned on text buffering through the `TEFeatureFlag` function, you should call `TEIdle` before any pause of more than a few ticks—for example, before `WaitNextEvent`. A possibility of a long delay before characters appear on the screen exists—especially in non-Roman systems. Blinking the caret alerts the user to this delay.

To blink the caret at a constant frequency, you should call `TEIdle` at least once through your main event loop—otherwise, the caret blinks irregularly. No matter how often you call `TEIdle`, the time between blinks is never to be less than the minimum interval.

Listing 2-4 shows a sample application-defined procedure, `MyDoIdle`, that calls `TEIdle` to handle a null event.

Listing 2-4 An idle-processing procedure

```
PROCEDURE MyDoIdle(myWindow: WindowPtr);
VAR
   myData:     MyDocRecHnd;   {handle to a document record}
   myTERec:    TEHandle;      {handle to TextEdit record}
BEGIN
   myData   := MyDocRecHnd(GetWRefCon(myWindow));
   IF myData <> NIL THEN
      BEGIN
         myTERec  := myData^^.editRec;
         IF myTERec <> NIL THEN
            TEIdle(myTERec);
      END;
END;
```

Note
The value stored in the low-memory global `CaretTime` determines the blinking time for the caret. (The user can also set the minimum interval through the General Controls control panel.) You can use the Event Manager's `GetCaretTime` function to retrieve this value. For more information, see the chapter "The Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*. ◆

Activating an Edit Record

When a window becomes active or inactive, the Window Manager updates the frames of the windows on the screen, and then informs the Event Manager that an activate event has occurred. The next time `WaitNextEvent` is called from your main event loop, the Event Manager notifies your application that an activate event has occurred. (An activate event can have a flag set indicating that a window is to be deactivated.) When your application receives this notification, it needs to call `TEActivate` for an activate event and `TEDeactivate` for a deactivate event. When you call `TEActivate`, you pass it the handle to the edit record to be activated; when you call `TEDeactivate`, you pass it the handle to the currently active edit record.

An application can have more than one edit record associated with it. The active edit record is the one where the next editing operation is to take place. The `TEActivate` procedure identifies an edit record as the active one by either highlighting the selection range or displaying a caret at the insertion point. The `TEDeactivate` procedure changes an edit record's status from active to inactive and removes the highlighting or the caret. If outline highlighting is on, `TEDeactivate` frames the selection range or displays a dimmed caret.

Note
The `TEActivate` procedure does not set the selection range; it uses the current values in the `selStart` and `selEnd` fields of the edit record to highlight the specified text or display a caret at the insertion point. The `TEDeactivate` procedure does not affect the current settings of these fields. ◆

Before you can activate an edit record, you need to deactivate the currently active edit record, if there is one. If your application has a routine which it calls to activate and deactivate its own windows, you can include processing in that routine to make an edit record the active one or make the currently active record inactive. Because deactivate events happen before activate events, these events occur in the proper order when the user switches from one window to another.

If there is more than one edit record associated with a window, you'll probably want to call `TEDeactivate` whenever the mouse button is clicked in an edit record other than the active one. In this case, each `TEDeactivate` call not associated with a window deactivate event would be coupled with a call to `TEActivate`.

You can modify the text of an edit record associated with a background window; however, to do so, you need to call `TEActivate` for that edit record before you call any other TextEdit routines.

Note
When you use `TEClick` and `TESetSelect` to set the selection range or insertion point, the selection range is not highlighted nor is a blinking caret displayed at the insertion point until the edit record is activated through `TEActivate`. However, if you had already turned on outline highlighting (through the `TEFeatureFlag` function), the text of the selection range is framed or a gray, unblinking caret is displayed at the insertion point. ◆

Handling Mouse-Down Events

When your application receives notification of a mouse-down event that it determines TextEdit should handle, it needs to pass the click on to the `TEClick` procedure. Before calling `TEClick`, your application needs to perform the following steps:

1. Convert the mouse location that is passed in the event record from global to local coordinates, so that it can pass those local coordinates to `TEClick`. To perform the conversion, you can use the `GlobalToLocal` QuickDraw procedure. (For more information, see *Inside Macintosh: Imaging*.)
2. Determine if the Shift key was held down at the time of the click to extend the selection. The behavior of `TEClick` depends on the user's actions.
   - If the Shift key was down, `TEClick` extends the current selection range.
   - If the Shift key was not held down, `TEClick` removes highlighting of the current selection range and positions the insertion point as close as possible to the location where the mouse click occurred.

- When the mouse is moved or dragged, `TEClick` expands or shortens the selection range a character at a time. The `TEClick` procedure keeps control until the user releases the mouse button.
- If the mouse button is clicked twice (a double-click), `TEClick` extends the selection to include the entire word where the cursor is positioned.

Note
As long as the mouse button is held down, `TEClick` repeatedly calls the click loop routine pointed to from the `clikLoop` field of the edit record. ◆

Listing 2-5 shows an application-defined procedure, `MyDoContentClick`, that calls `TEClick`, passing it a mouse-down event.

Listing 2-5 Passing a mouse-down event to TextEdit

```
PROCEDURE MyDoContentClick (myWindow: WindowPtr; event: EventRecord);
VAR
   myData:  MyDocRecHnd;      {handle to a document record}
   myTERec: TEHandle;         {handle to TextEdit record}
   mouse:   Point;
BEGIN
   myData   := MyDocRecHnd(GetWrefCon(myWindow));  {get window's data record}
   IF myData = NIL THEN
      exit(MyDoContentClick);
   myTERec  := myData^^.editRec;                   {get TERec}
   IF myTERec = NIL THEN
      exit(MyDoContentClick);
   SetPort(myWindow);
   mouse := event.where;                     {get the click position}
   GlobalToLocal(mouse);                     {convert to local coordinates}
   IF PtInRect(mouse, myTERec^^.viewRect) THEN
      BEGIN
         shiftDown := BAnd (event.modifiers, shiftKey) <> 0;
                           {extend if Shift is down}
         TEClick(mouse, shiftDown, myTERec);
      END;
END;
```

When `TEClick` is called, the `clickTime` field of the edit record contains the time when `TEClick` was last called. When `TEClick` returns, it sets the `clickTime` field, adjusting the current tick count. The default click loop procedure uses this value.

Responding to an Update Event

After changing any fields of the edit record that affect the appearance of the text or after any editing or scrolling operation that alters the onscreen appearance of the text, you need to call `TEUpdate`.

Your application needs to call `TEUpdate` every time the Event Manager function `WaitNextEvent` reports an update event for a text editing window—after you call the Window Manager procedure `BeginUpdate`, and before you call the `EndUpdate` procedure. You call the following routines when an update event occurs:

```
BeginUpdate(myWindow);
EraseRect(myWindow^.portRect);
TEUpdate(myWindow^.portRect, hTE);
EndUpdate(myWindow);
```

If you don't include the `EraseRect` procedure, the caret may sometimes remain visible when the window is deactivated. For more information about responding to events, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*. For more information about the Window Manager, see the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Accepting Text Input Through Key-Down Events

When the user enters text through the keyboard, your application needs to call the `TEKey` procedure to accept the keyboard input a byte at a time or to delete a character when the user backspaces over it. Call `TEKey` every time the Event Manager function `WaitNextEvent` reports a key-down event that your application determines TextEdit should handle.

Because `TEKey` accepts every character it is passed, your application needs to first filter out Command-key equivalents, special keys, and nonprinting characters as appropriate, such as Enter or Tab, and only pass `TEKey` a text, a Return key character, an arrow key character, or a backspace key character.

Note
If you want to display the text as multiple paragraphs, don't filter out Return key characters. ◆

Listing 2-6 shows the `MyHandleKeyDown` procedure which calls `TEKey` to accept text a character at a time. First `MyHandleKeyDown` filters out special characters. For example, it treats the Tab key as a special character, and calls an application-defined routine, `MyDoTab`, to handle this character appropriately for the document. Then it checks to make sure that inserting the character won't exceed the maximum text length allowed. It does not count the Delete or arrow keys because they are not text characters.

If the maximum text length is not exceeded, the code passes the character to `TEKey`. Otherwise, it calls an application-defined routine, `MyAlertUser`, to notify the user that the character is not inserted, and that inserting it would exceed the edit record text

limitation. In this example listing, the maximum text length is set to the highest possible value; you can specify a lower limit.

Listing 2-6 Inserting text in a document

```
PROCEDURE MyHandleKeyDown(myWindow: WindowPtr; event: EventRecord);
CONST
   kMaxTELength = 32767;
   kTab = $09;
   kDel = $08;
   kRightArrow = $1D;
   kLeftArrow = $1C;
   kDownArrow = $1F;
   kUpArrow = $1E;
VAR
   myData:     MyDocRecHnd;      {handle to a document record}
   myTERec:    TEHandle;         {handle to TextEdit record}
   key:        CHAR;
BEGIN
   myData   := MyDocRecHnd(GetWRefCon(myWindow));  {get window's data record}
   IF myData = NIL THEN
      exit(MyDoContentClick);
   myTERec  := myData^^.editRec;                   {get TERec}
   IF myTERec = NIL THEN
      exit(MyDoContentClick);
   key := CHR(BAnd(event.message, charCodeMask));
   IF key = char(kTab) THEN {handle special characters}
      MyDoTab(event)
   ELSE
      BEGIN
         IF (key = CHR(kDel)) | (key = CHR(kRightArrow)) |
         (key = CHR(kLeftArrow)) | (key = CHR(kUpArrow)) |
         (key = CHR(kDownArrow)) | {don't count deletes or arrow keys}
         (LongInt(myTERec^^.teLength - MyGetTESelLength(myTERec) + 1 <
          kMaxTELength)
         THEN
            BEGIN
               TEKey(key, myTERec); {insert character in document}
               MyAdjustScrollbars(window, FALSE);
            END
         ELSE
```

```
            MyAlertUser(eExceedChar);
      END;
END;
```

Before testing to ensure that the input character does not exceed the edit record's text limitation, the code subtracts the length of the selection range, which the inserted character is to replace, from the current length of the text. To get the length of the selection range, the code calls an application-defined function, `MyGetTESelLength`. Listing 2-7 shows this function. Several other sample application-defined routines in this chapter also call this function.

Listing 2-7 Getting the selection range length

```
FUNCTION MyGetTESelLength (myTERec: TEHandle): Integer;
   Begin
      MyGetTESelLength := myTERec^^.selEnd - myTERec^^.selStart;
   END;
```

If the selection range is an insertion point and the key is not an arrow key character or a Backspace key character, `TEKey` inserts the character before the insertion point. When the character direction is right-to-left, the character is inserted to the right of the insertion point. When the character direction is left-to-right, the character is inserted to the left of the insertion point.

When you call `TEKey` and the keyboard script is different from the font script, TextEdit changes the font script to correspond to the keyboard script. If the font at the insertion point is the same as the keyboard script, then this font is used. If a font was written to the TextEdit style scrap record (in the null scrap) and never used and that font script coincides with the keyboard script, then it is used. Otherwise, TextEdit searches through the fonts in the style table until it locates a font that corresponds to the keyboard. If one does not exist, then it uses the application font.

When the user backspaces over characters of a multistyled edit record, `TEKey` deletes the characters but it saves the character attributes associated with the last character deleted in order to apply it to any new characters that the user might enter; the character attributes are saved in the null scrap's style scrap record. As soon as the user clicks in another area of the text, `TEKey` clears the attributes from the null scrap.

Moving Text In and Out of Edit Records

This section describes how to cut, copy, and paste text, and insert and delete it. Because TextEdit manages the varying character attribute information associated with multistyled text, you use separate routines for monostyled and multistyled text to perform some of these tasks; this section explains those differences. If your application supports both monostyled and multistyled text, you need to handle these cases separately.

Using TextEdit to Cut, Copy, and Paste Text

You can use TextEdit to cut, copy, and paste text within a single edit record, between edit records, or across applications, and to handle menu commands that let the user perform these actions. You use the `TECut` and `TECopy` procedures to cut and copy both monostyled and multistyled text. To paste monostyled text, you use the `TEPaste` procedure. To paste multistyled text, you use the `TEStylePaste` procedure. To move monostyled text across applications or between an application and a desk accessory, you use the `TEFromScrap` and `TEToScrap` functions. This section describes how to use these routines and what they do.

Note
This section and those that follow do not describe how to create menus and their commands. For guidelines and a complete discussion of how to create and manage the menus in your application, see the chapter "The Menu Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*. ◆

The `TECut` procedure removes and transfers the selected text. The `TECopy` procedure copies the selected text, leaving the original text intact. To implement cut-and-paste or copy-and-paste services, you can couple either of these calls with `TEPaste` or `TEStylePaste` to overlay a text selection or insert the text to be pasted at an insertion point.

To cut, copy, and paste text within the same edit record or between two edit records within the same application, you do not need to write the text to and from the desk scrap, although this is always done automatically for multistyled text. However, to carry text across applications or between an application and a desk accessory, whether the text is multistyled or monostyled, you must write it to and from the desk scrap.

For monostyled text, `TECut` and `TECopy` write the text to the private scrap only. The `TEPaste` procedure pastes the monostyled text from the private scrap to the edit record. To determine the length of the text to be pasted, you can call the `TEGetScrapLength` function which returns the size in bytes of the text in the private scrap, or you can check the value of the global variable `TEScrapLength`.

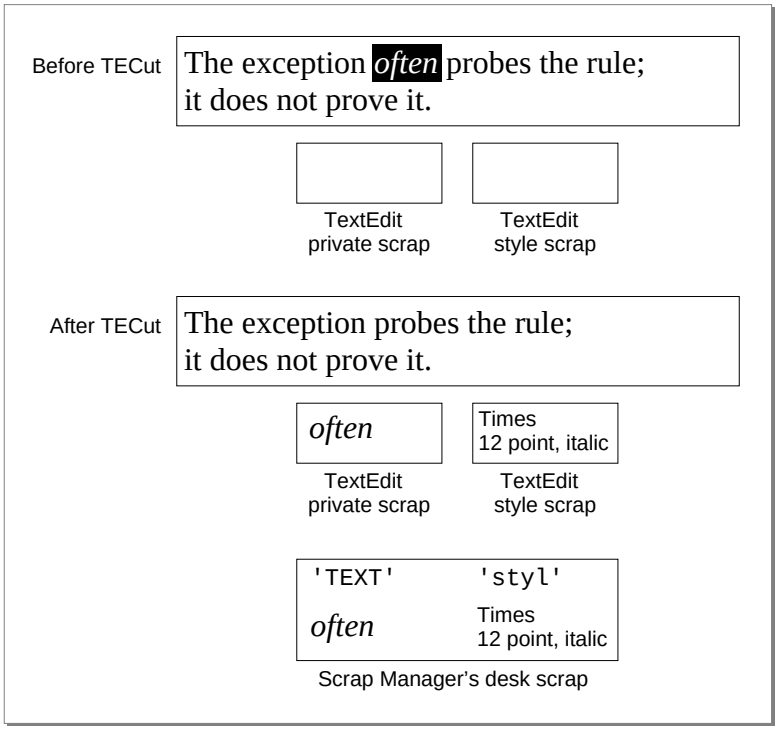
To move monostyled text across applications or between an application and a desk accessory, you need to use the `TEFromScrap` and `TEToScrap` functions, which write text to and from the desk scrap.

For multistyled text, `TECut` and `TECopy` always write both the text and its associated character attribute information to the Scrap Manager's desk scrap under scrap types `'TEXT'` and `'styl'`. For more information, see the chapter "Scrap Manager" in *Inside Macintosh: More Macintosh Toolbox*.

The `TEStylePaste` procedure reads both the text and its attributes back from the desk scrap and writes the multistyled text into the edit record's text buffer at the current selection range or insertion point.

You can use these procedures to move multistyled text across two applications or between an application and a desk accessory; you don't need to call `TEFromScrap` and `TEToScrap` for multistyled text. To either copy or move the text selection from the text buffer to the desk scrap, `TECut` and `TECopy` write the text to the private scrap and to the Scrap Manager's desk scrap. To copy or move the attributes along with the text, `TECut` and `TECopy` write the character attribute information stored in the style table to both the style scrap and the Scrap Manager's desk scrap. Figure 2-9 shows what happens when you cut multistyled text using `TECut`.

Figure 2-9 Cutting text from a multistyled edit record



The `TEStylePaste` procedure either pastes the text from the desk scrap at the insertion point or replaces the current selection range with the text to be pasted. Along with the text, `TEStylePaste` writes the character attribute information to the style record's style table and applies it to the inserted text.

For multistyled text, text is pasted from the desk scrap. Therefore, before you call `TEStylePaste`, use the Scrap Manager's `GetScrap` procedure to check the size of the text (`'TEXT'` data) to be pasted.

To calculate the amount of memory required for the style scrap before you cut or copy multistyled text, you can use the information returned by the `TENumStyles` function. This function returns the number of attribute changes contained in a range of text. Since the style scrap is linear in structure, with one element for each attribute change, you can multiply the number returned by `TENumStyles` by `SizeOf(ScrpSTElement)` and add 2 to get the number of bytes needed.

Listing 2-8 shows a sample application-defined procedure that handles cut, copy, and paste menu commands. Before the application pastes the multistyled text into the edit record's text at the current selection range, it calls the Scrap Manager's `GetScrap` function to get the size of the text to be pasted. The code adds the returned value to the size of the text in the edit record, subtracts the size of the selection range, then compares the result against the maximum length of the edit record text to make sure that pasting the text won't exceed it. (To get the selection range length, the code calls the application-defined function `MyGetTESelLength`, as shown in Listing 2-7 on page 2-39.)

To avoid copying the data when you want only the length of the text returned, pass a value of `NIL` for the `hDest` parameter to `GetScrap`. For more information about `GetScrap`, see the chapter "Scrap Manager" in *Inside Macintosh: More Macintosh Toolbox*.

Listing 2-8 Handling Cut, Copy, and Paste commands on an Edit menu

```
PROCEDURE MyHandleEditMenu (myWindow: WindowPtr; menuItem: Integer);
CONST
   kMaxTELength = 32000;
   kTESlop  = 1024;
      {kTESlop provides some extra security when preflighting edit commands.}
VAR
   myData:           MyDocRecHnd;      {handle to a document record}
   myTERec:          TEHandle;         {handle to TextEdit record}
   myErr:            OSErr;
   offset:           LONGINT;
   aHandle:          Handle;
   oldSize, newSize: LONGINT;
   saveErr:          OSErr;
BEGIN
   myData   := MyDocRecHnd(GetWrefCon(myWindow));  {get window's data record}
   IF myData = NIL THEN
      exit(MyDoContentClick);
   myTERec  := myData^^.editRec;                   {get TERec}
   IF myTERec = NIL THEN
      Exit(MyDoContentClick);
   CASE menuItem OF
      iCut:
```

```
   BEGIN
      IF ZeroScrap = noErr THEN
         BEGIN
            PurgeSpace(total, contig);
            IF MyGetTESelLength(myTERec) + kTESlop >
             contig THEN
               MyAlertUser(eNoSpaceCut)
            ELSE
               TECut(myTERec);
         END;
   END;
iCopy:
   BEGIN
      IF ZeroScrap = noErr THEN
         TECopy(myTERec);
   END;
iPaste:
   BEGIN
      IF GetScrap(NIL, 'TEXT', offset) +
         (myTERec^^.teLength - MyGetTESelLength(myTERec)) >
         kMaxTELength
      THEN
         MyAlertUser(eExceedPaste)
      ELSE
         BEGIN
            aHandle := Handle(TEGetText(myTERec));
            oldSize := GetHandleSize(aHandle);
            newSize := oldSize + GetScrap + kTESlop
            SetHandleSize(aHandle, newSize);
               {see if handle can be resized}
            saveErr := MemError;
            SetHandleSize(aHandle, oldSize);
            IF saveErr <> noErr THEN
               MyAlertUser(eNoSpacePaste)
            ELSE
               TEStylePaste(myTERec);
         END;
      END;
   END;
END;
```

Inserting and Deleting Text

You can use TextEdit routines to delete and insert text. You use `TEInsert` to insert monostyled text into the edit record's text buffer if the current selection range is an insertion point. If the current selection range is a range of text, `TEInsert` replaces it with the text to be inserted. You use `TEStyleInsert` to insert multistyled text in the same way; however, the text *and* its associated character attribute information are inserted.

To delete text, your application calls the same routine whether the text is multistyled or monostyled. The `TEDelete` procedure removes the text of the current selection range. When the text is multistyled, `TEDelete` saves the character attributes in the null scrap to be applied to characters that the user might enter following the deletion. After each editing procedure, TextEdit redraws the text if necessary from the insertion point to the end of the text.

You can handle a Clear command using `TEDelete`; you call `TEDelete` with the handle to the edit record containing the text you want to eliminate. The `TEDelete` procedure removes the selected text without transferring it to the scrap.

```
iClear:
   TEDelete(myTERec);
```

Text Attributes

This section describes how your application can check the current attributes of a range of text to determine which ones are consistent across the text. It also describes how you can manipulate the font, style, size, and color of a range of text; the text selection can consist of a segment of text, the entire text of the edit record, a single character, or even an insertion point.

You use the `TEContinuousStyle` function to determine the current attributes for a range of text, and you use the `TESetStyle` procedure to change them. You can change character attributes singly, collectively, or in any combination using `TESetStyle`. For example, you can change the font style to bold or italic, and you can underline, outline, or shadow the selected text. You can increase or decrease the type size incrementally, or change the color in which the text is displayed. You use the `TESetAlignment` procedure to change the alignment of the entire text of an edit record.

This section describes these tasks in this order:

- checking the text attributes across a selection range
- toggling an attribute
- handling a font menu that lets the user change the font family
- handling a font size menu that lets the user change the text size
- handling a style menu that lets the user change the style of the text
- changing the text alignment

Some general information about `TESetStyle` that applies to many of the tasks for which you can use it is discussed here. If you call `TESetStyle` for an insertion point, TextEdit stores the input character attribute information in the null scrap's style scrap record. If the user then enters text (pastes without attributes, inserts, or types it), the attributes are written to the style record and applied to that text.

There are many ways in which you can use `TESetStyle` to handle menu commands that let the user modify text attributes. If your application allows a user to change any or all the text attributes from a single format menu before redrawing the text, you can make one call to `TESetStyle` specifying the particular attributes to be changed. If your application provides separate menus to let a user manipulate different aspects of the text, you can make separate calls to `TESetStyle` specifying the discrete text attribute to be changed.

Note
A field in the text style record is only valid if the corresponding bit is set in the `mode` parameter; otherwise, the field contains invalid information. ◆

The value of `mode` specifies which existing character attributes are to be changed to the new character attributes specified by `newStyle`.

Constant	Value	Description
`doFont`	1	Sets the font family ID
`doFace`	2	Sets the character style
`doSize`	4	Sets the type size
`doColor`	8	Sets the color
`doAll`	15	Sets all attributes
`addSize`	16	Increases or decreases the type size
`doToggle`	32	Modifies the mode

Checking the Text Attributes Across a Selection Range

When a particular attribute is set for an entire selection range, that attribute is said to be *continuous* over the selection. For example, in the selected text in Figure 2-10, the bold attribute is continuous over the selection range and italic is not.

Figure 2-10 Continuous attributes over a selection range

To determine the actual values for continuous attributes, you can use the `TEContinuousStyle` function. This function takes two variable parameters: `mode` and `aStyle`. For its input value, `mode` specifies the attributes to be checked; for its output value, `mode` specifies those attributes that are continuous over the selection range. For the input value of `aStyle`, you pass a pointer to a text style record (of type `TextStyle`); for those attributes that are continuous, the text style record fields contain the actual values when `TEContinuousStyle` returns.

A field in the text style record is only valid if the corresponding bit is set in the `mode` parameter; otherwise, the field contains invalid information. Possible values for the `TEContinuousStyle` `mode` parameter are defined by the following constants.

Constant	Value	Description
`doFont`	1	Specifies the font family number
`doFace`	2	Specifies the character style
`doSize`	4	Specifies the type size
`doColor`	8	Specifies the color
`doAll`	15	Specifies all the attributes

Listing 2-9 illustrates how to use the `TEContinuousStyle` function to determine the font, style, size, and color of the current selection range. The code sets the `mode` parameter. Then it calls `TEContinuousStyle`, passing it the text style record. When `TEContinuousStyle` returns, it checks each bit of the `mode` parameter to see which attributes are continuous across the selection.

Listing 2-9 Determining the font, style, size, and color of the current selection range

```
PROCEDURE MyGetCurrentSelection (VAR mode: Integer;
         VAR continuous: Boolean; VAR astyle: TextStyle;
         myTERec: TEHandle);
BEGIN
   mode := doFont + doFace + doSize + doColor;
   continuous := TEContinuousStyle(mode, aStyle, myTERec);
   IF BitAnd(mode, doFont) <> 0 THEN
      {font for selection = aStyle.tsFont}
   ELSE
      {more than one font in selection};
   IF BitAnd(mode, doFace) <> 0 THEN
      {aStyle.tsFace contains the text faces (or plain) that }
      { are common to the selection.}
   ELSE
      {No text face is common to the entire selection.};
   IF BitAnd(mode, doSize) <> 0 THEN
      {size for selection = aStyle.tsSize}
```

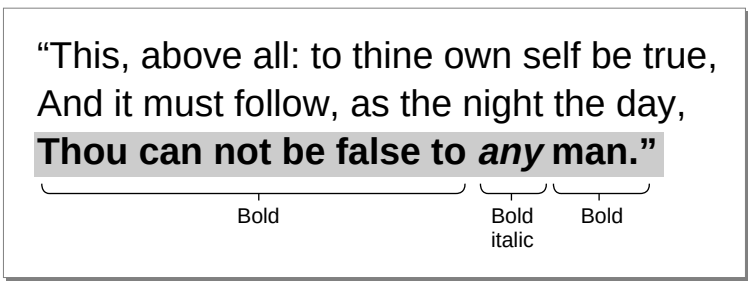
```
   ELSE
      {more than one size in selection};
   IF BitAnd(mode, doColor) <> 0 THEN
      {color for selection = aStyle.tsColor}
   ELSE
      {more than one color in selection}
END;
```


Toggling an Attribute

Once you know what attributes are continuous across a selection range, you can use `TESetStyle` to toggle an attribute on and off. For example, if you specify a `mode` parameter for `TESetStyle` that includes both `doToggle` and `doFace`, and an attribute that has been set in the `tsFace` field of the text style record exists across the current selection range, then `TESetStyle` removes that attribute. However, if the attribute isn't continuous over the current selection, then all of the selected text is set to include it.

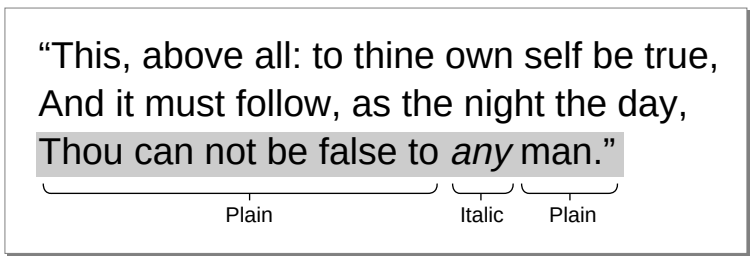
For example, in the selected text shown in Figure 2-11, the bold style is continuous over the selection range and the italic style is not.

Figure 2-11 An initial selection before `TESetStyle` is called



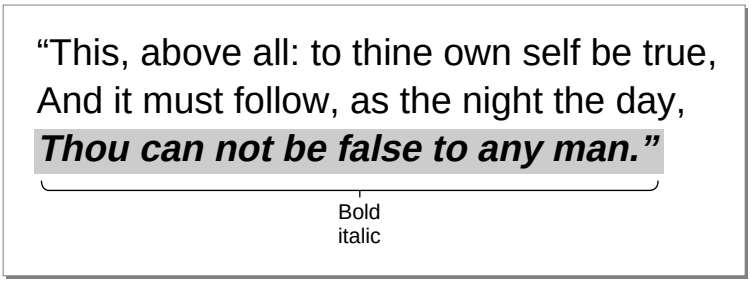
If you call `TESetStyle` with a `mode` of `doFace` + `doToggle` and a text style record parameter with its `tsFace` field set to `bold`, the resulting selection is no longer bold, as shown in Figure 2-12.

Figure 2-12 The result of calling `TESetStyle` to toggle to bold



On the other hand, if instead you call `TESetStyle` with a `mode` of `doFace` + `doToggle` and a text style record with its `tsFace` field set to `italic`, the resulting selection is all bold italic as shown in Figure 2-13.

Figure 2-13 The result of calling `TESetStyle` to toggle italics



Handling a Font Menu

You can use `TESetStyle` to handle a Font menu that allows the user to change the font family for a text selection. The user might select the entire text of an edit record or a portion of it, then choose a different font family from your menu to be used to render the text. Listing 2-10 shows how to handle a Font menu that allows the user to do this. The code determines which font the user has selected from the menu. Next, it calls the Font Manager’s `GetFNum` procedure to get the font family ID for the font of the selected text. Then it calls `TESetStyle` passing it the text style record with the `tsFont` field set to the font ID. Because the `redraw` parameter is set to `TRUE`, the current selection range is redrawn immediately in the new font.

Listing 2-10 Handling the Font menu

```
PROCEDURE MyHandleFontMenu (myWindow: WindowPtr; myTERec: TEHandle;
                            menuItem: Integer);
VAR
   txStyle:    TextStyle;  {holds style selected}
   fontName:   Str255;     {name of font selected}
   fontID:     Integer;    {ID of font selected}
BEGIN
   GetItem(GetMenuHandle(mFont), menuItem, fontName);
   GetFNum(fontName, fontID);
   txStyle.tsFont := fontID;
   TESetStyle(doFont, txStyle, true, myTERec);
   MyAdjustScrollBars(window, FALSE);
END;
```

Handling a Font Size Menu

If your application includes a menu that allows users to change the font size of the selected text, you can use the `TESetStyle` procedure to handle this modification. The code in Listing 2-11 sets the `tsSize` field of the text style record to the font size that the user selects; then it calls `TESetStyle` to apply the new font size immediately. The `doSize` mode parameter value forces all the text to the new size.

Listing 2-11 Handling the Size menu

```
PROCEDURE MyHandleSizeCommand (myTERec: TEHandle; menuItem: Integer);
VAR
   txStyle:    TextStyle;
BEGIN
   MyGetSize(GetMenuHandle(mSize), menuItem, sizeChosen);
   txStyle.tsSize := sizeChosen;
   TESetStyle(doSize, txStyle, TRUE, myTERec);
   MyAdjustScrollBars(window, FALSE);
END;
```

Handling a Style Menu

Your application can also use `TESetStyle` to handle Style menu commands. For example, you can set the `mode` parameter to `doFace` and set the `tsFace` field of the text style record to any of the font attributes that the user selects. If your menu supports a Plain option to remove all attributes from the text selection, you need to explicitly set `tsFace`. Because of the behavior of `TESetStyle`, you cannot implement a Plain selection by passing a null (empty set) text style record to remove the current attributes. Listing 2-12 shows how to use `TESetStyle` to change the text attributes, including how to render plain text.

Listing 2-12 Handling a Style menu

```
PROCEDURE MyHandleStyleMenu (myWindow: WindowPtr; myTERec: TEHandle;
                             menuItem: Integer);
VAR
   txStyle:    TextStyle;
   anIntPtr:   Integer;
BEGIN {mStyle}
   WITH txStyle DO BEGIN
      CASE menuItem OF
         plainItem:
            BEGIN
               anIntPtr := @txStyle.tsFace;
```

```
            anIntPtr^ := 0;
            tsFace := [];
          END;
        boldItem:
         tsFace := [bold];
        italicItem:
         tsFace := [italic];
        underlineItem:
         tsFace := [underline];
        outlineItem:
         tsFace := [outline];
        shadowItem:
         tsFace := [shadow];
    END; {case}

    IF menuItem <> 1 THEN
       TESetStyle(doFace + doToggle, txStyle, TRUE, myTERec)
          {if we don't select plain then use doToggle}
    ELSE
       TESetStyle(doFace, txStyle, TRUE, myTERec);
          {TESetStyle has problems with plain and doToggle }
          { has no effect!so we need to special case it.}
    MyAdjustScrollBars(window, FALSE);
  END;
END;
```

If you set `redraw` to `TRUE`, TextEdit redraws the current selection with the new attributes, recalculating line breaks, line heights, and font ascents. If you call `TESetStyle` with a value of `FALSE` for the `redraw` parameter, TextEdit does not redraw the text or recalculate line breaks, line heights, and font ascents until the next update event occurs. Consequently, when your application calls a routine that uses any of this information, such as `TEGetHeight` (which returns a total height between two specified lines), the routine uses the old character attribute information that existed before you called `TESetStyle` to change it. To be certain that the new information is always reflected immediately, call the `TESetStyle` procedure with a `redraw` parameter of `TRUE`.

Listing 2-13 shows a sample procedure that calls `TEContinuousStyle` to check the character attributes of the current selection range; it determines whether the style is plain, bold, or italic. For each style that is continuous across the text, the `MyAdjustStyleNew` procedure marks the item on the style menu. In this case, if `TEContinuousStyle` returns a `mode` parameter that contains `doFace` and the text style record `tsFace` field is bold, it means that the selected text is all bold, but may contain other text styles, such as italic, as well. Italic does not apply to all of the selected text, or it would have been included in the `tsFace` field. If the `tsFace` field is an empty set, then all of the selected text is plain.

Listing 2-13 Checking the style and marking Style menu items to reflect the current selection range

```
PROCEDURE MyAdjustStyleNew (myTERec: TEHandle);
VAR
styleMenu:  MenuHandle;
aStyle:     TextStyle;
mode:       Integer;
BEGIN
   mode := doFace;
   styleMenu := GetMenuHandle(mStyle);
   IF TEContinuousStyle(mode, aStyle, myTERec) THEN
      BEGIN
         {There is at least one style that is continuous over }
         { the selection. Note that it might be plain, which is }
         { actually the absence of all styles.}
         CheckItem(styleMenu, plainItem, aStyle.tsFace = []);
         CheckItem(styleMenu, boldItem, bold IN aStyle.tsFace);
         CheckItem(styleMenu, italicItem, italic IN aStyle.tsFace);
         {Set other menu items appropriately.}
   END

   ELSE
      BEGIN
         {No text face is common to the entire selection.}
         CheckItem(styleMenu, plainItem, FALSE);
         CheckItem(styleMenu, boldItem, FALSE);
         CheckItem(styleMenu, italicItem, FALSE);
         {Set other menu items appropriately.}
   END;
END;
```

Changing the Text Alignment

Your application can change the alignment of the entire text of an edit record by calling the `TESetAlignment` procedure. The default alignment used to display the text of an edit record is based on the primary line direction of the system script. For example, when the system script is Arabic or that of any language that is read from right to left, the default line direction is right to left and the text is right aligned.

For a script system whose primary line direction is right to left, you can force left alignment of the text by specifying `teFlushLeft` as the value of the `align` parameter, as shown in the following example:

```
TESetAlignment (teFlushLeft, myTERec);
```

You can use any of the following constants to specify how text is aligned.

Constant	Description
`teFlushDefault`	Default alignment according to the primary line direction
`teCenter`	Center for all scripts
`teFlushRight`	Right for all scripts
`teFlushLeft`	Left for all scripts

Make sure that you call the Window Manager's `InvalRect` procedure after you change the alignment so the text is redrawn with the new alignment. For more information about `InvalRect`, see the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Saving and Restoring a TextEdit Document, and Implementing Undo

This section describes how to save to disk the contents of a document created using TextEdit, and restore it when the user opens the document. For both monostyled and multistyled text, you need to save and restore the text and its character attribute information. This section also discusses how to implement an Undo feature.

Saving a TextEdit Document

To save the contents of a document created using TextEdit and a monostyled edit record, you store the text. You can also save the text characteristics, such as the font and its size and style, and the text margins; you can store this information in a resource. (Save the font name, not the font number.)

To save the contents of a document created using TextEdit and a multistyled edit record, you need to save all of the associated character attribute information in addition to the text. Because the text format of the character attribute information in the style scrap is easier to export than the style record itself—it uses the Desk Manager's `'styl'` format—you should use the TextEdit routines that use the style scrap for moving character attribute information: `TEGetStyleScrapHandle` and `TEUseStyleScrap`. For example, you can use the following steps to save a multistyled text document to disk:

1. Create a text file, select all the text of the edit record, and save it in the text file's data fork.
2. Call `TEGetStyleScrapHandle` to get a handle to the style scrap record. This creates the style scrap record and uses it to store the character attribute information.
3. Save the character attribute information in the resource fork of the file.

The application-defined procedure `MyDoSaveAsTextEdit` shown in Listing 2-14 uses this method. Notice that this procedure avoids using `TESetSelect` to select all of the edit record's text. The `TESetSelect` procedure sets and highlights the selection range that you specify. Because you are selecting the text to save it, you don't want it to be

highlighted. (Highlighting the text before saving it can mislead a user to presume that some other action is required.)

However, if you want to use `TESetSelect`, you can circumvent highlighting of the selection range if you first render the edit record inactive; before you call `TESetSelect`, call `TEDeactivate`. Also, if you have outline highlighting turned on through the `TEFeatureFlag` function's `teFOutlineHilite` feature, turn it off. When the edit record is not the active one, `TESetSelect` can set the selection range without causing it to be highlighted.

Listing 2-14 Saving a multistyled text edit record to disk

```
PROCEDURE MyDoSaveAsTextEdit(textToSave: TEHandle);
      CONST
      kFileType    = 'TEXT'; {file type of text file}
      kFileCreator = 'NIIM'; {creator code of text file}

VAR
      reply: StandardFileReply;
         {location, name of file to save text to}
      styles:    StScrpHandle; {contains all character }
                               { attributes in text}
      dataLength: LongInt; {number of bytes of text to write}
      dataRefNum: Integer; {ref number of text file's data fork}
      rsrcRefNum: Integer; {ref number of text file's rsrc fork}
      savedStart: Integer; {saves offset of start of selection}
      savedEnd:   Integer; {saves offset of end of selection}
      error:      OSErr;   {error code from toolbox}

BEGIN
   StandardPutFile( '', '', reply);
   IF reply.sfGood THEN
      BEGIN
      {save the current starting and ending offsets of selection}
         savedStart := textToSave^^.selStart;
         savedEnd := textToSave^^.selEnd;

      {select all text; don't use TESetSelect because it }
         { draws selection}
            textToSave^^.selStart := 0;
            textToSave^^.selEnd := textToSave^^.teLength;

      {get a list of all the attributes in the text}
            styles := TEGetStyleScrapHandle(textToSave);
```

```
      {reset the selection back to what it was}
            textToSave^^.selStart := savedStart;
            textToSave^^.selEnd := savedEnd;

      {create the text file if it didn't exist before}
            IF NOT reply.sfReplacing THEN
               BEGIN
                  error := FSpCreate(reply.sfFile,
                        kFileCreator, kFileType, reply.sfScript);
                  FSpCreateResFile(reply.sfFile, kFileCreator,
                        kFileType, reply.sfScript);
                  error := ResError;
               END;

         {open the text file}
         error := FSpOpenDF(reply.sfFile, fsCurPerm, dataRefNum);
         rsrcRefNum := FSpOpenResFile(reply.sfFile, fsCurPerm);
         error := ResError;

         {write the text to the file}
         dataLength := textToSave^^.teLength;
         error := FSWrite(dataRefNum, dataLength,
                         textToSave^^.hText^ );

         {Write the attributes to the file}
         AddResource(Handle(styles), 'styl', 0, '');
         WriteResource(Handle(styles));
         ReleaseResource(Handle(styles));

         {close the text file}
         error := FSClose(dataRefNum);
         CloseResFile(rsrcRefNum);
         error := ResError;
      END;
END;
```

Restoring an Existing TextEdit Document

You can restore the text of an edit record when a user opens a document that was created using TextEdit. One way to do this is to read the text from the data fork into a handle, then write the handle to the `hText` field of the edit record; call `TECalText` after you do this. Before you write the new handle to the `hText` field, dispose of the existing handle, if there is one. For a multistyled edit record, you need to reinstate both the text and the character attribute information for it. (For information about how to open a file, see *Inside Macintosh: Files*.)

You can use a method similar to the one shown in Listing 2-14 on page 2-53 to save a multistyled text document. However, to restore the text, you retrieve the data from the file's data fork and write it to a buffer, then call `TESetText` to make a copy of the text and set the `hText` field of the edit record to point to it. The `MyDoOpenTextEdit` procedure shown in Listing 2-15 shows an example of this. Before copying the text to a buffer, the `MyDoOpenTextEdit` procedure checks to ensure that the text length does not exceed the 32 KB limit; if it does, TextEdit truncates the text before it copies it.

The `MyDoOpenTextEdit` procedure retrieves the character attribute information from the resource fork of the disk file and reinstates it in the edit record's style record by calling `TEUseStyleScrap`.

Listing 2-15 Restoring a document that uses multistyled TextEdit

```
PROCEDURE MyDoOpenTextEdit(textToOpen: TEHandle);
   CONST
      kFileType = 'TEXT'; {file type of text file}

   VAR
      reply:      StandardFileReply; {location, name of file to get text from}
      typeList:   SFTypeList;    {specifies 'TEXT' files in SF dialog}
      dataRefNum: Integer;  {ref number of text file's data fork}
      rsrcRefNum: Integer;  {ref number of text file's rsrc fork}
      textBuffer: Handle;  {holds text from file}
      textLength: LongInt;  {number of bytes of text to read}
      styles:     StScrpHandle;  {contains all character attributes in text}
      error:      OSErr;      {error code from toolbox}
      savedState: SignedByte; {saves state of 'styl' resource}

BEGIN
   typeList[0] := kFileType;
   StandardGetFile(NIL, 1, typeList, reply);
   IF reply.sfGood THEN
      BEGIN
         {open the data fork of the text file}
         error := FSpOpenDF(reply.sfFile, fsCurPerm, dataRefNum);
         error := SetFPos(dataRefNum, fsFromStart, 0);
         {get the number of bytes of text in the file; limit to 32KB}
         error := GetEOF( dataRefNum, textLength );
         IF textLength > 32767 THEN
            textLength := 32767;
         {allocate a buffer for the text}
         textBuffer := NewHandle(textLength);
         {read the text into the buffer}
```

```
            error := FSRead( dataRefNum, textLength, textBuffer^ );
            {put the text into the TextEdit record}
            LockHHi(TextBuffer);
            TESetText(textBuffer^, textLength, textToOpen);
            HUnlock(textBuffer);
            {get rid of the text buffer}
            DisposeHandle(textBuffer);
            {close the data fork of the text file}
            error := FSClose(dataRefNum);
            {open the resource fork of the text file}
            rsrcRefNum := FSpOpenResFile(reply.sfFile, fsCurPerm);
            error := ResError;
            {get the style scrap}
            styles := StScrpHandle(GetResource('styl', 0));
            error := ResError;
            IF styles <> NIL THEN
               BEGIN
                  savedState := HGetState(Handle(styles));
                  {apply the character attributes to the TextEdit record}
                  TEUseStyleScrap(0, textLength, styles, true, textToOpen);
                  {restore state of 'styl' resource}
                  HSetState(Handle(styles), savedState);
               END;
            {close the forks of the text file}
            error := FSClose(dataRefNum);
            CloseResFile(rsrcRefNum);
            error := ResError;
         END;
END;
```

Handling Undo

Application users find Undo an especially useful feature. Users might accidently choose Clear from the Edit menu instead of Cut, or they might backspace over more words than intended. In these and cases like them, Undo is invaluable.

If you are implementing Undo for multistyled text, you need to save the character attribute information along with the text. Although this section discusses one method, there are a number of ways that you can do this. For example, when you want to save the current attributes of the selected text to allow the user to revert to them, your application calls the `TEGetStyleScrapHandle` function, which returns a handle to the style scrap's style record containing the attributes used for the selected text. To restore the style later, you call the `TEUseStyleScrap` procedure. You also need to save the offsets into the edit record's text buffer of the first and last characters to which the character attribute information is to be applied.

If your application supports any 2-byte script systems, your Undo operations needs to check for 2-byte characters. Normal cut or paste operations do not present a problem, but be careful when undoing a backspace. When TextEdit backspaces over single characters, it checks `CharByte` to determine if the character to be removed is a 2-byte character. If it is, it removes 2 bytes. (For more information about the `CharByte` function, see the chapter the "Script Manager," in this book.) When an application program maintains a buffer of characters that have been backspaced over in order to support Undo, it needs to make a test similar to that in Listing 2-16.

Listing 2-16 Checking for 2-byte characters when backspacing

```
IF myChar = BS then aTeHandle^^ do begin
   {support backspace undo}
   IF selStart <> selEnd then begin
   {not an insertion point save the selection}
   END
   ELSE begin
      i := selStart;
      IF i > 0 then begin
            repeat i := i - 1
            until CharByte(hText^, i) <= 0;
      {Note: Guarantees that CharByte(x,0) <= 0}
      {Also, CharByte does not touch the heap}
      {Put bytes from i to selStart into buffer}
      END;
   END;
END;
```

Customizing TextEdit

This section describes how to customize TextEdit using the `TECustomHook` routine to replace the end-of-line, drawing, width-measuring, and hit test default hook routines. It also describes the multi-purpose low-memory global variable `TEDoText` hook routine that displays, highlights, and hit-tests characters, and positions the pen to draw a caret. Finally, this section discusses how to customize word selection, automatic scrolling, and how to determine the length of a line of text in order to justify it. (For a brief discussion of hook fields and hook routines, see "Related Data Structures" on page 2-17.)

The next four sections describe how to customize TextEdit using the `TECustomHook` procedure. Information about the use of `TECustomHook` that is common to all four sections is provided here.

You can customize TextEdit's behavior by replacing any of the default hook routines with those of your own. You use the `TECustomHook` procedure to replace a routine installed in a hook field of the dispatch record (`TEDispatchRec`). Initially, each hook field of the dispatch record contains the address of the default hook routine that TextEdit uses.

The `TECustomHook` procedure returns the address of the default routine that it replaces so that your application-supplied routine can call the default routine, daisy-chaining it, if you want it to. For example, your routine can add additional functionality, then call the default routine instead of replicating all of its behavior. If you replace the address of a default hook routine with that of your own customized version, the next time you call `TECustomHook` for that hook field, `TECustomHook` will return the address of your routine. (For more information, see "TECustomHook" on page 2-112.) To ensure future compatibility, use the TextEdit customization routines to modify hooks rather than write directly to these fields.

If you replace a default hook routine with a customized version that you write in a high-level language, such as Pascal or C, you need to provide assembly-language glue code that utilizes the registers for your high-level language routine. Refer to "TECustomHook" on page 2-112 for a description of the register contents on entry and return for each of the hook routines.

If you replace a default routine, take the following precautions:

- Before placing the address of your routine in the TextEdit dispatch record, strip the addresses, using the Operating System Utilities `StripAddress` function, to guarantee that your application is 32-bit clean. For more information, see *Inside Macintosh: Operating System Utilities*.
- Before replacing a TextEdit routine with a customized one, determine whether more than one script system is installed, and, if so, ensure that your customized routine accommodates all of the installed script systems. This avoids the problem of your customized routine producing results that are incompatible with the Script Manager.
- When you use assembly language, note that all registers must be preserved except those specified as containing return values. Register A3 contains a pointer to the edit record and Register A4 contains a handle to it. You can obtain line start positions from the `lineStarts` array in the edit record. Register A5 is always valid. Refer to `TECustomHook` in the TextEdit Reference section for complete coverage of the register content requirements for all hook routines.

Replacing the End-of-Line Routine

You can replace the address of the default end-of-line hook routine with the address of your own routine that determines an end-of-line character if you want the end-of-line to be defined by a character other than the carriage return.

The default routine compares a given character with $0D (a carriage return) to determine whether it is an end-of-line character, and returns with the appropriate status flags (either `TRUE` or `FALSE`) in the status register.

Replacing the Drawing Routine

TextEdit calls the draw hook routine any time the various components of a line are drawn. The appropriate font, face, and size characteristics have already been set into the current graphics port by the time this routine is called.

If your application uses an outline font, the default behavior of the Font Manager ensure's that glyphs fit within the font's ascent and descent. Glyphs that extend beyond the ascent or descent, such as certain accented fonts, are scaled down to fit.

If your application has set the `preserveGlyph` parameter of the Font Manager's `SetPreserveGlyph` procedure to `TRUE` to preserve the original unscaled shape of the glyph, note that TextEdit sets it to `FALSE` before it calls the draw hook to perform any drawing. This is to guarantee that the glyphs whose bounding boxes exceed the font's ascent or descent are scaled down to prevent them from colliding with other glyphs on the lines above or below. TextEdit then restores the `preserveGlyph` parameter to its previous value before proceeding.

Replacing the Width-Measuring Routines

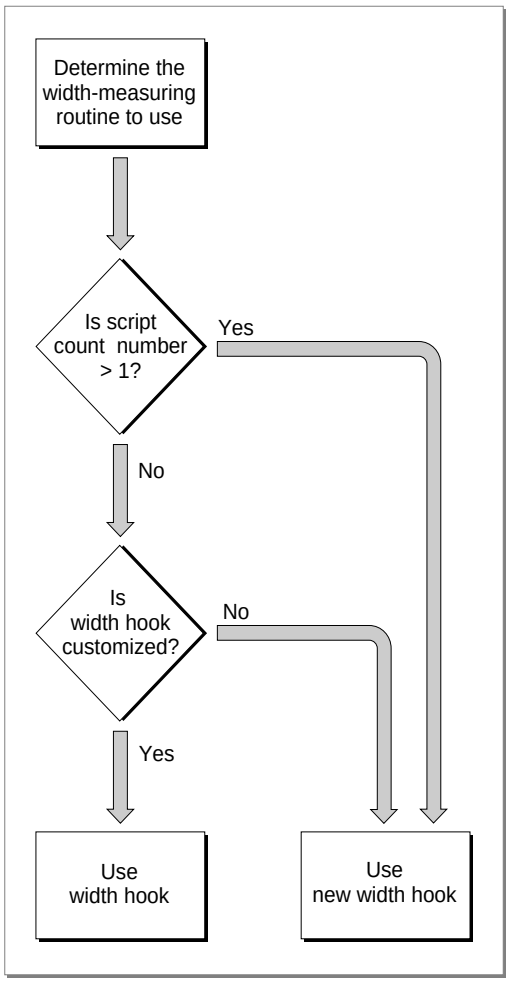
A width measurement hook routine measures portions of a line of text, and TextEdit calls one each time the width of various components of a line is calculated. There are three width measurement hooks: the width measurement hook, the new width measurement hook, and the text width measurement hook. Default hook routines of the same name as the hook field are installed in each of these hooks.

The width measurement hook, which TextEdit used in the past, now exists to provide backward compatibility for applications that have replaced the default routine with a customized one. TextEdit uses the routine whose address is installed in this field only when both of the following conditions exist: when only the Roman script system is installed and the field contains the address of a customized routine.

In all other cases—when more than one script system is installed or when the width measurement hook has not been customized—TextEdit calls the routine whose address is installed in the new width measurement hook field to measure text.

Figure 2-14 shows a flow chart illustrating when the width measurement hook and the new width measurement hook routines are used.

Figure 2-14 Determining when to use `WIDTHHook` and `nWIDTHHook`



The new width measurement hook routine is called to measure text for both Roman and non-Roman script systems. If you replace this routine, make sure that your customized routine is script-aware.

The default action for the new width measurement hook routine is to call the QuickDraw Manager's `CharToPixel` function or `TextWidth` procedure to measure for non-Roman scripts. By default, the `TextWidthHook` field contains the address of the QuickDraw `TextWidth` function. You can use this hook to replace TextEdit's use of the QuickDraw `TextWidth` function with your own measuring routine. If you replace this hook routine with a customized version, when the routine whose address is installed in the new width measurement hook field makes a call to `TextWidth`, your customized routine is invoked.

To test for the availability of the width-measuring hooks, you can call the `Gestalt` function with the `gestaltTextEditVersion` selector. A result of `gestaltTE2` or greater returned in the `response` parameter indicates that the new width measurement hook is available, and a result of `gestaltTE5` or greater indicates that `TextWidthHook` is available.

Replacing the Hit Test Routine

TextEdit calls the hit test hook routine to determine the glyph position in a line, given the pixel width from the left edge of the view rectangle. For versions of software earlier than 7.0, the default action is to call the `TextWidth` function to determine if the pixel width of the measured text is greater than the input width. If it is, then the hit test hook routine calls the QuickDraw `PixelToChar` function and returns. For system software version 7 and later, the default action is to call the QuickDraw `PixelToChar` function. In addition to the values defined by the register contents on entry, when TextEdit calls the `PixelToChar` function, it passes a value of `OnlyStyleRun` for the `styleRunPosition` parameter and scaling factors of 1/1 for the `numer` and `demon` parameters. See "Hit Test Hook Registers" on page 2-115.

Customizing Word Selection

A word-selection break routine determines which word is highlighted when the user double-clicks in the text. It also determines where TextEdit breaks the text at the end of a line. You can use `TESetWordBreak` to replace the default routine, installed in the edit record's `wordBreak` field, that is used for word selection and line breaking under certain circumstances. Whether or not TextEdit uses the word break hook routine installed in this field is determined by the algorithm implemented in the default `TEFindWord` routine, which is described below.

When you replace the `wordBreak` field hook routine, your customized word-selection break routine is used instead of the default one. The default routine breaks words at any character with an ASCII value of $20 or less (the space character or nonprinting control characters).

Before non-Roman script systems were supported, TextEdit used the word-selection break routine referenced by the `wordBreak` field for all word selection and line breaking. However, in order to support both Roman and non-Roman script systems, TextEdit now uses the routine referenced by the low-memory global variable `TEFindWord`. The default `TEFindWord` hook routine determines which hook TextEdit should use for word selection and line breaking—the `wordBreak` hook or the Text Utilities `FindWordBreaks` procedure—based on what script systems are installed and some other factors. You can replace the `TEFindWord` hook routine with a customized version.

The `TEFindWord` hook routine is a higher level routine than `wordBreak`. Because of this, when you customize the `TEFindWord` hook you are completely changing how TextEdit handles word selection and line breaking. However, when you replace `wordBreak`, you are only impacting those aspects of word selection and line breaking that are normally handled by the `wordBreak` routine.

The `TEFindWord` hook routine gives your application more control over the breaking process and allows you to write more efficient routines. However, unless you include explicit tests for scripts in your customized routine, the algorithms you provide may be incorrect for non-Roman scripts. If you replace `TEFindWord`, you should understand the behavior of the default routine.

Here's how the default `TEFindWord` routine works:

- TextEdit initially determines whether a non-Roman script system is installed. If more than the Roman script system is installed, TextEdit always uses the Text Utilities `FindWordBreaks` procedure for line breaking and word selection.
- When TextEdit determines that only the Roman script system is installed and the `TEFindWord` routine is being called for line breaking (not word selection), TextEdit calls the `wordBreak` hook.
- If `TEFindWord` is called for word selection for system software with only the Roman script system installed, TextEdit checks to see if your application has placed the address of a customized word-selection breaks routine in the `wordBreak` field of the edit record. If so, TextEdit calls your word-selection breaks routine. Otherwise, if the `wordBreak` field contains the address of TextEdit's internal word-selection breaks routine, TextEdit uses the Text Utilities `FindWordBreaks` procedure to determine word-selection breaks.

When TextEdit calls the Text Utilities `FindWordBreaks` procedure, it uses information in the edit record to provide the necessary parameters. TextEdit determines the current script boundaries from the Text Utilities `FindWordBreaks` procedure by using the font run information in the style record (of type `TEStyleRec`). TextEdit also determines the length of the script run and the offset within the script run from which to begin searching for a word boundary. TextEdit uses the value in the `clikStuff` field of the edit record to determine the leading edge flag for the `FindWordBreaks` procedure. You must use similar information to replace `TEFindWord` correctly for non-Roman scripts.

Customizing Automatic Scrolling

Scroll bars associated with the text are not automatically scrolled with the text unless you replace the address of the default click loop routine with that of a customized routine that updates the scroll bars. You can write your own click loop routine that includes code to update the scroll bars along with the text and install its address in the `clikLoop` field. To replace the default click loop routine with your customized version, you call the `TESetClickLoop` procedure.

You can write a routine that manages the scroll bars, then calls the default click loop routine, rather than replicating its behavior in your routine. However, if your routine scrolls the text and updates scroll bars, you should consider what the default click loop routine does. It adjusts the value in the `clickTime` field of the edit record to allow for slower scrolling.

When `TEClick` is called, the `clickTime` field contains the time when `TEClick` was last called. TextEdit sets the `clickTime` field with the current tick count on exit from the `TEClick` procedure and uses the new value at reentry the next time `TEClick` is called.

If you code a click loop routine in Pascal, it should have no parameters and it should return a Boolean value. You can declare a click loop routine named `MyClickLoop` like this:

```
FUNCTION MyClickLoop: Boolean;
```

The function should return `TRUE`. Returning `FALSE` from your click loop routine tells the `TEClick` procedure that the mouse button has been released, which aborts `TEClick`.

Installing a customized default click loop routine

If you code a click loop routine in Pascal, then call the `TESetClickLoop` procedure to install the Pascal routine in the `clikLoop` field, `TESetClickLoop` installs a glue code routine in the `clikLoop` field because `clikLoop` expects a routine that uses assembly-language conventions. Because of this, you must always use `TESetClickLoop` to install a Pascal routine, while you must always directly install an assembly routine in the `clikLoop` field. ◆

If you code a click loop routine in assembly, it should set register D0 to 1 and preserve register D2. Returning 0 in register D0 aborts `TEClick`.

You can write a routine that manages the scroll bars, then calls the default click loop routine, rather than replicating its behavior in your routine. If your customized routine calls the default click loop routine, it must use assembly-language calling convention.

Determining the Line Length

This section describes how to determine the length of a line. You can use this information, for example, to justify a line of text; although TextEdit aligns text with the right or left margins, or centers it, it does not justify it.

To determine the length of a line, you use the information contained in the edit record's line starts array and `nLine`s field. The line starts array is a variable-length field in the edit record that contains the byte offset for the first character of each line. This array has the following boundary conditions:

- The first entry has index 0 and value 0.
- The last entry in the array has index `nLines` and value `teLength` (therefore, there are `nLines` + 1 entries).
- The beginning of the first line is given by `lineStarts`[0], and the beginning of the second line is given by `lineStarts`[1]; therefore, the length of the first line is given by `lineStarts`[1] – `lineStarts`[0].
- The maximum number of entries is 16,000.

For example, if you want to determine the length of the line *n* (where *n* = 0 for the first line), subtract its start location (contained in the array entry with index *n*) from its end location (contained in the array with index *n* + 1):

```
lengthOfLineN := myTE^^.lineStarts[n+1] – myTE^^.lineStarts[n];
```

The terminating condition for this measurement is when *n* is equal to `nLines` plus 1.

IMPORTANT

Do not change the information contained in the `lineStarts` array. ▲

Advanced Customization

The low-memory global variable `TEDoText` is a hook which contains the address of a multi-purpose text editing routine that advanced programmers may find useful. It lets you display, highlight, and hit-test characters, and position the pen to draw the caret. Hit-testing is the process of determining where to place the insertion point when the user clicks the mouse button; the point selected with the mouse is in the `SelPoint` field. The registers contain the following values.

Registers on entry

A3 Pointer to the locked edit record

D3 Position of the first character (word)

D4 Position of the last character; used as defined below (word)

D7 Selectors for `TEDoText` (word)

```
teFind          EQU     0       to hit-test the character specified in D3
teHighlight     EQU     1       to highlight the text range specified in D3
                                and D4
teDraw          EQU     -1      to display the range of text specified in D3
                                and D4
teCaret         EQU     -2      to draw the caret at the position specified
                                in D3
teFind          EQU     0       to hit-test the character specified in D3
```

Registers on exit

A0 Pointer to current graphics port

D0 If hit-testing, byte offset where hit, or –1 for none (word)

Note

You need to use the value stored in the edit record `selPoint` field for hit-testing if you replace the routine pointed to by the global variable `TEDoText`. (The assembly-language offset for this field is named `teSelPoint`.) ◆

TextEdit Reference

This section describes the data structures and routines that comprise TextEdit. The "Data Structures" section shows the Pascal data structures including the edit record and subsidiary structures that allow for text styling and customization of TextEdit. Together with the TextEdit private scrap and the TextEdit style scrap, these data structures define the TextEdit environment.

The "Routines" section describes the routines that provide applications with the means of creating edit records and accessing, editing, and displaying multistyled and monostyled text, including text highlighting and scrolling.

The constants that define values for some of the parameters used in several of these routines are listed in the "Summary of TextEdit" on page 2-120.

Data Structures

This section describes the data structures and their contents which provide information to the TextEdit routines. Both monostyled and multistyled edit records have a 32 KB maximum text size.

The TextEdit data structures are defined as follows:

- The edit record, defined by the `TERec` data type, stores the display and editing information for TextEdit.
- Along with various subsidiary data structures, the style record, defined by the `TEStyleRec` data type, stores the character attribute information for the text of the edit record.
- The style run table, defined by the `StyleRun` data type, is an array that contains the boundaries of each style run and an index to its character attribute information in the style element array.
- The style table, defined by the `TEStyleTable` data type, contains one entry for each distinct set of character attributes used in the text of the edit record.
- The line-height table, defined by the `LHTable` data type, provides an array of line heights to hold the vertical spacing information for a given edit record. It also contains line ascent information.

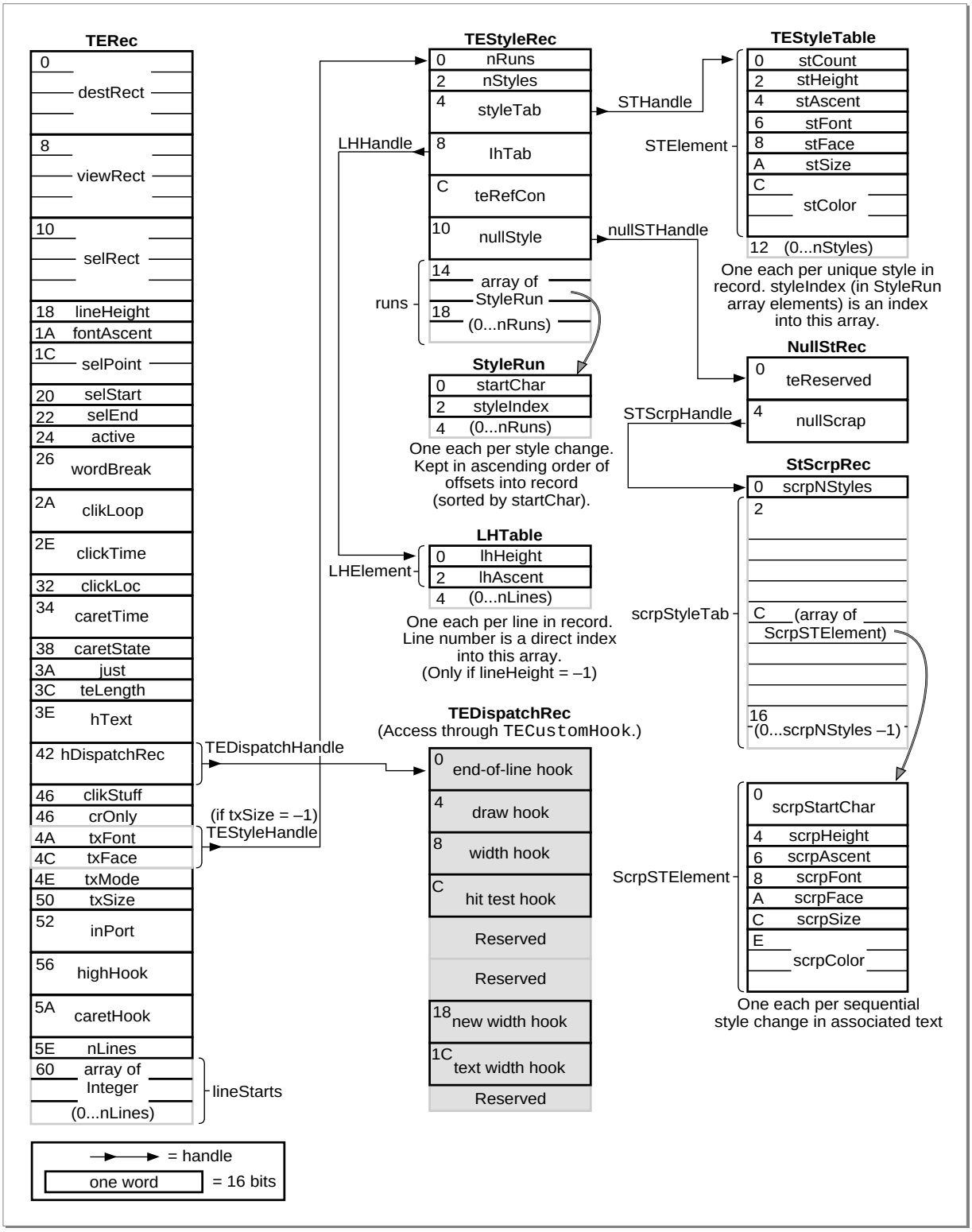
- The null style record, defined by the `NullStRec` data type, contains the null scrap which is used to store character attribute information for a null selection.
- The style scrap record, defined by the `StScrpRec` data type, is used by routines to store character attribute information temporarily.
- The scrap style table, defined by the `scrpStyleTab` data type, is contained in the style scrap record.
- The scrap style element record, defined by the `ScrpSTElement` data type, contains the character attribute information for an element in the scrap style table. One scrap style element record exists for each sequential attribute change in the associated text.
- The TextEdit dispatch record, defined by the `TEDispatchRec` data type, contains the internal addresses of the TextEdit routines for the end-of-line hook, the draw hook, the width measurement hook, the new width measurement hook, and the text width measurement hook, unless you replace them with the addresses of your own customized versions of these routines.
- The text style record, defined by the `TextStyle` data type, is used by several routines to pass character attribute information between the application and a routine. The record is passed as a variable or reference parameter.

Figure 2-15 shows the TextEdit data structures and their fields to help you understand how the TextEdit data structures are organized and related. (For a monostyled edit record, TextEdit creates only the `TERec` and `TEDispatchRec` data structures.) To read from and write to these data structures, use the TextEdit routines rather than modifying these fields directly. This practice ensures future compatibility.

For most operations, you do not need to know the exact structure of an edit record; TextEdit routines gain access to the record for you. However, when manipulating character attribute information, you might find it helpful to understand how the data structures used to contain and track character attribute information are organized.

Note
The space beyond the hooks in the TextEdit dispatch record is reserved for internal use. If you attempt to use this private area, you may corrupt TextEdit data. ◆

Figure 2-15 The TextEdit data structures and fields


TERec
0
destRect
8
viewRect
10
selRect
18 lineHeight
1A fontAscent
1C selPoint
20 selStart
22 selEnd
24 active
26 wordBreak
2A clikLoop
2E clickTime
32 clickLoc
34 caretTime
38 caretState
3A just
3C teLength
3E hText
42 hDispatchRec
46 clikStuff
46 crOnly
4A txFont
4C txFace
4E txMode
50 txSize
52 inPort
56 highHook
5A caretHook
5E nLines
60 array of Integer
(0...nLines)
lineStarts
TEDispatchHandle
(if txSize = –1)
TEStyleHandle
TEStyleRec
0 nRuns
2 nStyles
4 styleTab
8 lhTab
C teRefCon
10 nullStyle
14 array of StyleRun
18 (0...nRuns)
runs
LHHandle
STHandle
nullSTHandle
StyleRun
0 startChar
2 styleIndex
4 (0...nRuns)
One each per style change. Kept in ascending order of offsets into record (sorted by startChar).
LHTable
0 lhHeight
2 lhAscent
4 (0...nLines)
LHElement
One each per line in record. Line number is a direct index into this array. (Only if lineHeight = –1)
TEDispatchRec
(Access through TECustomHook.)
0 end-of-line hook
4 draw hook
8 width hook
C hit test hook
Reserved
Reserved
18 new width hook
1C text width hook
Reserved
TEStyleTable
0 stCount
2 stHeight
4 stAscent
6 stFont
8 stFace
A stSize
C stColor
12 (0...nStyles)
STElement
One each per unique style in record. styleIndex (in StyleRun array elements) is an index into this array.
NullStRec
0 teReserved
4 nullScrap
STScrpHandle
StScrpRec
0 scrpNStyles
2
C (array of ScrpSTElement)
16 (0...scrpNStyles –1)
scrpStyleTab
0 scrpStartChar
4 scrpHeight
6 scrpAscent
8 scrpFont
A scrpFace
C scrpSize
E scrpColor
ScrpSTElement
One each per sequential style change in associated text
= handle
one word
= 16 bits

The Edit Record

The edit record contains display, storage, styling, and other information related to editing that TextEdit requires. Although some fields are used differently for multistyled edit records and monostyled edit records, the structure of an edit record is the same whether the text is multistyled or monostyled.

```
TYPE  TERec =
      RECORD
         destRect:   Rect;    {destination rectangle}
         viewRect:   Rect;    {view rectangle}
         selRect:    Rect;    {the selection rectangle}
         lineHeight: Integer; {used for vertical spacing of lines}
         fontAscent: Integer; {used for caret/highlighting }
                              { position}
         selPoint:   Point;   {point selected with the mouse}
         selStart:   Integer; {start of selection range}
         selEnd:     Integer; {end of selection range}
         active:     Integer; {set when record is activated or }
                              { deactivated}
         wordBreak:  ProcPtr; {word break hook}
         clikLoop:   ProcPtr; {click loop hook}
         clickTime:  LongInt; {used internally}
         clickLoc:   Integer; {used internally}
         caretTime:  LongInt; {used internally}
         caretState: Integer; {used internally}
         just:       Integer; {alignment of text}
         teLength:   Integer; {length of text}
         hText:      Handle;  {handle to text to be edited}
         hDispatchRec: Handle;{handle to TextEdit dispatch record}
         clikStuff:  Integer; {used internally}
         crOnly:     Integer; {if <0, new line at Return only}
         txFont:     Integer; {text font.Otherwise, if txSize is }
                              { -1, combines with txFace to hold }
                              { a handle to the style record.}
         txFace:     Style;   {character style; unpacked byte. }
                              { Otherwise, if txSize is -1, }
                              { combines with txFont to hold a }
                              { handle to the style record}
         txMode:     Integer; {pen mode}
         txSize:     Integer; {tells if multistyled }
                              { edit record; if not, font size}
         inPort:     GrafPtr; {a pointer to the graphics port }
                              { for this TERec}
```

```
        highHook:   ProcPtr; {used for text highlighting}
        caretHook:  ProcPtr; {used for caret appearance}
        nLines:     Integer; {number of lines}
        lineStarts: ARRAY[0..16000] OF Integer;
                             {positions of line starts}
   END;

TYPE  TEPtr    = ^TERec;
      TEHandle = ^TEPtr
```

Field descriptions

`destRect` The destination rectangle, in local coordinates.

`viewRect` The view rectangle, in local coordinates.

`selRect` The selection rectangle, whose boundaries are defined in local coordinates. This value is the current selection range or insertion point.

`lineHeight` The vertical spacing of lines of text. Vertical spacing may be fixed or it may vary from line to line, depending upon specific text attributes. If the value of `lineHeight` is greater than 0, this field specifies the fixed vertical distance from the ascent line of one line of text down to the ascent line of the next.

If the value of `lineHeight` is less than 1, then this field specifies the vertical distance from the ascent line of one line of text down to the ascent line of the next calculated independently for each line, based on the maximum value for any individual character attribute on that line.

`fontAscent` The font ascent line. If the value of `fontAscent` is greater than 0, this field specifies how far above the base line the pen is positioned to begin drawing the caret or highlighting.

For single-spaced text, this is the height of the text in pixels (the height of the tallest characters in the font from the base line). If the value of `fontAscent` is less than 1, this field specifies the font ascent calculated independently for each line, based on maximum value for any individual character attribute on that line.

`selPoint` The point selected with the mouse, in the local coordinates of the current graphics port. The assembly-language offset for this field is named `teSelPoint`.

`selStart` The byte offset of the beginning of a selection range. Note that byte offset 0 refers to the first byte in the text buffer.

`selEnd` The byte offset of the end of a selection range. To include that byte, this value must be 1 greater than the position of the last byte offset of the text.

`active` This field is used internally by TextEdit. It is set when an edit record is activated through `TEActivate` and then reset when the edit record is rendered inactive through `TEDeactivate`. To ensure future compatibility, use `TEActivate` or `TEDeactivate` to access this field.

`wordBreak`	The record's word selection break routine. This routine determines the word that is highlighted when the user double-clicks in the text and the position at which text is wrapped at the end of a line.
`clikLoop`	The pointer to the click loop routine.The specified click loop routine is called repeatedly by the `TEClick` procedure as long as the mouse button is held down within the text.
`clickTime`	This field is for internal use only.
`clickLoc`	This field is for internal use only.
`caretTime`	This field is for internal use only.
`caretState`	This field is for internal use only.
`just`	The type of text alignment: default (according to primary line direction), left, center, or right.
`teLength`	The number of bytes in the text to be edited. For two-byte systems, potentially twice the number of characters. Initially set to zero. The maximum length is 32767 bytes.
`hText`	A handle to the text. Initially, it points to a zero-length block of text in the heap.
`hDispatchRec`	The handle to the TextEdit dispatch record. This field is for internal use only; do not modify this field, or copy it to another edit record. Each edit record has its own dispatch record. Attempting to use the dispatch record of one edit record with another edit record can cause TextEdit to crash.
`clikStuff`	This field is for internal use only. TextEdit sets this field to reflect whether the most recent mouse-down event occurred on the leading or trailing edge of a glyph. TextEdit uses this value in determining a caret position.
`crOnly`	A value specifying whether or not text wraps at the right edge of the destination rectangle. If `crOnly` is positive, text does wrap. If `crOnly` is negative, new lines are specified explicitly by `Return` characters only; text does not wrap at the edge of the destination rectangle. (This is useful in an application similar to a programming-language editor, where you may not want a single line of code to be split onto two lines.)
`txFont`	The font of all the text in the edit record if the `txSize` field of this edit record ≥ 0. If you change this value, the entire text of this edit record has the new characteristic when it is redrawn; also, remember to change the `lineHeight` and `fontAscent` fields as well. If the `txSize` field is –1, this field combines with `txFace` to hold a handle to the associated style record.
`txFace`	The character attributes of all the text in an edit record if the `txSize` field of this edit record ≥ 0. If you change this value, the entire text of this edit record has the new characteristic when it is redrawn; also, remember to change the `lineHeight` and `fontAscent` fields as well. If the `txSize` field is –1, this field combines with `txFont` to hold a handle to the associated style record.

`txMode`	The pen mode of all the text in the edit record. If you change this value, the entire text of this edit record has the new characteristic when it is redrawn; also, remember to change the `lineHeight` and `fontAscent` fields as well.
`txSize`	Depending on its value, `txSize` either contains the point size of all of the text or it acts as a flag indicating whether or not there is associated character attribute information. If `txSize` ≥ 0, this is a monostyled edit record, that is, all text is set in a single font, size, and face, and the value of `txSize` is the size of the text. If `txSize` is –1, the edit record contains associated character attribute information and the `txFont` and `txFace` fields combine to form a handle to the style record.
`inPort`	A pointer to the graphics port associated with this edit record.
`highHook`	A pointer to the routine that deals with text highlighting. In assembly language, the `highHook` field is located at the offset `teHiHook`. For more information, see the following section, "The High Hook and Caret Hook Fields."
`caretHook`	A pointer to the routine that controls the appearance of the caret. In assembly language, the `caretHook` field is located at the offset `teCarHook`. For more information, see the following section, "The High Hook and Caret Hook Fields."
`nLines`	The number of lines in the text.
`lineStarts`	An array containing the character position of the first character in each line. It is declared to have 16001 elements to comply with Pascal range checking. This is a dynamic data structure having only as many elements as needed. TextEdit calculates these values internally, so do not change the elements of the `lineStarts` array. Because this data structure grows and shrinks, the size of the edit record changes.

The High Hook and Caret Hook Fields

The `highHook` and `caretHook` fields—at the offsets `teHiHook` and `teCarHook` in assembly language—contain the addresses of routines that deal with text highlighting and the caret. These routines pass parameters in registers; if you replace these routines, your application must save and restore the registers' contents.

If you store the address of a routine in `teHiHook`, that routine is used instead of the QuickDraw procedure `InvertRect`, which is called by default, whenever a selection range is to be highlighted. Your routine can destroy the contents of registers A0, A1, D0, D1, and D2. On entry, A3 is a pointer to a locked edit record; the stack contains the rectangle enclosing the text being highlighted. (Use of the A3 register is equivalent to the `InvertRect r` parameter of type `RECT`. See the QuickDraw chapters in *Inside Macintosh: Imaging* for more information about the `InvertRect` procedure.) For example, if you store the address of the following routine in `teHiHook`, selection range is underlined instead of inverted.

```
UnderHigh
   MOVE.L   4(SP),A0            ;get address of rectangle to be
                                ;highlighted
   MOVE     bottom(A0),top(A0)  ;make the top coordinate equal to
   SUBQ     #1,top(A0)          ;the bottom coordinate minus 1
   _InverRect                   ;invert the resulting rectangle
   RTS
```

The routine whose address is stored in `teCarHook` acts exactly the same way as the `teHiHook` routine, but on the caret instead of the selection range, allowing you to change the appearance of the caret. The routine is called with the stack containing the rectangle address that encloses the caret.

The Style Record

The style record stores the character attribute information for the text of a multistyled edit record. If an edit record has associated character attribute information, its `txFont` and `txFace` fields combine to hold a style handle, of type `TEStyleHandle`, to its style record. The text is divided into style runs, summarized in the style run table, of type `StyleRun`, which is part of the style record. Each entry in the style run table gives the starting character position of a run and an index into the style table, of type `TEStyleTable`.

The style table element pointed to by the style run index describes the character attributes for that run.

To determine the length of a run, you subtract its start position from that of the next entry in the style run table. A dummy entry at the end of the style run table delimits the length of the last run; its start position is equal to the overall number of characters in the text, plus 1. The `TEStyleRec` data type defines the style record.

```
TYPE  TEStyleRec =
      RECORD
         nRuns:      Integer;       {number of style runs}
         nStyles:    Integer;       {size of style table}
         styleTab:   STHandle;      {handle to style table}
         lhTab:      LHHandle;      {handle to line-height table}
         teRefCon:   LongInt;       {reserved for application use}
         nullStyle:  NullStHandle;  {handle to style set at }
                                    { null selection}
         runs:       ARRAY [0..8000] OF StyleRun;
      END;

TEStylePtr = ^TEStyleRec;
TEStyleHandle = ^TEStylePtr;
```

```
StyleRun = RECORD
   startChar:  Integer;       {starting character position}
   styleIndex: Integer;       {index in style table}
END;
```

Field descriptions

`nRuns`	The number of style runs in the text.
`nStyles`	The number of distinct sets of character attributes used in the text; this forms the size of the style table.
`styleTab`	A handle to the style table.
`lhTab`	A handle to the line height table.
`teRefCon`	A reference constant for use by applications. The application can use this 32-bit field to suit its needs.
`nullStyle`	A handle to the style scrap record used to store the character attribute information for a null selection.
`runs`	A table of style runs that is of indefinite length.

```
TEStylePtr = ^TEStyleRec;
TEStyleHandle = ^TEStylePtr;

StyleRun = RECORD
   startChar:  Integer;       {starting character position}
   styleIndex: Integer;       {index in style table}
END;
```

The Style Table

The style table contains one entry for each distinct set of character attributes used in the text of an edit record. Each entry is defined in a style element record. The size of the table is given by the `nStyles` field of the style record. There is no duplication; each set of character attributes appears exactly once in the table. A reference count tells how many times each set of attributes is used in the table. The `TEStyleTable` data type defines the style table. The `STElement` data type defines the style element record.

```
TYPE  STElement =
      RECORD
         stCount:    Integer;    {number of runs in this style}
         stHeight:   Integer;    {line height}
         stAscent:   Integer;    {font ascent}
         stFont:     Integer;    {font family ID}
         stFace:     Style;      {character style}
         stSize:     Integer;    {size in points}
         stColor:    RGBColor;   {absolute RGB color}
      END;
```

```
STHandle = ^STPtr;
STPtr    = ^TEStyleTable;

TEStyleTable = ARRAY [0..1776] OF STElement;
```

Field descriptions

`stCount`	A reference count of character runs using this set of character attributes.
`stHeight`	The line height for this run, in points.
`stAscent`	The font ascent for this run, in points.
`stFont`	The font family ID.
`stFace`	The character style (bold, italic, and so forth). This field consists of two bytes. The low-order byte contains the character style. TextEdit uses the high bit (bit 15) of the high-order byte to store the style run direction: it uses 0 for left-to-right text, and 1 for right-to-left text.
`stSize`	The text size, in points.
`stColor`	The RGB (red, green, blue) color.

The Line Height Table

The line height table holds vertical spacing information for the text of an edit record. This table parallels the `lineStarts` array in the edit record itself. Its length equals the edit record's `nLines` field plus 1 for a dummy entry at the end, just as the `lineStarts` array ends with a dummy entry that has the same value as the length of the text. The table's contents are recalculated whenever the line starting values are themselves recalculated with the `TECalText` routine or whenever an editing action causes recalibration.

The line height table is used only if the `lineHeight` and `fontAscent` fields in the edit record are negative; positive values in those fields specify fixed vertical spacing, overriding the information in the table. The line height table is of type `LHTable`, which is an array of elements of `LHElement`.

```
TYPE LHElement =
   RECORD
      lhHeight:      Integer;    {maximum height in line}
      lhAscent:      Integer;    {maximum ascent in line}
   END;

LHPtr = ^LHTable;
LHHandle = ^LHPtr;

LHTable = ARRAY [0..8000] OF LHElement;
```

Field descriptions

`lhHeight`	The line height in points. This is the maximum value for any individual character attribute in the line.
`lhAscent`	The font ascent in points; this is the maximum value for any individual character attribute in a line.

The Null Style Record

The null style record contains the null scrap, which is used to store the character attribute information for a null selection (insertion point). A number of routines either write this character attribute information to the null scrap or read it from this scrap (to be applied to inserted text). The null scrap is created and initialized when an application calls `TEStyleNew` to create a multistyled edit record. The null scrap is retained for the life of the edit record; it is destroyed when `TEDispose` destroys the edit record and releases the memory allocated for it.

The `NullSTRec` data type defines the null style record.

```
TYPE NullStRec =
   RECORD
      teReserved:    LongInt; {reserved for future expansion}
      nullScrap:     StScrpHandle;  {handle to the style scrap }
                                    { record}
   END;

NullStPtr = ^NullStRec;
NullStHandle = ^NullStPtr;
```

Field descriptions

`teReserved`	This field is reserved for future expansion.
`nullScrap`	A handle to the style scrap record.

The Style Scrap Record

The style scrap is used for storing character attribute information associated with the current text selection or insertion point, character attribute information to be applied to text, or multistyled text that is cut or copied. When multistyled text is cut or copied, the character attribute information is written to both the style scrap and the desk scrap.

In most cases, the style scrap is created dynamically as needed by routines. However, a style scrap record can be created directly without using the `TEGetStyleScrapHandle` function; the character attribute information written to it can be applied to inserted text through `TEStyleInsert` or to existing text through `TEUseStyleScrap`.

The format of the style scrap is defined by a style scrap record of type `STScrpRec`.

```
TYPE  StScrpRec =
      RECORD
         scrpNStyles:   Integer;  {number of sets of }
                                  { character attributes in scrap}
         scrpStyleTab:  ScrpSTTable; {table of attributes for }
                                     { scrap}
      END;

StScrpPtr = ^StScrpRec;
StScrpHandle = ^StScrpPtr;
```

Field descriptions

`scrpNStyles` The number of style runs used in the text. This determines the size of the style table. When character attribute information is written to the null scrap, this field is set to 1; when the character attribute information is removed, this field is set to 0.

`scrpStyleTab` The scrap style table containing an element for each style run.

The Scrap Style Table

The style scrap record contains the scrap style table. Unlike the main style table for an edit record, the scrap style table may contain duplicate elements; the entries in the table correspond one-to-one with the style runs in the text. The `scrpStartChar` field of each entry gives the starting position for the run.

The `scrpStyleTab` data type defines the scrap style table data structure, which is an array of scrap style element records. The `ScrpSTElement` data type defines each scrap style element record.

```
TYPE  ScrpSTElement =
      RECORD
         scrpStartChar: LongInt;    {offset to start of style}
         scrpHeight:    Integer;    {line height}
         scrpAscent:    Integer;    {font ascent}
         scrpFont:      Integer;    {font family ID }
         scrpFace:      Style;      {character style}
         scrpSize:      Integer;    {size in points}
         scrpColor:     RGBColor;   {absolute (RGB) color}
      END;

ScrpSTTable = ARRAY[0..1600] OF ScrpSTElement;
```

Field descriptions

`scrpStartChar` The offset to the beginning of a style record in the scrap.

`scrpHeight`	The line height. You can determine the line height and the font ascent using the QuickDraw routine `GetFontInfo` described in the chapter "QuickDraw Text" in this book.
`scrpAscent`	The font ascent. See `scrpHeight`.
`scrpFont`	The font family ID.
`scrpFace`	The style (such as plain, bold, underline).
`scrpSize`	The size in points.
`scrpColor`	The RGB (red, green, blue) color for the style scrap.

Text Style Record

Text style records are used for communicating character attribute information between the application and several TextEdit routines, such as `TEContinuousStyle` and `TEReplaceStyle`. They carry the same information as the style element records in the style table, but without the reference count, line height, and font ascent.

The `TextStyle` data type defines a text style record.

```
TYPE  TextStyle =
      RECORD
         tsFont:  Integer;    {font family number}
         tsFace:  Style;      {character style}
         tsSize:  Integer;    {size in points}
         tsColor: RGBColor;   {absolute RGB color}
      END;

TextStylePtr = ^TextStyle;
TextStyleHandle = ^TextStylePtr;
```

Field descriptions

`tsFont`	The font family number.
`tsFace`	The character style (bold, italic, plain, and so forth).
`tsSize`	The text size in points.
`tsColor`	The RGB (red, green, blue) color.

Routines

This section describes the TextEdit routines that an application can call to

- initialize TextEdit and create an edit record
- activate and deactivate an edit record
- set and get the text and character attribute information of an edit record

- set the caret and selection range
- display and scroll text
- modify the text of an edit record
- manage the TextEdit private scrap
- check, set, and replace character attributes
- use byte offsets and corresponding points
- toggle automatic scrolling, outline highlighting, and text buffering on and off
- customize TextEdit

Each routine description defines a Pascal interface, provides related assembly-language information, and lists possible result codes, if any are returned.

Initializing TextEdit, Creating an Edit Record, and Disposing of an Edit Record

Preparation of a window for text editing involves setting up TextEdit's internal data structures by calling the `TEInit` procedure and creating an edit record for the window with the `TEStyleNew` function or the `TENew` function.

The `TEStyleNew` function creates a new multistyled edit record. A multistyled edit record contains text whose attributes, including font, size, and style, can vary from character to character. The `TENew` function creates a new monostyled edit record. A monostyled edit record contains text that is set in a single font, size, and style. Before either of these functions is called, the window must be in the current graphics port.

The `TEDispose` procedure destroys an edit record and releases the memory used for it. For a complete description of the edit record and its fields, see "An Overview of the TextEdit Data Structures" on page 2-16 and "Data Structures" on page 2-65.

TEInit

The `TEInit` procedure initializes TextEdit.

```
PROCEDURE TEInit;
```

DESCRIPTION

In addition to initialization of miscellaneous global variables, such as `TEDoText` and `TERecal`, the `TEInit` procedure sets up the private scrap and allocates a handle to it. Call `TEInit` at the beginning of your program after you initialize QuickDraw, the Font Manager, and the Window Manager, in that order, and before you initialize the Dialog Manager. You should call `TEInit` even if your application doesn't use TextEdit, so that desk accessories and dialog and alert boxes, which use TextEdit routines, work correctly.

TEStyleNew

The `TEStyleNew` function creates a multistyled edit record and allocates a handle to it.

```
FUNCTION TEStyleNew (destRect: Rect; viewRect: Rect): TEHandle;
```

`destRect` The destination rectangle for the new edit record, specified in the local coordinates of the current graphics port. This is the area in which text is laid out.

`viewRect` The view rectangle for the new edit record, specified in the local coordinates of the current graphics port. This is the area of the window in which text is actually displayed.

DESCRIPTION

Always use the `TEStyleNew` function to create an edit record for text that uses varying character attributes. The `TEStyleNew` function sets the `txSize`, `lineHeight`, and `fontAscent` fields of the edit record to –1, allocates a style record, and stores a handle to the style record in the `txFont` and `txFace` fields. The `TEStyleNew` function creates and initializes a null scrap that is used by TextEdit routines throughout the life of the edit record.

Call `TEStyleNew` once for every edit record you want allocated. Your application needs to store the handle to the edit record that is returned; many routines require it as an input parameter.

If your application contains more than one window where text editing occurs, you need to create an edit record for each window.

TENew

The `TENew` function creates and initializes a monostyled edit record and allocates a handle to it.

```
FUNCTION TENew (destRect,viewRect: Rect): TEHandle;
```

`destRect` The destination rectangle for the new edit record, specified in the local coordinates of the current graphics port. This is the area in which text is laid out.

`viewRect` The view, or visible, rectangle for the new edit record, specified in the local coordinates of the current graphics port. This is the area of the window in which text is actually displayed.

DESCRIPTION

A monostyled edit record is one in which all text is restricted to a single font, size, and style. Use `TENew` when the text is to be rendered in attributes that are consistent from character to character. Otherwise, use `TEStyleNew`.

Call `TENew` once for every edit record you want allocated. Your application should store the handle to the edit record that is returned; many routines require it as an input parameter. The edit record assumes the drawing environment of the graphics port.

If your application contains more than one window where text editing occurs, you need to create an edit record for each window.

TEDispose

The `TEDispose` procedure removes a specified edit record and releases all memory associated with it.

```
PROCEDURE TEDispose (hTE: TEHandle);
```

`hTE` A handle to the edit record for which the allocated memory should be released.

DESCRIPTION

Call the `TEDispose` procedure only when you're completely through with an edit record.

Note that if your program retains a handle to text associated with the edit record that you are destroying with `TEDispose`, the handle becomes invalid because the `TEDispose` procedure disposes of it, as well as the dispatch record handle. If the record is multistyled, `TEDispose` also disposes all of the style-related handles: `STHandle`, `LHHandle`, `STScrpHandle`, `nullSTHandle`, and `TEStyleHandle`.

To continue to refer to the text after you've destroyed the edit record, you need to make a copy of the handle in the `hText` field of the edit record using the Operating System Utilities `HandToHand` function before you call `TEDispose`. (See *Inside Macintosh: Operating System Utilities* for more information.)

In addition to disposing of the edit record, the edit record handle, and the dispatch record handle, the `TEDispose` procedure destroys the null scrap associated with the edit record and releases the memory used for it.

Activating and Deactivating an Edit Record

When your application receives notification of an activate event, it can call the `TEActivate` procedure, which activates an edit record and highlights the selection range or displays a caret at the insertion point. When the activate event flag is set to deactivate the window, your application can call the `TEDeactivate` procedure, which changes an edit record's status from active to inactive and removes the selection range highlighting or the caret. (When outline highlighting is on, `TEDeactivate` frames the text or displays a dimmed caret.)

TEActivate

The `TEActivate` procedure activates the specified edit record.

```
PROCEDURE TEActivate (hTE: TEHandle);
```

`hTE` A handle to the specified edit record.

DESCRIPTION

When you call `TEActivate` for an edit record, the selection range is highlighted. If the selection range is an insertion point, `TEActivate` displays a caret there.
Call this procedure every time the Event Manager function `WaitNextEvent` reports that the window containing the edit record has become active.

If you do not call `TEActivate` before you call `TEClick`, `TEIdle`, or `TESetSelect`, the selection range is not highlighted, or, if the selection range is set to an insertion point, a caret is not displayed at the insertion point. However, if you have turned on outline highlighting through the `TEFeatureFlag` function for the edit record, the text of the selection range is framed or a dimmed or an unblinking caret is displayed at the insertion point.

SEE ALSO

For a description of the `WaitNextEvent` function, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

TEDeactivate

The `TEDeactivate` procedure deactivates an edit record.

```
PROCEDURE TEDeactivate (hTE: TEHandle);
```

`hTE` A handle to the specified edit record.

DESCRIPTION

When you call `TEDeactivate` for an edit record, the highlighted selection range is no longer displayed. If the selection range is an insertion point, `TEDeactivate` no longer displays the caret. However, if you turned on outline highlighting through the `TEFeatureFlag` function for the edit record, the text of the selection range is framed or a dimmed or an unblinking caret is displayed at the insertion point when the record is deactivated.

Call this procedure every time the Event Manager function `WaitNextEvent` reports that the window containing the edit record has become inactive.

SEE ALSO

For a description of the `WaitNextEvent` function, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Setting and Getting an Edit Record's Text and Character Attribute Information

The TextEdit procedure `TEKey` allows you to handle key-down events and enter text input through the keyboard. The procedure `TESetText` lets you incorporate existing text into the text buffer of an edit record. Once an edit record contains text, you can use the `TEGetText` function to get a handle to the text itself. For a multistyled edit record, you can get a handle to the style record by calling `GetStyleHandle`. You can set the handle to the style record using the `TESetStyleHandle` procedure. This section describes these routines.

TEKey

The `TEKey` procedure replaces the selection range in the text of the specified edit record with the input character and positions the insertion point just past the inserted character.

```
PROCEDURE TEKey (key: Char; hTE: TEHandle);
```

`key`	The input character.
`hTE`	A handle to the edit record in whose text the character is to be entered.

DESCRIPTION

If the selection range is an insertion point, `TEKey` inserts the character. (Two-byte characters are passed one byte at a time.) If the `key` parameter contains a backspace character, the selection range or the character immediately before the insertion point is deleted. When the primary line direction is right-to-left, the character to the right of the insertion point is deleted. When the primary line direction is left-to-right, the character to the left of the insertion point is deleted.

When the user deletes text up to the beginning of a set of character attributes, `TEKey` saves the attributes in the null scrap's style scrap record. The attributes are saved temporarily to be applied to characters inserted after the deletion. As soon as the user clicks in another area of the text, `TEKey` removes the attributes. `TEKey` redraws the text as necessary.

Call `TEKey` every time the Event Manager function `WaitNextEvent` reports a keyboard event that your application determines should be handled by TextEdit.

Because `TEKey` inserts every character passed in the `key` parameter, your application must filter all characters which aren't actual text, such as keys typed in conjunction with the Command key.

SEE ALSO

For a description of the `WaitNextEvent` function, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

TESetText

The `TESetText` procedure incorporates a copy of the specified text into the designated edit record.

```
PROCEDURE TESetText (text: Ptr; length: LongInt; hTE: TEHandle);
```

`text`	A pointer to the text to be copied and incorporated.
`length`	The number of characters in the text to be incorporated.
`hTE`	A handle to the edit record into which the text is to be copied.

DESCRIPTION

The `TESetText` procedure copies the specified text into the existing `hText` handle of the edit record, resizing the buffer, if necessary; it doesn't bring in the original text. The copied text is wrapped to the destination rectangle, and its `lineStarts` and `nLines` fields are calculated accordingly. The selection range is set to an insertion point at the end of the incorporated text. The `TESetText` procedure does not display the copied text on the screen. To do this, call `TEUpdate`.

TEGetText

The `TEGetText` function returns a handle to the text of the specified edit record.

```
FUNCTION TEGetText (hTE: TEHandle): CharsHandle;
```

`hTE` A handle to the edit record containing the text whose handle you want returned. You pass this handle as an input parameter.

`CharsHandle`
A handle to the text of the edit record.

DESCRIPTION

The `TEGetText` function doesn't make a copy of the text. Rather, it returns the handle to the text which is stored as a packed array of characters. (This handle belongs to TextEdit; your application must not destroy it.) The `teLength` field of the edit record contains the length of the text whose handle is returned.

The handle of type `CharsHandle` that is returned by `TEGetText` corresponds to the `hText` field of the edit record, but the data type is defined as follows:

```
TYPE  CharsHandle = ^CharsPtr;
      CharsPtr    = ^Chars;
      Chars       = PACKED ARRAY[0..32000] OF CHAR;
```

TESetStyleHandle

The `TESetStyleHandle` procedure sets an edit record's style handle, which is stored in the `txFont` and `txFace` fields.

```
PROCEDURE TESetStyleHandle (theHandle: TEStyleHandle;
                            hTE: TEHandle);
```

`theHandle` The style handle to be set in the combined `txFont` and `txFace` fields of the specified edit record.

`hTE` A handle to the edit record.

DESCRIPTION

The `TESetStyleHandle` procedure has no effect on monostyled edit records.

Your application should always use `TESetStyleHandle` rather than manipulate the fields of the edit record directly.

TEGetStyleHandle

The `TEGetStyleHandle` function returns the style handle stored in the designated edit record's `txFont` and `txFace` fields. The style handle points to the associated style record.

```
FUNCTION TEGetStyleHandle (hTE: TEHandle): TEStyleHandle;
```

`hTE` A handle to the multistyled edit record containing the style handle to be returned.

DESCRIPTION

The `TEGetStyleHandle` function returns a handle to the style record (of type `TEStyleRec`), not a copy of it. Because only multistyled edit records have style records, `TEGetStyleHandle` returns `NIL` when used with a monostyled edit record. To ensure future compatibility, your application should always use this function rather than manipulate the fields of the edit record directly.

Setting the Caret and Selection Range

Your application can call `TEIdle` to blink a caret at an insertion point during idle processing, the `TEClick` procedure to control the placement and highlighting of the text selection range in response to mouse-down events generated when a user clicks the mouse button, and the `TESetSelect` procedure to set the text selection range to be edited next or denote the insertion point. This section describes these routines.

TEIdle

When called repeatedly, the `TEIdle` procedure displays a blinking caret at the insertion point, if any exists, in the text of the specified edit record of an active window.

```
PROCEDURE TEIdle (hTE: TEHandle);
```

`hTE` A handle to the edit record.

DESCRIPTION

You need to call `TEIdle` only when the window containing the text is active; the caret is blinked only then. TextEdit observes a minimum blink interval, initially set to 32 ticks. No matter how often you call `TEIdle`, the time between blinks is never less than the minimum interval. (The user can adjust the minimum interval setting with the General Controls control panel.)

To maintain a constant frequency of blinking, you need to call `TEIdle` at least once each time through your main event loop. Call it more than once if your application does an unusually large amount of processing each time through the loop.

Call the Event Manager's `GetCaretTime` function to get the blink rate. (See the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.)

TEClick

The `TEClick` procedure controls placement and highlighting of the selection range as determined by mouse events.

```
PROCEDURE TEClick (pt: Point; extend: Boolean; hTE: TEHandle);
```

`pt`	The mouse location in local coordinates at the time the mouse button was pressed, obtainable from the event record (in global coordinates).
`extend`	A flag denoting the state of the Shift key at the time of the click as indicated by the Event Manager. If the Shift key was held down at the time of the click to extend the selection, pass a value of `TRUE`.
`hTE`	A handle to the edit record whose text is displayed in the view rectangle where the click occurred.

DESCRIPTION

Call `TEClick` whenever a mouse-down event occurs in the view rectangle of the edit record and the window associated with that edit record is active. The `TEClick` procedure keeps control until the mouse button is released. Use the QuickDraw procedure `GlobalToLocal` to convert the global coordinates of the mouse location given in the event record to the local coordinate system for `pt`.

The `TEClick` procedure removes highlighting of the old selection range unless the selection range is being extended. If the mouse moves, meaning that a drag is occurring, `TEClick` expands or shortens the selection range accordingly a character at a time. In the case of a double-click, the word where the cursor is positioned becomes the selection range.

SEE ALSO

For more information about the `GlobalToLocal` procedure, see the QuickDraw chapters in *Inside Macintosh: Imaging*.

TESetSelect

The `TESetSelect` procedure sets the selection range within the text of the specified edit record.

```
PROCEDURE TESetSelect (selStart, selEnd: LongInt; hTE: TEHandle);
```

`selStart`	The byte offset at the start of the text selection range.
`selEnd`	The byte offset at the end of the text selection range.
`hTE`	A handle to the edit record.

DESCRIPTION

The `TESetSelect` procedure removes highlighting of the old selection range and highlights the new one. If `selStart` equals `selEnd`, the new selection range is an insertion point, and a caret is displayed. If `selEnd` is anywhere beyond the last character of the text, `TESetSelect` uses the first position past the last character. The `selEnd` and `selStart` fields can range from 0 to 32767.

SPECIAL CONSIDERATIONS

When only the Roman script system is used, the selection range is always displayed and highlighted as a continuous range of text. However, when one or more script systems requiring mixed-directional display of text are installed, a continuous sequence of characters in memory may appear as a discontinuous selection when displayed.

Displaying and Scrolling Text

The routines that this section describes let you control how text is displayed. `TESetAlignment` lets you specify whether text is to be right aligned, left aligned, or centered. `TEUpdate` draws the text, updating the text editing window. `TETextBox` lets you draw static text in a box, such as a dialog box, without requiring that you first create an edit record. `TECalText` recalculates line breaks. `TEGetHeight` returns the height of all the lines of text between two lines. `TEScroll` scrolls the text by the amount you specify. `TEPinScroll` scrolls the text, automatically stopping when it scrolls the last line into view. `TEAutoView` lets you turn automatic scrolling on or off. `TESelView` automatically scrolls the text into view, if automatic scrolling is turned on through `TEAutoView`.

TESetAlignment

The `TESetAlignment` procedure sets the alignment of the specified text in an edit record so that it is centered, right aligned, or left aligned, or aligned according to the line direction.

```
PROCEDURE TESetAlignment (align: Integer; hTE: TEHandle);
```

`align` The alignment for the specified text.

`hTE` A handle to the edit record containing the text.

DESCRIPTION

You can use the following constants to specify the text alignment through the `align` parameter.

Constant	Value	Description
`teFlushDefault`	0	Align according to primary line direction
`teCenter`	1	Centered for all scripts
`teFlushRight`	–1	Right aligned for all scripts
`teFlushLeft`	–2	Left aligned for all scripts

For compatibility, the previous names of these constants are still supported. They are `teJustLeft`, `teJustCenter`, `teJustRight`, and `teForceLeft`.

The default value of the `just` field of the edit record is `teFlushDefault`. This means that text alignment is based on the primary line direction which is set by default according to the system script.

For languages that are read from right to left, text is right aligned by default. For languages that are read from left to right, text is left aligned by default. If you change the alignment, call the `InvalRect` procedure after `TESetAlignment` to redraw the text with the new alignment.

TextEdit does not support justified alignment. To draw justified text, use the QuickDraw Text routines.

SEE ALSO

For more information about the `InvalRect` procedure, see the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*. For more information about drawing justified text, see the chapter "QuickDraw Text" in this book.

TEUpdate

The `TEUpdate` procedure draws the specified text within a given update rectangle.

```
PROCEDURE TEUpdate (rUpdate: Rect; hTE: TEHandle);
```

`rUpdate` The update rectangle, given in the coordinates of the current graphics port, where the specified text is to be drawn.

`hTE` A handle to the edit record containing the text to be drawn.

DESCRIPTION

Call `TEUpdate` every time the Event Manager function `WaitNextEvent` reports an update event for a text editing window—after you call the Window Manager procedure `BeginUpdate`, and before you call the `EndUpdate` procedure. You also need to erase the update region with the `EraseRect` procedure. If you don't the caret can sometimes remain visible when the window is deactivated.

SEE ALSO

For a description of the `WaitNextEvent` function, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*. For more information about the `BeginUpdate` and `EndUpdate` procedures, see the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

TETextBox

The `TETextBox` procedure draws the indicated text in a given rectangle with the specified alignment.

```
PROCEDURE TETextBox (text: Ptr; length: LongInt; box: Rect;
                     align: Integer);
```

`text` A pointer to the text to be drawn.

`length` The number of bytes comprising the text.

`box` The rectangle where the text is to be drawn. The rectangle is specified in local coordinates (of the current graphics port) and must be at least as wide as the first character drawn. (A good rule of thumb is to make the rectangle at least 20 pixels wide.)

`align` The kind of alignment for the specified text.

DESCRIPTION

The `TETextBox` procedure provides you with an easy way to display static text to a user. It creates its own monostyled edit record, which it deletes when finished with it, so you cannot edit the text it draws. The `TETextBox` procedure breaks a line of text correctly. You can specify how text is aligned in the box using any of the following alignment constants:

Constant	Description
`teFlushDefault`	Aligned according to primary line direction
`teCenter`	Centered for all scripts
`teFlushRight`	Right aligned for all scripts
`teFlushLeft`	Left aligned for all scripts

TECalText

The `TECalText` procedure recalculates the beginnings of all lines of text in the specified edit record.

```
PROCEDURE TECalText (hTE: TEHandle);
```

`hTE` A handle to the edit record whose text lines are to be recalculated.

DESCRIPTION

The `TECalText` procedure updates elements of the `lineStarts` array in an edit record. Call `TECalText` if you've changed the destination rectangle, the `hText` field, or any other property of the edit record that pertains to line breaks and the number of characters per line—for example, font, size, style, and so on.

ASSEMBLY-LANGUAGE INFORMATION

The low-memory global variable `TERecal` contains the address of the routine called by `TECalText` to recalculate the line starts and set the first and last characters that need to be redrawn. The `TERecal` default hook routine calls the Text Utilities `StyledLineBreak` function. If you replace the default `TERecal` hook routine with a customized version and your application supports non-Roman script systems, make sure that your customized hook routine is script-aware. The registers on entry and exit for this hook routine are:

Registers on entry

A3 Pointer to the locked edit record

D7 Change in the length of the record (word)

Registers on exit

D2 Line start of the line containing the first character to be redrawn (word)

D4 Position of last character to be redrawn (word)

TextEdit uses the low-memory global variable `WordRedraw` widely, but primarily for line calculations and to determine how much of a line to redraw after the user types in a character. TextEdit sets the correct value for `WordRedraw` in `TEInit` based upon the installed script systems. If a 2-byte script is installed, `TEInit` performs an OR operation on `WordRedraw` with a 1; if a right-to-left script is installed, `TEInit` performs an OR operation on `WordRedraw` with an $FF. The size of this global is one byte.

TextEdit interprets the final value of `WordRedraw` as follows:

Value	Description
0	Redraws the character before the entered character.
1	Redraws the word before the entered character.
$FF	Redraws the whole line.

TEGetHeight

The `TEGetHeight` function returns the total height of all of the lines in the text between and including the specified starting and ending lines.

```
FUNCTION TEGetHeight (endLine, startLine: LONGINT;
                      hTE: TEHandle): INTEGER;
```

`endLine` The number of the last line of text whose height is to be included in the total height. You can specify a value that is greater than or equal to 1 for this parameter.

`startLine` The number of the first line of text whose height is to be included in the total height. You can specify a value that is greater than or equal to 1 for this parameter.

`hTE` A handle to the edit record containing the lines of text whose height is to be returned.

DESCRIPTION

For monostyled text, the `TEGetHeight` function uses the value of the edit record's `lineHeight` field. For multistyled text, it uses the line height element (`LHElement`) of the line height table (`LHTable`). Note that `TEGetHeight` does not take into account the height of any blank lines at the end of the text. You need to consider this when scrolling text.

TEScroll

The `TEScroll` procedure scrolls the text within the view rectangle of the specified edit record by the designated number of pixels.

```
PROCEDURE TEScroll (dh,dv: Integer; hTE: TEHandle);
```

`dh`	The distance in pixels that the text is to be scrolled horizontally. A positive value moves the text to the right; a negative value moves the text to the left.
`dv`	The distance in pixels that the text is to be scrolled vertically. A positive value moves the text down; a negative value moves the text up.
`hTE`	A handle to the edit record whose text is to be scrolled.

DESCRIPTION

The `TEScroll` procedure updates the text on the screen automatically to reflect the new scroll position. The destination rectangle is offset by the amount scrolled. The `TEScroll` and `TEPinScroll` procedures behave the same, except that `TEPinScroll` stops scrolling when the last line of text is scrolled into view.

TEPinScroll

The `TEPinScroll` procedure scrolls the text within the view rectangle of the specified edit record by the designated number of pixels. Scrolling stops when the last line of text is scrolled into view.

```
PROCEDURE TEPinScroll (dh: Integer; dv: Integer; hTE: TEHandle);
```

`dh`	The distance in pixels that the text is to be scrolled horizontally. A positive value moves the text to the right; a negative value moves the text to the left.
`dv`	The distance in pixels that the text is to be scrolled vertically. A positive value moves the text down; a negative value moves the text up.
`hTE`	A handle to the edit record whose text is to be scrolled.

DESCRIPTION

The `TEPinScroll` procedure updates the text on the screen automatically to reflect the new scroll position, as does the `TEScroll` procedure. The destination rectangle is offset by the amount scrolled. When the edit record is longer than the text it contains, `TEPinScroll` displays up to the last line of text inclusive, and not beyond it.

TEAutoView

The `TEAutoView` procedure enables and disables automatic scrolling of the text in the specified edit record.

```
PROCEDURE TEAutoView (fAuto: Boolean; hTE: TEHandle);
```

`fAuto` A flag indicating whether to enable or disable automatic scrolling. A value of `TRUE` enables automatic scrolling. A value of `FALSE` disables automatic scrolling.

`hTE` A handle to the edit record for which automatic scrolling is to be enabled or disabled.

DESCRIPTION

The `TEAutoView` procedure does not actually scroll the text automatically: `TESelView` does. However, when `fAuto` is set to `FALSE`, a call to `TESelView` has no effect.

If there is a scroll bar associated with the edit record, your application must manage scrolling of it. You can replace the default click loop routine, which scrolls the text only, with a customized version that also updates the scroll bar.

You can also enable or disable automatic scrolling for an edit record through the `teFAutoScroll` feature of the `TEFeatureFlag` function.

SEE ALSO

For more information, see "TEFeatureFlag" on page 2-109.

TESelView

Once automatic scrolling has been enabled by a call to the `TEAutoView` procedure or through the `TEFeatureFlag` function, the `TESelView` procedure ensures that the selection range is visible and scrolls it into the view rectangle if necessary.

```
PROCEDURE TESelView (hTE: TEHandle);
```

`hTE` A handle to the edit record containing the text selection range.

DESCRIPTION

The top left part of the selection range is scrolled into view. If the text is displayed in a rectangle that is not high enough, automatic scrolling can cause text to appear to flicker. If automatic scrolling is disabled, `TESelView` has no effect.

SEE ALSO

For more information, see "TEFeatureFlag" on page 2-109.

Modifying the Text of an Edit Record

Although all of the TextEdit routines provide and support editing capabilities, the set of routines described in this section implement the standard Macintosh editing features. An application can use these routines to delete, insert, cut, copy, or paste multistyled or monostyled text. The routines that you use for these purposes are `TEDelete` to remove a selected range of text, `TEInsert` to insert text, `TECut` to remove the text, but save it to be inserted, `TECopy` to copy the selected text with affecting the selection range, `TEPaste` to replace the selected text with the text in the private scrap, without applying character attribute information, `TEStylePaste` to replace the selected text with text and its character attribute information from the desk scrap, and `TEToScrap` and `TEFromScrap` to move monostyled text across applications or between applications and a desk accessory.

TEDelete

The `TEDelete` procedure removes the selected range of text from the text of the designated edit record and redraws the remaining text as necessary.

```
PROCEDURE TEDelete (hTE: TEHandle);
```

`hTE` A handle to the edit record containing the text to be deleted.

DESCRIPTION

When the `TEDelete` procedure deletes a selected range of text, it does not transfer the text to either the private scrap or the Scrap Manager's desk scrap.

For multistyled records, when you use `TEDelete` to delete a selected range of text, the associated character attributes are saved in the null scrap to be applied to characters entered after the text is deleted. When the user clicks in some other area of the text, the character attributes are removed from the null scrap. You can use `TEDelete` to implement the Clear command. The `TEDelete` procedure recalculates line starts and line heights.

TEInsert

The `TEInsert` procedure inserts the specified text immediately before the selection range or the insertion point in the text of the designated edit record, redrawing the text as necessary.

```
PROCEDURE TEInsert (text: Ptr; length: LongInt; hTE: TEHandle);
```

`text`	A pointer to the text to be inserted.
`length`	The number of characters to be inserted.
`hTE`	A handle to the edit record containing the text buffer into which the new text is to be inserted.

DESCRIPTION

When you call the `TEInsert` procedure and a range of text is selected, `TEInsert` doesn't affect the selection range. The `TEInsert` procedure does not check for a 32 KB limit, so your application must ensure that the inserted text does not exceed this text size limit of 32 KB. The `TEInsert` procedure recalculates line starts and line heights to adjust for the inserted text.

TECut

The `TECut` procedure removes the current selection range from the text of the designated edit record, redrawing the text as necessary.

```
PROCEDURE TECut (hTE: TEHandle);
```

`hTE`	A handle to the edit record containing the text to be cut.

DESCRIPTION

For monostyled text, the `TECut` procedure writes the cut text to the private scrap.

For multistyled text, `TECut` writes the cut text to the private scrap and its character attributes to the style scrap; it also writes both to the Scrap Manager's desk scrap. For multistyled text, the `TECut` procedure removes the character attributes from the style record's style table when the text is cut.

For both monostyled and multistyled text, if the selection range is an insertion point, TextEdit deletes everything from the private scrap. When the selection range is an insertion point and the text is multistyled, `TECut` has no effect on the style scrap or the Scrap Manager's desk scrap.

SEE ALSO

For more information about the desk scrap, see the chapter "Scrap Manager" in *Inside Macintosh: More Macintosh Toolbox*.

TECopy

The `TECopy` procedure copies the text selection range from the edit record, leaving the selection range intact.

```
PROCEDURE TECopy (hTE: TEHandle);
```

`hTE` A handle to the edit record containing the text to be copied.

DESCRIPTION

The `TECopy` procedure copies the text to the private scrap. For text of a monostyled edit record, the text is written to the private scrap only. For text of a multistyled edit record, the text is written to the TextEdit private scrap, the character attribute information is written to the TextEdit style scrap, and both are written to the Scrap Manager's desk scrap. Anything previously in the private scrap is deleted before the copied text is written to it.

For both multistyled and monostyled text, if the selection range is an insertion point, TECopy empties the TextEdit private scrap. When the selection range is an insertion point and the text is multistyled, `TECopy` has no effect on the null scrap, the style scrap, or the Scrap Manager's desk scrap.

SEE ALSO

For more information about the desk scrap, see the chapter "Scrap Manager" in *Inside Macintosh: More Macintosh Toolbox*.

TEPaste

The `TEPaste` procedure replaces the edit record's selected text with the contents of the private scrap and leaves an insertion point after the inserted text. If the selection range is an insertion point, `TEPaste` inserts the contents of the private scrap there.

```
PROCEDURE TEPaste (hTE: TEHandle);
```

`hTE` A handle to the edit record into which the text is to be pasted.

DESCRIPTION

When you call `TEPaste`, after it pastes the text from the private scrap, it redraws all of the text as necessary. If the private scrap is empty, `TEPaste` deletes the selection range. If you call `TEPaste` for a multistyled edit record, it pastes only the text in the private scrap. In this case, `TEPaste` ignores any associated character attribute information stored in the style scrap; instead, it applies the character attributes of the first character of the selection range being replaced to the text. If the selection range is an insertion point, `TEPaste` applies the character attributes of the character preceding the insertion point.

TEStylePaste

The `TEStylePaste` procedure pastes text and its associated character attribute information from the desk scrap into the edit record's text at the insertion point—if the current selection range is an insertion point—or it replaces the current selection range.

```
PROCEDURE TEStylePaste (hTE: TEHandle);
```

`hTE` A handle to the edit record into which the text is to be pasted.

DESCRIPTION

When you call `TEStylePaste` and there is no character attribute information associated with text in the desk scrap, `TEStylePaste` first checks the null scrap. If the null scrap contains character attribute information, this is used. If the null scrap is empty, `TEStylePaste` gives the text the same attributes as those of the first character of the replaced selection range or that of the preceding character if the selection is an insertion point.

For a monostyled edit record, `TEStylePaste` pastes the text only; there is no associated character attribute information because all the text uses the same attributes.

TEToScrap

The `TEToScrap` function copies the contents of the TextEdit private scrap to the desk scrap.

```
FUNCTION TEToScrap: OSErr;
```

DESCRIPTION

You use the `TEToScrap` function to move monostyled text across applications or between an application and a desk accessory. Call the Scrap Manager function `ZeroScrap` to initialize the desk scrap or clear its contents before calling `TEToScrap`.

ASSEMBLY-LANGUAGE INFORMATION

Copy the contents of the private scrap to the desk scrap by calling the Scrap Manager function `PutScrap`; you can get the values you need from the global variables `TEScrpHandle` and `TEScrpLength`.

RESULT CODES

`noErr`	0	No error
`noScrapErr`	–100	Desk scrap isn't initialized

SEE ALSO

For more information about the `PutScrap` function, the `ZeroScrap` function, and the desk scrap, see the chapter "Scrap Manager" in *Inside Macintosh: More Macintosh Toolbox*.

TEFromScrap

The `TEFromScrap` function copies the contents of the desk scrap to the TextEdit private scrap.

```
FUNCTION TEFromScrap: OSErr;
```

DESCRIPTION

You use this function to move monostyled text across applications or between an application and a desk accessory.

ASSEMBLY-LANGUAGE INFORMATION

You can store a handle to the desk scrap in the global variable `TEScrpHandle` and the size of the desk scrap in the global variable `TEScrpLength`; get the desk scrap's handle and size by calling the Scrap Manager's `InfoScrap` function.

RESULT CODE

`noErr`	0	No error

SEE ALSO

For more information about the `InfoScrap` function and the desk scrap, see the chapter "Scrap Manager" in *Inside Macintosh: More Macintosh Toolbox*.

Managing the TextEdit Private Scrap

This section describes the routines that you use to manage the private scrap. You use the `TEScrapHandle` function get a handle to the private scrap, the `TEGetScrapLength` function to determine its size, and the `TESetScrapLength` procedure to set its size.

TEScrapHandle

The `TEScrapHandle` function returns a handle to the TextEdit private scrap.

```
FUNCTION TEScrapHandle: Handle;
```

ASSEMBLY-LANGUAGE INFORMATION

You can get the handle to the private scrap from the global variable `TEScrpHandle`.

TEGetScrapLength

The `TEGetScrapLength` function returns the size of the TextEdit private scrap in bytes.

```
FUNCTION TEGetScrapLength: LongInt;
```

ASSEMBLY-LANGUAGE INFORMATION

You can get the size of the private scrap in bytes from the global variable `TEScrpLength`.

TESetScrapLength

The `TESetScrapLength` procedure sets the size of the TextEdit private scrap to the specified number of bytes.

```
PROCEDURE TESetScrapLength (length: LongInt);
```

`length` The size of the private scrap in bytes.

ASSEMBLY-LANGUAGE INFORMATION

You can set the global variable `TEScrpLength` to the size of the private scrap.

Checking, Setting, and Replacing Styles

The routines described in this section let you manipulate the character attribute information associated with a range of text. You can use the following routines to set, replace, or copy character attribute information, or to check aspects of the text's character attributes. These routines are `TESetStyle`, `TEReplaceStyle`, `TEContinuousStyle`, `TEStyleInsert`, `TEGetStyleScrapHandle`, `TEUseStyleScrap`, and `TENumStyles`.

Note
In the original *Inside Macintosh* documentation the term *style* was used to refer to the text font, size, style (face), and color. In this chapter the term *character attributes* is used instead. This is so that the term *style* can be used consistently throughout all of the documentation to refer to the following text style attributes: bold, italic, underline, outline, condense, extend, and shadow. In the past, the term *face*, which is now obsolete, was used to refer to these attributes instead of *style*. ◆

TESetStyle

The `TESetStyle` procedure sets new character attributes for the current selection range in the specified edit record.

```
PROCEDURE TESetStyle (mode: Integer; newStyle: TextStyle;
                      redraw: Boolean; hTE: TEHandle);
```

`mode` A selector that specifies which character attributes are to be changed. The value for `mode` can be any additive combination of the `mode` constants for font, style, type size, color, and so forth.

`newStyle` A record of type `TextStyle` that specifies the new attributes to be set. This record contains the character attributes to be applied to the current selection range based on the value of `mode`.

`redraw`	A flag that specifies whether or not TextEdit should immediately redraw the affected text to reflect the new character attribute changes. A value of `TRUE` causes the text to be redrawn immediately. Line breaks, line heights, and line ascents are recalculated. A value of `FALSE` delays redrawing until another event forces the update.
`hTE`	A handle to the multistyled edit record containing the selected text.

DESCRIPTION

The `TESetStyle` procedure has no effect on a monostyled record. You can use any combination of the following constants to specify a value for the `mode` parameter. The value of `mode` specifies which existing character attributes are to be changed to the new character attributes specified by `newStyle`.

Constant	Value	Description
`doFont`	1	Sets the font family ID
`doFace`	2	Sets the character style
`doSize`	4	Sets the type size
`doColor`	8	Sets the color
`doAll`	15	Sets all attributes
`addSize`	16	Increases or decreases the current type size
`doToggle`	32	Modifies the mode

If `doToggle` is specified along with `doFace` and if an attribute specified in the given `newStyle` parameter exists across the entire selected range of text, then `TESetStyle` removes that attribute. Otherwise, if the attribute doesn't exist across the entire selection range, all of the selected text is set to include that character attribute.

If the `redraw` parameter is set to `TRUE`, TextEdit redraws the current selection range using the new character attributes, recalculating line breaks, line heights, and line ascents. If the `redraw` parameter is set to `FALSE`, TextEdit does not redraw the text or recalculate line breaks, line heights, and line ascents. Consequently, when you call a routine that uses any of this information, such as `TEGetHeight` (which returns a total height between two specified lines), it does not reflect the new character attributes set with `TESetStyle`. Instead, the routine uses the information that was available before `TESetStyle` was called. To update this information, call the `TECalText` procedure. (See "TECalText" on page 2-91 for more information.) To be certain that the new information is always reflected, call the `TESetStyle` procedure with the `redraw` parameter set to `TRUE`.

If you call the `TESetStyle` routine when the value of the `selStart` field of an edit record equals the value of the `selEnd` field (specifying an insertion point), TextEdit stores the input character attributes in the null scrap record pointed to by the null style handle.

TEReplaceStyle

The `TEReplaceStyle` procedure replaces any character attributes in the current selection range that match the specified existing character attributes with the specified new character attributes.

```
PROCEDURE TEReplaceStyle (mode: INTEGER;
                          oldStyle,newStyle: TextStyle;
                          redraw: BOOLEAN; hTE: TEHandle);
```

`mode` A selector that specifies which attributes to replace. It corresponds to any additive combination of the `mode` constants for font, character style, type size, color, and so forth.

`oldStyle` A pointer to a text style record that specifies the current character attributes to search for in the selected text.

`newStyle` A pointer to a text style record that specifies the new attributes to be set. This record contains the character attributes to be applied to the current selection range based on the value of `mode`.

`redraw` A flag that specifies whether or not TextEdit should immediately redraw the text to reflect the attribute changes. A value of `FALSE` delays redrawing until another event forces the update. A value of `TRUE` causes the text to be redrawn immediately using the new character attributes.

`hTE` A handle to the multistyled edit record containing the text selection whose character attributes are to be changed.

DESCRIPTION

The `TEReplaceStyle` procedure replaces any attribute in the current selection range that matches the attribute specified by `oldStyle` with that given by `newStyle`. Only the character attributes specified by `mode` are affected.

Attribute changes are made directly to the style elements (`STElement`) within the style table itself (`TEStyleTable`). If you specify the value `doAll` for the `mode` parameter, `newStyle` replaces `oldStyle` outright. Possible values for the `mode` parameter are defined by the following constants. The `TEReplaceStyle` procedure has no effect on a monostyled edit record.

Constant	Value	Description
`doFont`	1	Sets the font family ID
`doFace`	2	Sets the character style
`doSize`	4	Sets the type size
`doColor`	8	Sets the color
`doAll`	15	Sets all attributes
`addSize`	16	Increases or decreases the current type size

TEContinuousStyle

The `TEContinuousStyle` function determines whether a given character attribute is continuous over the current selection range.

```
FUNCTION TEContinuousStyle (VAR mode: Integer;
                            VAR aStyle: TextStyle;
                            hTE: TEHandle): Boolean;
```

`mode` On input, a selector specifying the attributes to be checked. On output, `mode` identifies only those attributes determined to be continuous over the selection range.

`aStyle` On input, a text style record. On output, this record contains the values for the `mode` attributes determined to be continuous over the selection.

`hTE` A handle to the edit record containing the selected text whose attributes are to be checked.

DESCRIPTION

This function does not modify the text selection. Possible values for the `mode` parameter are defined by the following constants.

Constant	Value	Description
`doFont`	1	Specifies the font family ID
`doFace`	2	Specifies the character style
`doSize`	4	Specifies the type size
`doColor`	8	Specifies the color
`doAll`	15	Specifies all the attributes

The `TEContinuousStyle` function returns `TRUE` if all of the attributes to be checked are continuous and returns `FALSE` if none or some are continuous.

If the current selection range is an insertion point, `TEContinuousStyle` first checks the null scrap. If the null scrap contains character attributes, then they are used based on the value of the `mode` parameter. Otherwise, if the null scrap is empty, `TEContinuousStyle` returns the attributes of the character preceding the insertion point. The `TEContinuousStyle` function always returns `TRUE` in this case, and each field of the text style record is set if the corresponding bit in the `mode` parameter is set.

If the value of `hTE` is a handle to a monostyled edit record, `TEContinuousStyle` returns the set of character attributes that are consistent for the entire record.

Note that fields in the text style record specified by `aStyle` are only valid if the corresponding bits are set in the `mode` variable.

How the `tsFace` field of the `aStyle` record is used requires some consideration. For example, if `TEContinuousStyle` returns a `mode` parameter that contains `doFace` and the text style record `tsFace` field is bold, it means that the selected text is all bold, but

may contain other text styles, such as italic, as well. Italic does not apply to all of the selected text, or it would have been included in the `tsFace` field. If the `tsFace` field is an empty set, then all of the selected text is plain.

TEStyleInsert

The `TEStyleInsert` procedure inserts the specified text immediately before the selection range or the insertion point in the edit record's text and applies the specified character attributes to the text, redrawing the text if necessary.

```
PROCEDURE TEStyleInsert (text: Ptr; length: LongInt;
                         hST: STScrpHandle; hTE: TEHandle);
```

`text`	A pointer to the text to be inserted.
`length`	The length in bytes of the text to be inserted.
`hST`	A handle to the style scrap record containing the character attribute information to be applied to the inserted text.
`hTE`	A handle to the edit record into which the text is to be inserted.

DESCRIPTION

You should create your own style scrap record, specifying the character attributes to be inserted and applied to the text, and pass its handle to `TEStyleInsert` as the value of the `hST` parameter. The character attributes are copied directly into the style record's (`TEStyleRec`) style table.

The `TEStyleInsert` procedure does not affects the current selection range.

TEGetStyleScrapHandle

The `TEGetStyleScrapHandle` function creates a style scrap record, copies the character attributes associated with the current selection range into it, and returns a handle to it.

```
FUNCTION TEGetStyleScrapHandle (hTE: TEHandle): STScrpHandle;
```

`hTE`	The handle to the edit record containing the text selection range whose character attributes are to be copied.

DESCRIPTION

The `TEGetStyleScrapHandle` function creates a style scrap record of type `StScrpRec` and copies the character attributes associated with the current selection

range of the designated edit record into it. If the current selection range is an insertion point, `TEGetStyleScrapHandle` first checks the null scrap. If the null scrap contains character attributes, they are written to the newly created style scrap record. If the null scrap is empty, the attributes associated with the character preceding the insertion point are copied to the style scrap record.

The `TEGetStyleScrapHandle` function has no impact on the Scrap Manager's desk scrap. The `TEGetStyleScrapHandle` function returns a `NIL` value if called with a handle to a monostyled record.

TEUseStyleScrap

The `TEUseStyleScrap` procedure assigns new character attributes to the specified range of text in the designated edit record.

```
PROCEDURE TEUseStyleScrap (rangeStart: LongInt; rangeEnd: LongInt;
                           newStyles: STScrpHandle; redraw: Boolean;
                           hTE: TEHandle);
```

`rangeStart`
: The offset of the first character in the text of the edit record to which the character attributes are to be applied.

`rangeEnd`
: The offset of the last character in the text of the edit record to which the character attributes are to be applied.

`newStyles`
: A handle to a style scrap record. The style scrap record contains the attributes to be applied to the specified range of text. If the value of `newStyles` is `NIL`, no action is performed.

`redraw`
: A flag that specifies whether TextEdit should immediately redraw the selection range using the new character attributes.

`hTE`
: A handle to the edit record containing the range of text to which the character attributes are to be applied. If the handle points to a monostyled edit record (created using `TENew`), no action is performed.

DESCRIPTION

The `TEUseStyleScrap` procedure writes the character attribute information into the style record's style table and updates the style run table. If the `redraw` parameter is set to `TRUE`, the attributes are applied immediately to the specified range of text, and line breaks, line heights, and line ascents are recalculated. If `redraw` is set to `FALSE`, the new character attributes are not reflected in the view rectangle until the next update event occurs.

Regardless of whether the text is redrawn, the current selection range is not changed; if characters are highlighted before `TEUseStyleScrap` is called, they remain highlighted after it is called. However, if characters within the current selection range also fall within

the specified range of text, they are rendered in the new character attributes when the text is redrawn.

Each element in the style scrap record contains a field that is the offset of the beginning of the element's character attributes. This field (`scrpStartChar`) defines the boundaries for the scrap's style runs.

The `TEUseStyleScrap` procedure applies the first element's attributes to the characters from `rangeStart` up to the `scrpStartChar` field of the next element. The `TEUseStyleScrap` procedure terminates without error if it prematurely reaches the end of the range or if there are not enough scrap style elements to cover the whole range. In the latter case, `TEUseStyleScrap` applies the last set of character attributes in the style scrap record to the remainder of the range.

Depending on the requirements of your application, you can create a style scrap record directly and pass its handle to `TEUseStyleScrap` as the value of `newStyles` or you can use a style scrap record created by `TEGetStyleScrapHandle`.

TENumStyles

The `TENumStyles` function returns the number of character attribute changes contained in the specified range, counting one for the start of the range.

```
FUNCTION TENumStyles (rangeStart: LongInt; rangeEnd: LongInt;
                      hTE: TEHandle): LongInt;
```

`rangeStart`
: The beginning of the range of text for which the number of style runs (sets of character attributes) or changes is counted and returned.

`rangeEnd`
: The end of the range of text for which the number of style runs (sets of character attributes) or changes is counted and returned.

`hTE`
: A handle to the edit record containing the range of text.

DESCRIPTION

The number of character attribute changes that `TENumStyles` returns does not necessarily represent the number of unique sets of attributes for the range because some sets of attributes may be repeated. For monostyled edit records, `TENumStyles` always returns 1.

Using Byte Offsets and Corresponding Points

You can use the `TEGetOffset` function to convert a point to its corresponding byte offset and the `TEGetPoint` function to convert a byte offset to its corresponding point. These functions are discussed in this section.

TEGetOffset

The `TEGetOffset` function finds the byte offset of a character in an edit record's text that corresponds to the specified point.

```
FUNCTION TEGetOffset (pt: Point; hTE: TEHandle): Integer;
```

`pt`	A point in the displayed text of the specified edit record.
`hTE`	A handle to the edit record containing the text.

DESCRIPTION

The `TEGetOffset` function works for both monostyled and multistyled edit records. The `TEGetOffset` function returns the byte offset of the first byte for a 2-byte character.

TEGetPoint

The `TEGetPoint` function determines the point that corresponds to the specified byte offset of a character and returns the coordinates of that point.

```
FUNCTION TEGetPoint (offset: Integer; hTE: TEHandle): Point;
```

`offset`	A byte offset into the text buffer of an edit record.
`hTE`	A handle to the edit record containing the text.

DESCRIPTION

The `TEGetPoint` function returns a valid result even when the edit record does not contain any text. The point returned is based on the values in the record's destination rectangle. In the case of an offset being equal to a line end, which is also the start of the next line, `TEGetPoint` returns a point corresponding to the line start of the next line. In the case of a dual caret, the primary caret position, the one corresponding to the primary line direction, is returned.

The line height, taken either from the `lineHeight` field for a monostyled edit record or from the line-height array, `LHElement`, for a multistyled edit record, is also used to determine the vertical component. Both the text direction and the primary line direction are used to determine the horizontal component.

The `TEGetPoint` function works for both monostyled and multistyled edit records.

Additional TextEdit Features

The `TEFeatureFlag` function lets you check the status of additional TextEdit features and enable or disable them. It is described in this section.

TEFeatureFlag

The `TEFeatureFlag` function turns a specified feature on or off or returns the current status of that feature. Features supported are automatic scrolling, text buffering, outline highlighting, inline input, and text services.

```
FUNCTION TEFeatureFlag (feature: Integer; action: Integer;
                        hTE: TEHandle): Integer;
```

`feature` The feature for which the action is to be performed.

`action` A selector stipulating that the feature, specified by the `feature` parameter, is to be turned on or off, or that the current status of the feature is to be returned.

`hTE` A handle to the edit record for which the action should be performed.

DESCRIPTION

You can use the `TEFeatureFlag` function to check the status of additional TextEdit features—automatic scrolling, outline highlighting, and text buffering—and to enable or disable the feature. You can also use this function to disable inline input in a particular edit record and to enable several features that have been provided so that inline input works correctly with TextEdit.

To identify a feature, you specify one of the following constants as the value of the `feature` parameter.

Constant	Value	Description
`teFAutoScroll`	0	Automatic scrolling
`teFTextBuffering`	1	Text buffering
`teFOutlineHilite`	2	Outline highlighting
`teFInlineInput`	3	Inline input
`teFUseTextServices`	4	Use inline input service

You specify the `action` to be performed on a feature through the following constants.

Constant	Value	Description
`teBitClear`	0	Disables the specified feature
`teBitSet`	1	Enables the specified feature
`teBitTest`	–1	Returns the current setting of the specified feature

If `teBitTest` returns `teBitSet`, the feature is enabled; if it returns `teBitClear`, it is disabled.

You can use the `TEFeatureFlag` function to turn automatic scrolling on and off as an alternative to calling `TEAutoView`. The effect is the same.

The `teFOutlineHilite` selector specifies outline highlighting as the feature for which an action is to be performed. If a highlighted region exists in an edit record and the window is inactive, then the highlighted region is outlined or framed.

In the case that outline highlighting is enabled and the current selection range is an insertion point, the caret is then drawn in a gray pattern so that it appears dimmed. To do the framing and caret dimming, TextEdit temporarily replaces the current address in the `highHook` and `caretHook` fields of the edit record, redraws the caret or the highlighted region, and then immediately restores the hooks to their previous addresses.

The `teFTextBuffering` selector enables or disables text buffering for performance improvements of 2-byte scripts. This is a global buffer, as opposed to the `TEKey` procedure's internal 2-byte buffer, and it is used across all active edit records. When using text buffering, take the following precautions:

- Exercise care when you enable the text-buffering capability in more than one active record; otherwise, the bytes that are buffered from one edit record may appear in another edit record.
- Ensure that buffering is not turned off in the middle of processing a 2-byte character. To guarantee the integrity of your record, it is important that you wait for an idle event before you disable buffering or enable buffering in a second edit record.
- When text buffering is enabled, ensure that `TEIdle` is called before any pause of more than a few ticks—for example, before the Event Manager procedure `WaitNextEvent`. A possibility of a long delay before characters appear on the screen exists, especially in non-Roman systems. If you do not call `TEIdle`, the characters can end up in the edit record of another application. For more information, see "TEIdle" on page 2-86.

If text buffering is enabled on a non-Roman script system and the keyboard has changed, TextEdit flushes the text of the current script from the buffer before bringing characters of the new script into the buffer.

If your application follows the guidelines for inline input available from Macintosh Developer Technical Support, then you should set the `useTextEditServices` flag in the Size resource in your application. This allows inline input to work with your application. **Inline input** is a keyboard input method (often used for double-byte script systems) in which conversion from a phonetic to an ideographic representation of a character takes place at the current line position where the text is intended to appear. This allows the user to type text directly in the line as opposed to a special conversion window. If inline input is installed and the `useTextEditServices` flag in the Size resource is set, inline input sets TextEdit's `teFUseTextServices` feature bit whenever an edit record is created. TextEdit does not use this bit.

Inline input checks the `teFUseTextServices` bit during text editing to determine if an inline session should begin. If you want to disable inline input for a particular edit record, your application can clear this bit after the edit record is created. You can also clear this bit to disable inline input temporarily and then restore it, but the edit record must always be deactivated before the state of the bit is changed.

IMPORTANT

You *must* deactivate an edit record (using `TEDeactivate`) before changing the state of the feature bits or any fields in the edit record. ▲

In the future, other text services may use this same mechanism. If you follow the guidelines specified here, your application should also work with future text services. When an inline edit session begins, inline input also sets the `teFInlineInput` bit to provide the following features so that inline input works correctly with TextEdit:

- disabling font and keyboard synchronization
- forcing a multiple-line selection to be highlighted line by line using a separate rectangle for each line rather than using a minimum number of rectangles for optimization
- highlighting a line only to the edge of the text rather than beyond the text to the edge of the view rectangle

IMPORTANT

The `teFInlineInput` bit is cleared by inline input when an inline session ends. Use the `teFInlineInput` constant in the feature parameter of `TEFeatureFlag` to include these features in your application even when inline input is not installed. Be careful about changing the state of this bit if the `teFUseTextServices` bit is set. Again, the edit record should always be deactivated before you change the state of the `teFInlineInput` bit. If you clear the `teFUseTextServices` bit and you set the `teFInlineInput` bit, inline input is disabled, but your application retains the features listed above. ▲

To test for the availability of these features, you can call the `Gestalt` function with the `gestaltTextEditVersion` selector. A result of `gestaltTE4` or greater returned in the response parameter indicates that outline highlighting and text buffering are available. A result of `gestaltTE5` or greater returned in the response parameter indicates that the two inline input features are available.

The inline input features are also available on version 6.0.7 systems with non-Roman script systems installed. However, there is no Gestalt constant that indicates this availability.

SEE ALSO

For a description of the `WaitNextEvent` function, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Customizing TextEdit

The TextEdit `TECustomHook`, `TESetWordBreak`, and `TESetClickLoop` routines described in this section let you customize the behavior of TextEdit. You can use these routines to replace the default hook routines with your customized versions.

However, if you use any of the TextEdit hooks to override default TextEdit behavior, the results may no longer be Script Manager–compatible. Before replacing TextEdit routines, you should determine whether more than one script system is installed, and, if so, ensure that the replacement routine you provide is script-aware.

TECustomHook

The `TECustomHook` procedure replaces a default TextEdit hook routine with a customized routine and returns the address of the replaced routine.

```
PROCEDURE TECustomHook (which: TEIntHook; VAR addr: ProcPtr;
                        hTE: TEHandle);
```

`which` The hook whose default routine is to be replaced.

`addr` On input, the address of your customized procedure.

On output, the `addr` parameter contains the address of the routine that was previously installed in the field identified by the `which` parameter. This address is returned so that you can daisy-chain routines.

`hTE` A handle to the edit record to be modified.

DESCRIPTION

The `TECustomHook` procedure lets you alter the behavior of TextEdit to better suit your application's requirements and those of the script systems installed. If you replace a default hook routine with a customized version that you write in a high-level language, such as Pascal or C, you need to provide assembly-language glue code that utilizes the registers for your high-level language routine. The register contents for each of the hook routines are described in this section under "Assembly-Language Information."

The end-of-line hook, width measurement hook, new width measurement hook, text width measurement hook, draw hook. and hit test hook fields are hook fields in the TextEdit dispatch record. The `which` parameter identifies the hook whose default routine is to be replaced; you use the following constants to specify a value for this parameter.

Constant	Value	Description
`intEOLHook`	0	End-of-line hook
`intDrawHook`	1	Draw hook
`intWidthHook`	2	Width measurement hook
`intHitTestHook`	3	Hit test hook
`intNWidthHook`	6	New width measurement hook
`intTextWidthHook`	7	Text width measurement hook (low-memory global width measurement hook)

ASSEMBLY-LANGUAGE INFORMATION

The end-of-line hook, width measurement hook, new width measurement hook, text width measurement hook, draw hook, and hit test hook fields are hook fields in the TextEdit dispatch record. When you use `TECustomHook` to replace the default routines installed in these hook fields with customized ones, remember that the replacement routine must preserve all registers except those specified as containing return values.

End-of-Line Hook Registers

You specify the `intEOLHook` constant as the value of the `which` parameter to identify the end-of-line hook routine as the one you want to replace. This hook routine determines whether an incoming character is an end-of-line character. It tests the character, sets the appropriate status flags in the status register, and returns. The default action is to compare the character with $0D (a carriage return).

Registers on entry

D0 Character to compare (byte)

A3 Pointer to the edit record

A4 Locked handle to the edit record

Registers on exit

Z (Zero) flag in the status register clear if end-of-line character; set otherwise.

Draw Hook Registers

You specify the `intDrawHook` constant as the value of the `which` parameter to identify the drawing hook routine as the one you want to replace. The draw hook routine is called any time the components of a line are drawn. The appropriate font, style, and size characteristics have already been set into the current port by the time this routine is called. By default, the address of the QuickDraw `DrawText` procedure is stored in the draw hook field.

If your application is using outline (TrueType) fonts, TextEdit has also set the `preserveGlyph` parameter of the Font Manager's `SetPreserveGlyph` procedure to `FALSE`, so your customized hook procedure may need to reset this parameter if your application depends on it.

Registers on entry

D0 Offset into text (word)

D1 Length of text to draw (word)

A0 Pointer to text to draw

A3 Pointer to the edit record

A4 Locked handle to the edit record

Width Measurement Hook Registers

You specify the `intWidthHook` constant as the value of the `which` parameter to identify the width measurement hook routine as the one you want to replace. The width measurement hook routine measures portions of a line of text. It is used when only the Roman script system is installed and the field contains the address of a customized routine. It is supported for backward compatibility. In all other cases, when more than one script system is installed or when the width measurement hook field has not been customized, TextEdit calls the routine whose address is installed in the new width measurement hook field.

Registers on entry

D0 Length (in bytes) of text to measure (word)

D1 First byte of text to measure (word)

A0 Pointer to text buffer

A3 Pointer to the edit record

A4 Locked handle to the edit record

Registers on exit

D1 Pixel width of measured text (word)

Hit Test Hook Registers

You specify the `intHitTestHook` constant as the value of the `which` parameter to identify the hit test hook routine as the one you want to replace. The hit test hook routine determines the glyph position in a line given the pixel width from the left edge of the view rectangle. For system software before System 7, the default action is to call the QuickDraw `TextWidth` function to determine if the pixel width of the measured text is greater than the input width. If it is, then the hit test hook routine calls the QuickDraw `PixelToChar` function and returns. For System 7, the default action is to call the QuickDraw `PixelToChar` function.

Registers on entry

D0 Length of text block (style run) to measure (word)

D1 Pixel width from start of text block (word)

D2 Slop (should equal 0) (word)

A0 Pointer to start of text block

A3 Pointer to the edit record

A4 Locked handle to the edit record

Registers on exit

D0 Pixel width to character offset in text block (low word)

A Boolean that is `TRUE` if a character offset corresponding to the given pixel width was found (high word). Otherwise, `FALSE`.

D1 Character offset (word)

D2 A Boolean that is `TRUE` if the pixel width falls within the leading edge of the character (low word). Otherwise, `FALSE`.

TextEdit also uses the least significant bit of the high word. If the hit test hook routine calls `PixelToChar`, TextEdit sets this bit. If it uses `TextWidth`, TextEdit clears this bit. Your customized routine needs to do the same if you call either `PixelToChar` or `TextWidth` (high word). See the chapter "QuickDraw Text" in this book for more about these routines.

Note

In earlier versions of TextEdit, the value in register D2 on entry was not always used. If you daisy-chain in a routine and then call `the` hit test hook routine, D2 must be 0 (on entry). ◆

New Width Measurement Hook Registers

You specify the `intNWidthHook` constant as the value of the `which` parameter to identify the new width measurement hook routine as the one you want to replace. The new width measurement hook routine measures portions of a line of text when a non-Roman script system is installed. It is also used when only a Roman script system is installed and the width hook field does not contain the address of a customized routine.

The default procedure calls `CharToPixel` or `TextWidth`, depending on the primary line direction. The appropriate font, style, and size characteristics have already been set into the current graphics port by the time this routine is called.

The new width measurement hook routine is called to measure text for both Roman and non-Roman script systems, so make sure that your customized routine is script-aware.

Registers on entry

D0 Overall style run length, bounded by the line end (word)

D1 Offset position within style run on the current line (word)

D2 Slop (low word); direction flag (high word)

A0 Pointer to text buffer

A2 Pointer to current line start (from TextEdit's `lineStarts` array)

A3 Pointer to the edit record

A4 Locked handle to the edit record

Registers on exit

D1 Pixel width of measured text (word)

Text Width Measurement Hook Registers

You specify the `intTextWidthHook` constant as the value of the `which` parameter to identify the low-memory global text width measurement hook routine as the one you want to replace. By default, this hook field contains the address of the QuickDraw `TextWidth` function and provides a way to replace TextEdit's use of `TextWidth`. The new width measurement hook routine uses the routine whose address is installed in this field.

Registers on entry

D0 Length (in bytes) of text to be measured (word)

D1 Offset in text of first byte to measure (word)

A0 Pointer to text to measure

A3 Pointer to the edit record

A4 Locked handle to the edit record

Registers on exit

D1 Pixel width of measured text (word)

SEE ALSO

For more information about the `SetPreserveGlyph` procedure, see the chapter "Font Manager" in this book. For more information about `DrawText`, `TextWidth`, `PixelToChar`, and `CharToPixel`, see the chapter "QuickDraw Text" in this book.

SPECIAL CONSIDERATIONS

Take the following precautions if you replace a default routine:

- Before placing the address of your routine in the TextEdit dispatch record, strip the addresses, using the Operating System Utilities `StripAddress` function, to guarantee that your application is 32-bit clean. For more information, see *Inside Macintosh: Operating System Utilities*.
- Before replacing a TextEdit routine with a customized one, determine whether more than one script system is installed, and, if so, ensure that your customized routine accommodates all of the installed script systems. This avoids the problem of your customized routine producing results that are incompatible with the Script Manager.
- When you use assembly language, note that all registers must be preserved except those specified as containing return values. Registers A3 contains a pointer to the edit record and Register A4 contains a handle to it. You can obtain line start positions from the `lineStarts` array in the edit record. Register A5 is always valid. Refer to `TECustomHook` in the "TextEdit Reference" section for complete coverage of the register content requirements for all hook routines.

TESetWordBreak

The `TESetWordBreak` procedure installs the address of a customized word-selection break routine in the `wordBreak` field of the specified edit record.

```
PROCEDURE TESetWordBreak (wBrkProc: ProcPtr; hTE: TEHandle);
```

`wBrkProc`	A pointer to the customized word-selection break routine.
`hTE`	A handle to the edit record containing the `wordBreak` field to be modified.

DESCRIPTION

A word break routine determines which word is highlighted when the user double-clicks in the text. It also determines where TextEdit breaks the text at the end of a line. You can use `TESetWordBreak` to replace the default routine in the `wordBreak` field that is used for word selection and line breaking under certain circumstances. Whether or not TextEdit uses the word-selection break routine referenced by this field is determined by the algorithm implemented in the default `TEFindWord` routine. For a description of this algorithm, see "Customizing Word Selection" on page 2-61; this section also describes what to consider if you replace the `TEFindWord` hook routine.

When you replace the `wordBreak` field hook routine, your customized word break routine is used instead of the default one. The default routine breaks words at any character with an ASCII value of $20 or less (the space character or nonprinting control characters). Your routine can use a different value.

Before non-Roman script systems were supported, TextEdit used the word break hook routine installed in the `wordBreak` field for all word selection and line breaking.

However, in order to support both Roman and non-Roman script systems, TextEdit now uses the routine installed in the low-memory global variable `TEFindWord`. The default `TEFindWord` hook routine determines which hook TextEdit should use for word selection and line breaking—the `wordBreak` hook or the Text Utilities `FindWordBreaks` procedure—based on what script systems are installed and some other factors. You can replace the `TEFindWord` hook routine with a customized version.

ASSEMBLY-LANGUAGE INFORMATION

You must directly set the `wordBreak` field to point to your own word break routine; do not use the `TESetWordBreak` procedure.The registers for the word break routine must contain the following values.

Registers on entry

A0 Pointer to text

D0 Character position (word)

Register on exit

Z bit (zero flag)	Condition code:
status register	0 to break at specified character
	1 not to break there

If you replace `TEFindWord`, be careful to set the correct values in the appropriate registers. For `TEFindWord`, the registers are set on entry as specified below, and TextEdit depends on the registers being set at exit as specified below.

Registers on entry

D0 Current position (the value of `selStart` field in edit record) (word)

D2 Identifier of routine that called `FindWord`Breaks (word)

Identifier	Value	Explanation
`teWordSelect`	4	called for word selection
`teWordDrag`	8	called for extending word selection
`teFromFind`	12	called for determining new line breaks
`teFromRecal`	16	called for word breaking in line recalculation

A3 Pointer to the edit record

A4 Locked handle to the edit record

Registers on exit

D0 Word start (word)

D1 Word end (word)

SEE ALSO

For more information about the `FindWordBreaks` procedure, see the chapter "Text Utilities" in this book.

TESetClickLoop

The `TESetClickLoop` procedure installs in the `clikLoop` field of the edit record the address of the application-supplied click loop routine.

```
PROCEDURE TESetClickLoop (clickProc: ProcPtr; hTE: TEHandle);
```

`clickProc` A pointer to the customized click loop routine.

`hTE` A handle to the edit record whose `clikLoop` field is to be modified.

DESCRIPTION

The `TESetClickLoop` procedure lets you replace the default click loop routine. The `TEClick` procedure repeatedly calls the routine that the click loop field points to as long as the user holds down the mouse button within the text of the view rectangle. The default click loop routine scrolls only the text. However, you can provide a customized click loop procedure that scrolls the text and the scroll bars in tandem.

If automatic scrolling is enabled, the default click loop routine checks to see if the mouse has been dragged out of the view rectangle; if it has, the routine scrolls the text using `TEPinScroll`. (For more information, see "TEPinScroll" on page 2-93.) The amount by which `TEPinScroll` scrolls the text vertically is determined by the `lineHeight` field of the edit record for monostyled text and the `LHTable` for multistyled text.

ASSEMBLY-LANGUAGE INFORMATION

You can directly set the click loop (`clikLoop`) field; you don't need to use the `TESetClickLoop` procedure. Your routine should set register D0 to 1 and preserve register D2. Returning 0 in register D0 terminates `TEClick`.

Summary of TextEdit

Pascal Summary

Constants

```
CONST
      {Gestalt returned values}

      gestaltUndefSelectorErr {Systems before 6.0.4, multistyled TextEdit }
                              { on all hardware}
      gestaltTE1 = 1;         {System 6.0.4 Roman script system on }
                              { IIci-family hardware}
      gestaltTE2 = 2;         {Script Manager-compatible. System 6.0.4 }
                              { non-Roman script systems on all }
                              { IIci family hardware.New measuring hook }
                              { nWIDTHHook.}
      gestaltTE3 = 3;         {Script Manager–compatible. System 6.0.4 }
                              { non-Roman script systems on all non-IIci }
                              { family hardware.}
      gestaltTE4 = 4;         {Script Manager-compatible. System 6.0.5 on }
                              { all hardware. New TEFeatureFlag function.}
      gestaltTE5 = 5;         {Script Manager–compatible. System 7.0 on all }
                              {hardware}

      {alignment styles: new constant names for the align parameter }
      { of TESetAlignment and TETextBox}
      teFlushDefault =  0;          {flush according to system direction }
      teCenter       =  1;          {centered for all scripts }
      teFlushRight   = -1;          {flush right for all scripts}
      teFlushLeft    = -2;          {flush left for all scripts}

      {alignment styles; old constant names supported for }
      { backward-compatibility}
      teJustLeft           =  0;
      teJustCenter         =  1;
      teJustRight          = -1;
      teForceLeft          = -2;
```

```
{values for TEFeatureFlag feature parameter}
teFAutoScroll        = 0;        {automatic scrolling
teFAutoScr           = 0;        {old constant for automatic scrolling}
teFTextBuffering     = 1;        {text buffering}
teFOutlineHilite     = 2;        {outline highlighting}
teFInlineInput       = 3;        {inline input}
teFUseTextServices   = 4;        {use inline input service}

{values for TEFeatureFlag action parameter}
teBitClear           = 0;        {clear TEFeatureFlag features}
teBitSet             = 1;        {set TEFeatureFlag features}
teBitTest            = -1;       {test TEFeatureFlag features: return }
                                 { the current setting}

{selectors for identifying the routine that called TEFindWord }
teWordSelect         = 4;        {called for determining new }
                                 { line breaks}
teWordDrag           = 8;        {called for extending word selection}
teFromFind           = 12;       {called for word selection}
teFromRecal          = 16;       {called for word breaking in line }
                                 { recalculation}

{values for TESetStyle/TEContinuousStyle/TEReplaceStyle modes}
doFont               = 1;        {set font family number}
doFace               = 2;        {set character style}
doSize               = 4;        {set type size}
doColor              = 8;        {set color}
doAll                = 15;       {set all attributes}
addSize              = 16;       {adjust type size }
doToggle             = 32;       {toggle mode for TESetStyle}

{selectors for TECustomHook}
intEOLHook           = 0;        {end-of-line hook}
intDrawHook          = 1;        {drawing hook}
intWidthHook         = 2;        {width measurement hook}
intHitTestHook       = 3;     {hit-test hook}
intNWidthHook        = 6;     {nWIDTHHook measurement hook}
intTextWidthHook     = 7;     {TextWidth measurement hook}
```

Data Types

```
TYPE  TERec =
      RECORD
         destRect:   Rect;       {destination rectangle}
         viewRect:   Rect;       {view rectangle}
         selRect:    Rect;       {the selection rectangle}
         lineHeight: Integer;    {used for vertical spacing of lines}
         fontAscent: Integer;    {used for caret/highlighting position}
         selPoint:   Point;      {point selected with the mouse}
         selStart:   Integer;    {start of selection range}
         selEnd:     Integer;    {end of selection range}
         active:     Integer;    {set when record is }
                                 { activated/deactivated}
         wordBreak:  ProcPtr;    {word break hook}
         clikLoop:   ProcPtr;    {click loop hook}
         clickTime:  LongInt;    {used internally}
         clickLoc:   Integer;    {used internally}
         caretTime:  LongInt;    {used internally}
         caretState: Integer;    {used internally}
         just:       Integer;    {alignment of text}
         teLength:   Integer;    {length of text}
         hText:      Handle;     {handle to text to be edited}
         hDispatchRec: LongInt;  {handle to TextEdit }
                                 { dispatch record}
         clikStuff:  Integer;    {used internally}
         crOnly:     Integer;    {if <0, new line at Return only}
         txFont:     Integer;    {text font}
         txFace:     Style;      {character style; unpacked byte}
         txMode:     Integer;    {pen mode}
         txSize:     Integer;    {value indicates either a multistyled }
                                 { edit record or a font size}

         inPort:     GrafPtr;    {a pointer to the grafPort for this TERec}
         highHook:   ProcPtr;    {used for text highlighting }
         caretHook:  ProcPtr;    {used for the caret appearance}
         nLines:     Integer;    {number of lines}
         lineStarts: ARRAY[0..16000] OF Integer;
                                 {positions of line starts}
      END;

         TEPtr    = ^TERec;
         TEHandle = ^TEPtr;
```

```
   Chars= PACKED ARRAY[0..32000] OF CHAR;

   CharsHandle = ^CharsPtr;
   CharsPtr    = ^Chars;


TEStyleRec =
RECORD
   nRuns:      Integer;       {number of style runs}
   nStyles:    Integer;       {size of style table}
   styleTab:   STHandle;      {handle to style table}
   lhTab:      LHHandle;      {handle to line-height table}
   teRefCon:   LongInt;       {reserved for application use}
   nullStyle:  NullStHandle;  {handle to style set at }
                              { null selection}
   runs:       ARRAY [0..8000] OF StyleRun;
                              {ARRAY [0..8000] OF StyleRun}
END;

TEStylePtr = ^TEStyleRec;
TEStyleHandle = ^TEStylePtr;

StyleRun =
RECORD
   startChar:  Integer;       {starting character position}
   styleIndex: Integer;       {index in style table}
END;

STElement =
RECORD
   stCount:    Integer;       {number of runs in this style}
   stHeight:   Integer;       {line height}
   stAscent:   Integer;       {font ascent}
   stFont:     Integer;       {font (family) number}
   stFace:     Style;         {character style}
   stSize:     Integer;       {size in points}
   stColor:    RGBColor;      {absolute RGB color}
END;

STPtr    = ^TEStyleTable;
STHandle = ^STPtr;

TEStyleTable = ARRAY [0..1776] OF STElement;
```

```
LHElement =
RECORD
   lhHeight:      Integer;    {maximum height in line}
   lhAscent:      Integer;    {maximum ascent in line}
END;

LHPtr = ^LHTable;
LHHandle = ^LHPtr;

LHTable = ARRAY [0..8000] OF LHElement;

ScrpSTElement =
RECORD
   scrpStartChar:       LongInt;    {offset to start of style}
   scrpHeight:          Integer;    {line height}
   scrpAscent:          Integer;    {font ascent}
   scrpFont:            Integer;    {font (family) number}
   scrpFace:            Style;      {character style}
   scrpSize:            Integer;    {size in points}
   scrpColor:           RGBColor;   {absolute (RGB) color}
END;

ScrpStyl2Tab = ARRAY[0..1600] OF ScrpSTElement;

StScrpRec =
RECORD
   scrpNStyles:   Integer;          {number of styles in scrap}
   scrpStyleTab:  ScrpSTTable;      {table of styles for scrap}
END;

StScrpPtr = ^StScrpRec;
StScrpHandle = ^StScrpPtr;

NullStRec =
RECORD
   teReserved:          LongInt;    {reserved for future expansion}
   nullScrap:           StScrpHandle;  {handle to scrap style table}
END;

NullStPtr = ^NullStRec;
NullStHandle = ^NullStPtr;

TextStyle =
RECORD
```

```
      tsFont:  Integer;    {font (family) number}
      tsFace:  Style;      {character Style}
      tsSize:  Integer;    {size in points}
      tsColor: RGBColor;   {absolute (RGB) color}
   END;

   TextStylePtr = ^TextStyle;
   TextStyleHandle = ^TextStylePtr;

   TEIntHook = Integer;
```

Routines

Initializing TextEdit, Creating an Edit Record, and Disposing of an Edit Record

```
PROCEDURE TEInit;
FUNCTION TEStyleNew         (destRect: Rect; viewRect: Rect): TEHandle;
FUNCTION TENew              (destRect, viewRect: Rect): TEHandle;
PROCEDURE TEDispose         (hTE: TEHandle);
```

Activating and Deactivating an Edit Record

```
PROCEDURE TEActivate        (hTE: TEHandle);
PROCEDURE TEDeactivate      (hTE: TEHandle);
```

Setting and Getting an Edit Record's Text and Character Attribute Information

```
PROCEDURE TEKey             (key: Char; hTE: TEHandle);
PROCEDURE TESetText         (text: Ptr; length: LongInt; hTE: TEHandle);
FUNCTION TEGetText          (hTE: TEHandle): CharsHandle;
PROCEDURE TESetStyleHandle  (theHandle: TEStyleHandle; hTE: TEHandle);
FUNCTION TEGetStyleHandle   (hTE: TEHandle): TEStyleHandle;
```

Setting the Caret and Selection Range

```
PROCEDURE TEIdle            (hTE: TEHandle);
PROCEDURE TEClick           (pt: Point; extend: Boolean; hTE: TEHandle);
PROCEDURE TESetSelect       (selStart, selEnd: LongInt; hTE: TEHandle);
```

Displaying and Scrolling Text

```
PROCEDURE TESetAlignment    (align: Integer; hTE: TEHandle);
PROCEDURE TEUpdate          (rUpdate: Rect; hTE: TEHandle);
```

```
PROCEDURE TETextBox         (text: Ptr; length: LongInt;
                             box: Rect; just: Integer);
PROCEDURE TECalText         (hTE: TEHandle);
FUNCTION TEGetHeight        (endLine, startLine:LONGINT;
                             hTE: TEHandle): INTEGER;
PROCEDURE TEScroll          (dh,dv: Integer; hTE: TEHandle);
PROCEDURE TEPinScroll       (dh: INTEGER; dv: INTEGER; hTE: TEHandle);
PROCEDURE TEAutoView        (fAuto: Boolean; hTE: TEHandle);
PROCEDURE TESelView         (hTE: TEHandle);
```

Modifying the Text of an Edit Record

```
PROCEDURE TEDelete          (hTE: TEHandle);
PROCEDURE TEInsert          (text: Ptr; length: LongInt; hTE: TEHandle);
PROCEDURE TECut             (hTE: TEHandle);
PROCEDURE TECopy            (hTE: TEHandle);
PROCEDURE TEPaste           (hTE: TEHandle);
PROCEDURE TEStylePaste      (hTE: TEHandle);
FUNCTION TEToScrap: OSErr;
FUNCTION TEFromScrap: OSErr;
```

Managing the TextEdit Private Scrap

```
FUNCTION TEScrapHandle:     Handle;
FUNCTION TEGetScrapLength:  LongInt;
PROCEDURE TESetScrapLength  (length: LongInt);
```

Checking, Setting, and Replacing Styles

```
PROCEDURE TESetStyle        (mode: Integer; newStyle: TextStyle;
                             redraw: Boolean; hTE: TEHandle);
PROCEDURE TEReplaceStyle    (mode: INTEGER; oldStyle, newStyle: TextStyle;
                             redraw: BOOLEAN; hTE: TEHandle);
FUNCTION TEContinuousStyle  (VAR mode: Integer; VAR aStyle: TextStyle;
                             hTE: TEHandle) : Boolean;
PROCEDURE TEStyleInsert     (text: Ptr; length: LongInt;
                             hST: STScrpHandle; hTE: TEHandle);
FUNCTION TEGetStyleScrapHandle
                            (hTE: TEHandle): StyleScrpHandle;
PROCEDURE TEUseStyleScrap   (rangeStart: LongInt; rangeEnd: LongInt;
                             newStyles: STScrpHandle; redraw: Boolean;
                             hTE: TEHandle);
```

```
FUNCTION TENumStyles        (rangeStart: LongInt; rangeEnd: LongInt;
                             hTE: TEHandle): LongInt;
```

Using Byte Offsets and Corresponding Points

```
FUNCTION TEGetOffset        (pt: Point; hTE: TEHandle): Integer;
FUNCTION TEGetPoint         (offset: Integer; hTE: TEHandle): Point;
```

Additional TextEdit Features

```
FUNCTION TEFeatureFlag      (feature: Integer; action: Integer;
                             hTE: TEHandle) : Integer;
```

Customizing TextEdit

```
PROCEDURE TECustomHook      (which: TEIntHook; VAR addr: ProcPtr;
                             hTE: TEHandle);
PROCEDURE TESetWordBreak    (wBrkProc: ProcPtr; hTE: TEHandle);
PROCEDURE TESetClickLoop    (clickProc: ProcPtr; hTE: TEHandle);
```

C Summary

Constants

```
enum {
/*alignment styles of TESetAlignment and TETextBox*/
teFlushDefault    =  0,       /*flush according to system direction*/
teCenter          =  1,       /*centered for all scripts*/
teFlushRight      = –1,       /*flush right for all scripts*/
teFlushLeft       = -2,       /*flush left for all script*/

/*alignment styles; old names supported for backward-compatibility*/
teJustLeft        = 0,
teJustCenter      = 1,
teJustRight       = -1,
teForceLeft       = -2,

/*feature or bit definitions for TEFeatureFlag feature parameter*/
teFAutoScroll     = 0,        /*automatic scrolling*/
teFAutoScr        = 0,        /*old constant for automatic scrolling*/
teFTextBuffering  = 1,        /*text buffering*/
teFOutlineHilite  = 2,        /*outline highlighting*/
```

```
    teFInlineInput    = 3,        /*inline input*/
    teFUseTextServices   = 4,     /*use inline input service*/

    /* action for the new bit (un)set interface,TEFeatureFlag */
    teBitClear        = 0,        /*set the selector bit*/
    teBitSet          = 1,
    };
    enum {
    teBitTest         = -1, /*no change; just return the current setting*/
    teBitClear        = 0,
    teBitSet          = 1,  /*set the selector bit*/
    teBitTest         = -1, /*no change; just return the current setting*/

    /*constants for identifying the routine that called TEFindWord */
    teWordSelect      = 4,     /*clickExpand to select word*/
    teWordDrag        = 8,     /*clickExpand for extending word selection*/
    teFromFind        = 12,    /*FindLine called it ($0C)*/
    teFromRecal       = 16,
       /*called for word breaking in line recalculation*/

    /*values for TESetStyle/TEContinuousStyle/TEReplaceStyle modes*/
    doFont   = 1,     /*set font family number*/
    doFace   = 2,     /*set character style*/
    doSize   = 4,     /*set type size*/
    doColor  = 8,     /*set color*/
    doAll    = 15,    /*set all attributes*/
    addSize  = 16,    /*adjust type size*/
};
enum {
    doToggle = 32,    /*toggle mode for TESetStyle*/

    /*selectors for TECustomHook*/
    intEOLHook     = 0,     /*end-of-line hook*/
    intDrawHook    = 1,     /*drawing hook*/
    intWidthHook   = 2,     /*width measurement hook*/
    intHitTestHook = 3,     /*hit-test hook*/
    intNWidthHook  = 6,     /*nWIDTHHook measurement hook*/
    intTextWidthHook  = 7,  /*TextWidth measurement hook*/
};
```

Types

```
typedef pascal Boolean (*WordBreakProcPtr)(Ptr text, short charPos);
typedef pascal Boolean (*ClikLoopProcPtr)(void);

struct TERec {
 Rect destRect;      /*destination rectangle*/
 Rect viewRect;      /*view rectangle*/
 Rect selRect;       /*the selection rectangle*/
 short lineHeight;   /*used for vertical spacing of lines*/
 short fontAscent;   /*used for caret/highlighting position*/
 Point selPoint;     /*point selected with the mouse*/
 short selStart;     /*start of selection range*/
 short selEnd;       /*end of selection range*/
 short active;       /*set when record is activated/deactivated*/
 WordBreakProcPtr wordBreak;  /*word break hook*/
 ClikLoopProcPtr clikLoop;    /*click loop hook*/
 long clickTime;     /*used internally*/
 short clickLoc;     /*used internally*/
 long caretTime;     /*used internally*/
 short caretState;   /*used internally*/
 short just;         /*alignment of text*/
 short teLength;     /*length of text*/
 Handle hText;       /*handle to text to be edited*/
 long hDispatchRec;  /*handle to TextEdit dispatch record*/
 short clikStuff;    /*used internally*/
 short crOnly;       /*if <0, new line at Return only*/
 short txFont;       /*text font*/
 Style txFace;       /*character style; unpacked byte*/
 char filler;
 short txMode;       /*pen mode*/
 short txSize;
   /*value indicates either a multistyled edit record or a font size*/
 GrafPtr inPort;     /*a pointer to the grafPort for this TERec*/
 ProcPtr highHook;   /*used for text highlighting and the caret appearance*/
 ProcPts caretHook;  /*used from assembly language*/
 short nLines;       /*number of lines*/
 short lineStarts[16000];/*positions of line starts*/
};

 typedef struct TERec TERec;
 typedef TERec *TEPtr, **TEHandle;
```

```
typedef char Chars[32001];
typedef char *CharsPtr, **CharsHandle;

struct StyleRun {
  short startChar;  /*starting character position*/
  short styleIndex; /*index in style table*/
};

typedef struct StyleRun StyleRun;

struct STElement {
  short stCount;    /*number of runs in this style*/
  short stHeight;   /*line height*/
  short stAscent;   /*font ascent*/
  short stFont;     /*font family number*/
  Style stFace;     /*character style*/
  char filler;      /*stFace is unpacked byte*/
  short stSize;     /*size in points*/
  RGBColor stColor; /*absolute Red Green Blue color*/
};

typedef struct STElement STElement;

typedef STElement TEStyleTable[1777], *STPtr, **STHandle;

struct LHElement {
  short lhHeight;   /*maximum height in line*/
  short lhAscent;   /*maximum ascent in line*/
};

typedef struct LHElement LHElement;

typedef LHElement LHTable[8001], *LHPtr, **LHHandle;
  /* ARRAY [0..8000] OF LHElement */

struct ScrpSTElement {
  long scrpStartChar; /*starting character position*/
  short scrpHeight;    /*line height*/
  short scrpAscent;
  short scrpFont;
  Style scrpFace;   /*unpacked byte*/
  char filler;      /*scrpFace is unpacked byte*/
  short scrpSize;
```

```
  RGBColor scrpColor;
};

typedef struct ScrpSTElement ScrpSTElement;

typedef ScrpSTElement ScrpSTTable[1601]; /*ARRAY [0..1600] OF ScrpSTElement*/

struct STScrpRec {
  short scrpNStyle;          /*number of styles in scrap*/
  ScrpSTTable scrpStyleTab;  /*table of style for scrap*/
};

typedef struct StScrpRec StScrpRec;
typedef StScrpRec *StScrpPtr, **StScrpHandle;

struct NullSTRec {
  long teReserved;           /*reserved for future expansion*/
  StScrpHandle nullScrap;    /*handle to scrap style table*/
};

typedef struct NullStRec NullSTRec;
typedef NullStRec *NullStPtr, **NullStHandle;

struct TEStyleRec {
  short nRuns;         /*number of style runs*/
  short nStyles;       /*size of style table*/
  STHandle styleTab;   /*handle to style table*/
  LHHandle lhTab;      /*handle to line-height table*/
  long teRefCon;       /*reserved for application use*/
  NullSTHandle nullStyle; /*handle to style set at null selection*/
  StyleRun runs [8001];   /*ARRAY [0..8000] OF StyleRun*/
};

typedef struct TEStyleRec TEStyleRec;
typedef TEStyleRec *TEStylePtr, **TEStyleHandle;

struct TextStyle {
  short tsFont;     /*font family number*/
  Style tsFace;     /*character Style*/
  char filler;      /*tsFace is unpacked byte*/
  short tsSize;     /*size in points*/
  RGBColor tsColor; /*absolute red, green, and blue color*/
};
```

```
typedef struct TextStyle TextStyle;
typedef TextStyle *TextStylePtr, **TextStyleHandle;

typedef short TEIntHook;
```

Routines

Initializing TextEdit, Creating an Edit Record, and Disposing of an Edit Record

```
pascal void TEInit          (void);
pascal TEHandle TEStyleNew  (const Rect *destRect,const Rect *viewRect);
pascal TEHandle TENew       (const Rect *destRect, const Rect *viewRect);
pascal void TEDispose       (TEHandle hTE);
```

Activating and Deactivating an Edit Record

```
pascal void TEActivate      (TEHandle hTE);
pascal void TEDeactivate    (TEHandle hTE);
```

Setting and Getting an Edit Record's Text and Character Attribute Information

```
pascal void TEKey           (short key, TEHandle hTE);
pascal void TESetText       (const void *text, long length, TEHandle hTE);
pascal CharsHandle TEGetText
                            (TEHandle hTE);
pascal void TESetStyleHandle
                            (TEStyleHandle theHandle, TEHandle hTE);
pascal TEStyleHandle TEGetStyleHandle
                            (TEHandle hTE);
```

Setting the Caret and Selection Range

```
pascal void TEIdle          (TEHandle hTE);
pascal void TEClick         (Point pt, Boolean fExtend, TEHandle hTE);
pascal void TESetSelect     (long selStart, long selEnd, TEHandle hTE);
```

Displaying and Scrolling Text

```
pascal void TESetAlignment  (short just, TEHandle hTE);
pascal void TEUpdate        (const Rect *rUpdate, TEHandle hTE);
pascal void TETextBox       (const void *text, long length, const Rect *box,
                             short just);
pascal void TECalText       (TEHandle hTE);
pascal long TEGetHeight     (long endLine, long startLine, TEHandle hTE);
```

```
pascal void TEScroll        (short dh, short dv, TEHandle hTE);
pascal void TEPinScroll     (short dh, short dv, TEHandle hTE);
pascal void TEAutoView      (Boolean fAuto, TEHandle hTE);
pascal void TESelView       (TEHandle hTE);
```

Modifying the Text of an Edit Record

```
pascal void TEDelete        (TEHandle hTE);
pascal void TEInsert        (const void *text, long length, TEHandle hTE);
pascal void TECut           (TEHandle hTE);
pascal void TECopy          (TEHandle hTE);
pascal void TEPaste         (TEHandle hTE);
pascal void TEStylePaste    (TEHandle hTE);
pascal OSErr TEToScrap      (void);
pascal OSErr TEFromScrap    (void);
```

Managing the TextEdit Private Scrap

```
#define TEScrapHandle()     (* (Handle*) 0xAB4)
#define TEGetScrapLength()  ((long) * (unsigned short *) 0x0AB0)
pascal void TESetScrapLength
                            (long length);
```

Checking, Setting, and Replacing Styles

```
pascal void TESetStyle      (short mode, const TextStyle *newStyle, Boolean
                             fRedraw, TEHandle hTE);
pascal void TEReplaceStyle  (short mode, onst TextStyle *oldStyle,
                             const TextStyle *newStyle, Boolean fRedraw,
                             TEHandle hTE);
pascal Boolean TEContinuousStyle
                            (short *mode, TextStyle *aStyle, TEHandle hTE);
pascal void TEStyleInsert   (const void *text, long length,
                             STScrpHandle hSt, TEHandle hTE);
pascal StScrpHandle TEGetStyleScrapHandle
                            (TEHandle hTE);
pascal void TEUseStyleScrap (long rangeStart, long rangeEnd, StScrpHandle
                             newStyles, Boolean fRedraw, TEHandle hTE);
pascal long TENumStyles     (long rangeStart, long rangeEnd, TEHandle hTE);
```

Using Byte Offsets and Corresponding Points

```
pascal short TEGetOffset    (Point pt, TEHandle hTE);
pascal Point TEGetPoint     (short offset, TEHandle hTE);
```

Additional TextEdit Features

```
pascal short TEFeatureFlag  (short feature, short action, TEHandle hTE);
```

Customizing TextEdit

```
pascal void TECustomHook    (TEIntHook which, ProcPtr *addr, TEHandle hTE);
pascal void TESetWordBreak  (WordBreakProcPtr wBrkProc, TEHandle hTE);
pascal void TESetClickLoop  (ClikLoopProcPtr clickProc, TEHandle hTE);
```

Assembly-Language Summary

Trap Macros

Trap Macro Names

Pascal name	Trap macro name
TEContinuousStyle	_TEContinuousStyle
TEUseStyleScrap	_TEUseStyleScrap
TECustomHook	_TECustomHook
TENumStyles	_TENumStyles
TEFeatureFlag	_TEFeatureFlag
TEStylePaste	_TEStylePaste
TEReplaceStyle	_TEReplaceStyle
TEGetStyleHandle	_TEGetStyleHandle
TESetStyleHandle	_TESetStyleHandle
TEReplaceStyle	_TEReplaceStyle
TEGetStyleScrap	_TEGetStyleScrap
TEGetStyleHandle	_TEGetStyleHandle
TEGetStyleScrapHandle	_TEGetStyleScrapHandle
TEStyleInsert	_TEStyleInsert
TEGetPoint	_TEGetPoint
TEGetHeight	_TEGetHeight

Global Variables

`WordRedraw`	Used for line calculations to determine how much of a line must be redrawn after a character is entered.
`TEFindWord`	TextEdit's word selection and line breaking routine.
`TERecal`	The address of the routine called by `TECalText` to recalculate the line starts and set the first and last characters that need to be redrawn.
`TEDoText`	The address of a multi-purpose text editing routine used to display, highlight, and hit-test characters, and position the pen to draw the caret.
`TEScrpHandle`	A handle to the TextEdit private scrap.
`TEScrapLength`	The size of the TextEdit scrap in bytes.