17

Version 1.1

TEXT AND TEXTEDIT

Includes Demonstration Programs Text1 and Text2

Introduction

The subject of text on the Macintosh is quite a complex matter, involving as it does QuickDraw, TextEdit, the Font Manager, the Text Utilities, the Script Manager, the Text Services Manager, the Resource Manager, keyboard resources, and international resources. Part of that complexity arises from the fact that the system software supports many different writing systems, including Roman, Chinese, Japanese, Hebrew, and Arabic.[1]

Text on the Macintosh was touched on briefly at Chapter 10 — Basic QuickDraw, which included descriptions of QuickDraw routines used for drawing text and for setting the font, style, size, and transfer mode. In addition, Chapter 13 — Printing contained a brief treatment of considerations applying to the printing of text.

This chapter addresses:

- TextEdit, which is a collection of routines and data structures you can use to provide your application with basic text editing and formatting capabilities.
- The formatting and display of dates, times, and numbers.

Before addressing those particular subjects, however, a brief overview of various closely related matters is appropriate.

More on Text

Characters, Character Sets and Codes, Glyphs, Typefaces, Styles, Fonts and Font Families

Characters and Character Sets and Codes

A **character** is a symbol which represents the concept of, for example, a lowercase "b", the number "2" or the arithmetic operator "+". A collection of characters is called a **character set**. Individual characters within a character set are identified by a **character code**.

The **Apple Standard Roman character set** is the fundamental character set for the Macintosh computer. It uses all character codes from `0x00` to `0xFF`, and includes the uppercase versions of all of the lowercase accented Roman characters, a number of symbols, and other forms (see Fig 1). The Standard Roman

[1]Some of the information in this chapter is valid only in the case of the Roman writing system.

character set is an extended version of the **ASCII character set**, which uses character codes from 0x00 to 0x7F only, and which is highlighted at Fig 1.

	0x	1x	2x	3x	4x	5x	6x	7x	8x	9x	Ax	Bx	Cx	Dx	Ex	Fx
x0	nul	dle	sp	0	@	P	`	p	Ä	ê	†	∞	¿	–	‡	
x1	soh	DC1	!	1	A	Q	a	q	Å	ë	°	±	¡	—	·	Ò
x2	stx	DC2	"	2	B	R	b	r	Ç	í	¢	≤	¬	“	‚	Ú
x3	etx	DC3	#	3	C	S	c	s	É	ì	£	≤	√	”	„	Û
x4	eot	DC4	$	4	D	T	d	t	Ñ	î	§	¥	ƒ	‘	‰	Ù
x5	enq	nak	%	5	E	U	e	u	Ö	ï	•	µ	≈	’	Â	ı
x6	ack	syn	&	6	F	V	f	v	Ü	ñ	¶	∂	Δ	÷	Ê	ˆ
x7	bel	etb	'	7	G	W	g	w	á	ó	ß	∑	«	#	Á	˜
x8	bs	can	(	8	H	X	h	x	à	ò	®	∏	»	ÿ	Ë	¯
x9	ht	em	)	9	I	Y	i	y	â	ô	©	π	…	Ÿ	È	˘
xA	lf	sub	*	:	J	Z	j	z	ä	ö	™	∫		⁄	Í	˙
xB	vt	esc	+	;	K	[	k	{	ã	õ	´	ª	À	¤	Î	°
xC	ff	fs	,	<	L	\	l	|	å	ú	¨	º	Ã	‹	Ï	¸
xD	cr	gs	-	=	M	]	m	}	ç	ù	≠	Ω	Õ	›	Ì	˝
xE	so	rs	.	>	N	^	n	~	é	û	Æ	æ	Œ	fi	Ó	˛
xF	si	us	/	?	O	_	o	del	è	ü	Ø	ø	œ	fl	Ô	ˇ

CONTROL CODES	ROMAN CHARACTERS	SCRIPT-SPECIFIC CHARACTERS
LOW ASCII RANGE		HIGH ASCII RANGE

FIG 1 - THE STANDARD ROMAN CHARACTER SET

Glyphs

You never see a character on a display device. What you see on a display device is a **glyph**, which is the visual representation of a character. In other words, a glyph is the exact shape by which a character is represented. A specific character can be represented by many different shapes (that is, glyphs).

The Font Manager uses two types of glyphs: **bitmapped glyphs** and glyphs from **outline fonts**. A bitmapped glyph is a bitmap designed at a fixed size for a particular display device. A glyph from an outline font is a model of how the glyph should look. The "outline" is a mathematical description of the glyph in terms of lines and curves, and is used by the Font Manager to create bitmaps at any size for any display device.

Typefaces

If all glyphs for a particular character set share certain design characteristics, they form a **typeface**, which is a distinctively designed collection of glyphs. Each typeface has its own name, such as New York, Geneva, or Times. The same typeface can be used with different hardware, such as typesetting machines, monitors, and laser printers.

Styles

A **style** is a specific variation in the appearance of a glyph which can be applied consistently to all glyphs in a typeface. Styles available on the Macintosh include plain, bold, italic, underline, outline, shadow, condensed, and extended. QuickDraw can add styles to bitmaps, or a font designer can design a font in a specific style (for example, Courier Bold).

Fonts and Font Families

A **font** refers to a complete set of glyphs in a specific typeface and style — and, in the case of bitmapped fonts, a specific size. All fonts have a font name, which is stored in a string such as "Geneva" or "New York". The font name is usually the same name as the typeface from which it was derived. If a font is not in the plain style, its style becomes part of the font name, for example "Palatino Bold".

Fonts on the Macintosh are resources. The resource types are as follows:

- Bitmapped fonts are fonts of the `'FONT'` resource type (the original resource type for fonts) and the bitmapped font (`'NFNT'`) resource type (introduced with the 128K ROM). These resources provide a separate bitmap for each glyph in each size and style.

- Outline fonts are fonts of the outline font (`'sfnt'`) resource type which consist of glyphs in a particular typeface and style with no size restriction. The outline font resource type emerged at the time of the addition of TrueType outline font support with System 7.

When multiple fonts of the same typeface are present in the system software, the Font Manager groups them into **font families** of the font family (`'FOND'`) resource type. (This resource type was introduced with the 128K ROM.) A **font family ID** is the resource ID for a font family. Because there are so many font families available for the Macintosh, many families have the same ID.[2]

As an aside, most (though not all) fonts assign glyphs to character codes 0x20 to 0x7F which visually define the characters associated with those codes.[3] However, there are differences in the glyphs assigned to the high-ASCII range. Indeed, some fonts do not actually include glyphs for all, or part, of the high-ASCII range.

Font Measurements

Monospaced and Proportional Fonts. Fonts are either **monospaced** or **proportional**. All glyphs in a monospaced font are the same width. The glyphs in a proportional font have different widths, "m" being wider than "i", for example.

Base Line, Ascent Line and Descent Line. Most glyphs in a font sit on an imaginary line called the **base line**. The **ascent line** is an imaginary horizontal line chosen by the font's designer which corresponds approximately with the tops of the uppercase letters of the font. The **descent line** is an imaginary line which usually corresponds to the bottom of the descenders (the tails of glyphs like "p" and "g").

Glyph Origin and Advance Width. QuickDraw begins drawing a glyph at the **glyph origin**. There is some white space between the glyph origin and the beginning of the glyph called the **left side bearing**. The **advance width** is the full horizontal measurement of a glyph as measured from its glyph origin to the glyph origin of the next glyph.

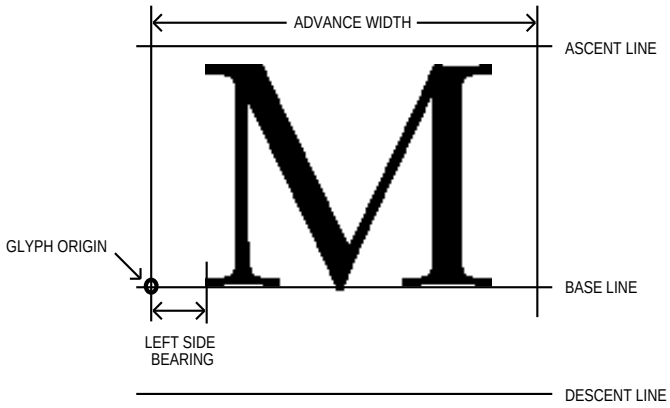

FIG 2 - FONT MEASUREMENT TERMS

Font Size. Font size indicates the size of the font's glyphs as measured from the base line of one line of the text to the base line of the next line. Font size is measured in **points** (1/72 of an inch). The size of a font is often, but not always, the sum of the ascent, descent and **leading** (pronounced "ledding")

[2]This is the reason why your application should refer to fonts by name and not by number when it stores font references in a document.

[3]Fonts such as Zapf Dingbats assign glyphs of pictorial symbols to this range. as well as the low-ASCII range.

values for a font. (The leading value is the amount of blank vertical space between the descent line of one line and the ascent line of the next line.)

The base line, ascent line, descent line, and glyph origin are illustrated at Fig 2.

System Font and Application Font

Macintosh system software recognises the following two special fonts, which should always be present:

- The **system font**, which is used for menus, dialog boxes, and other messages to the user from the Finder and the Operating System. The system font is 12-point Chicago, whose font family ID is 0.

- The **application font**, which is the suggested default font for use by monostyled TextEdit and by applications which do not support user selection of fonts. The application font is 12-point Geneva, whose font family ID is 3.

The system font and application font have **special font designators**. The system font designator is 0 and the application font designator is 1. These special designators are *not* actual font family resource ID numbers and cannot be used as such in Resource Manager calls; however, they can be used in place of the font family ID in the `txFont` field of the graphics port and in text-related calls that take a font family ID. The system maps the special designators to the actual font family IDs.

The Font Manager and QuickDraw

The Font Manager keeps track of all fonts available to an application and supports QuickDraw by providing the character bitmaps that QuickDraw needs. If QuickDraw requests a typeface that is not represented in the available fonts, the Font Manager substitutes one that is. Where necessary, QuickDraw scales the font to the requested size and applies the specified style.

Aspects of Text Editing - Caret Position, Text Offsets, Selection Range, Insertion Point, and Highlighting

Caret Position and Text Offset

In the world of text editing, the **caret** is defined as the blinking bar which marks the insertion point of text and the **cursor** is the arrow, I-beam or other icon that moves with the mouse.

A caret position is a location on the screen which corresponds to an insertion point in memory. A caret position is always *between* glyphs on the screen. The caret is always positioned on the leading edge of the glyph corresponding to the character at the insertion point in memory. When a new character is inserted, it displaces the character at the insertion point, shifting it and all subsequent characters in the buffer forward by one position.

The relationship between caret position, insertion point and offset is illustrated at Fig 3.

FIG 3 - CARET POSITION AND INSERTION POINT

Converting Screen Position to Text Offset

A mouse-down event can occur anywhere within the area of a glyph, but the caret position which is derived from that event must be an infinitesimally thin line falling between the two glyphs.

As shown at Fig 4, a line of displayed glyphs is divided into a series of **mouse-down regions**. A mouse-down region is a screen area within which any mouse click will yield the same caret position. It extends from the centre of one glyph to the centre of the next glyph (except at the ends of lines).

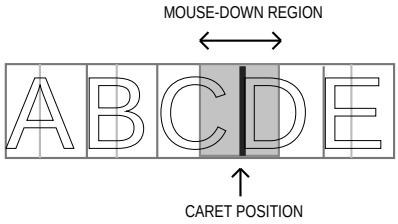

FIG 4 - INTERPRETING CARET POSITION FROM A MOUSE-DOWN EVENT

Selection Range and Insertion Points

The **selection range** is the sequence of zero or more characters, contiguous in memory, where the next editing operation is to occur. A selection range of zero characters is called an **insertion point**.

Highlighting

A selection range is typically marked by **highlighting**, that is, by drawing the glyphs in inverse video or with a coloured background. The limits of highlighting rectangles are measured in terms of caret position. For example, if the characters B, C, and D at Fig 3 were highlighted, the highlighting would extend from the leading edge of B (offset = 1) to the leading edge of E (offset = 4).

Outline Highlighting. Outline highlighting is the "framing" of text in the selection range in an inactive window. If there is no selection range, a grey, unblinking caret is displayed. By default, outline highlighting is disabled.

Keyboards and Text

Each keypress on a particular keyboard generates a value called a **raw key code**. The keyboard driver which handles the keypress uses the **key-map** (`'KMAP'`) **resource** to map the raw key code to a keyboard-independent **virtual key code**. It then uses the Event Manager and the **keyboard layout** (`'KCHR'`) **resource** to convert a virtual keycode into a character code. The character code is passed to your application in the event record generated by the keypress.

Introduction to TextEdit

TextEdit is a collection of routines and data structures which give your application basic text formatting and editing capabilities. Its routines can be used in applications such as spreadsheets, on-line (data entry) forms, simple text editors, and drawing and painting programs with simple text-editing features. TextEdit relies on Script Manager, QuickDraw, and Text Utilities routines to handle text correctly, eliminating the need for your application to call these routines directly.

TextEdit was originally designed to handle editable text items in dialog boxes and other parts of the system software. It was subsequently enhanced to support some of the cumbersome tasks that a text processor needs to perform. That said, it was never intended to manipulate lengthy text documents in excess of 32 KB. Indeed, the limit for documents created by TextEdit is 32,767 characters.

Editing Tasks Performed by TextEdit

The fundamental editing tasks which TextEdit can perform for your application are as follows:

- Selection of text by clicking and dragging the mouse.
- Double-clicking to select words.

- Extending or shortening selection ranges by Shift-clicking.
- Highlighting the current text selection, or displaying a blinking vertical bar (the caret) at the insertion point.
- Line breaking, that is, preventing a word from being split inappropriately between lines when text is drawn.
- Cutting, copying, and pasting within and between applications.
- Managing the use of more than one font, size, colour and stylistic variation from character to character.

Thus, if you do not need to manipulate large files and do not need extensive formatting capabilities, TextEdit is a convenient alternative to writing your own specialised text processing routines.

TextEdit Options

You can use TextEdit at different levels of complexity.

Using TextEdit Indirectly

For the simplest level of text handling (that is, in dialog boxes), you need not even call TextEdit directly but rather use the Dialog Manager. The Dialog Manager, in turn, calls TextEdit to edit and display text.

Displaying Static Text

If you simply want to display one or more lines of static (non-editable) text, you can call `TETextBox`, which draws your text in the location you specify. `TETextBox` may be used to display text that you cannot edit. You do not need to create an edit record (see below) because `TETextBox` creates its own edit record. `TETextBox` draws the text in a rectangle whose size you specify in the coordinates of the current graphics port. Using the following constants, you can specify how text is aligned in the box:

Constant	Description
`teFlushDefault`	Default alignment according to primary line direction of the script system. (Left for Roman script system.)
`teCenter`	Centre alignment.
`teFlushRight`	Right alignment.
`teFlushLeft`	Left alignment.

Text Handling — Monostyled Text

If your application requires very basic text handling in a single typeface, style, and size, you probably only need **monostyled TextEdit**. You can use monostyled TextEdit with the application font (if you do not allow the user to select the font) or with any single available font (if you do allow user selection).

Text Handling — Multistyled Text

If your application requires a somewhat higher level of text handling (allowing the user to change typeface, style, and size within the document, for example), you must use **multistyled TextEdit**.

Caret Position and Movement in TextEdit

TextEdit marks the position in the displayed text where the next editing operation will occur with the caret. When TextEdit pastes text into a record, it positions the caret after the newly pasted text. When the user presses the Up Arrow key or the Down Arrow key, the caret moves up or down one line respectively. When the caret is on the first line of an edit record, and the user presses the Up Arrow key, TextEdit moves the caret to the beginning of text on that line. When the caret is 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.[4]

If spaces at the end of a text line extend beyond the view rectangle (see below), 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 the `clikStuff` field 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 the 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.

Automatic Scrolling

One way for the user is to select large blocks of text is to click in the text and, holding the mouse button down, drag the cursor above, below, left of, or right of TextEdit's view rectangle. While the mouse button remains down, and provided that your application has enabled automatic scrolling, TextEdit continually calls its **click loop procedure** to automatically scroll the text.

Although TextEdit's default click loop routine automatically scrolls the text, it cannot adjust the scroll box position in an application's scrollbars to follow up the scrolling. The default click loop procedure can, however, be replaced with an application-defined click loop procedure which accommodates scroll bars.

TextEdit Private, Null, and Style Scraps

Internally, TextEdit uses three scrap areas, namely, the **private scrap**, the **null scrap**, and the **style scrap**. The null scrap and the style scrap apply only to multistyled TextEdit.

Private Scrap

The private scrap, which belongs to your application, is used for all cut, copy, and paste activity. When the text is multistyled, TextEdit also copies the text to the Scrap Manager's desk scrap.

Null Scrap

The null scrap is used to store **character attribute** information[5] associated with a null selection (that is, an insertion point) or text that is deleted when the user backspaces over it.

Character attribute information is retained in the null scrap until it is used, that is, when it is 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 which deal with multistyled text check the null scrap for character attribute information and, if there is any, apply it to the newly inserted text when character attributes for that text are not available.

TextEdit creates and initialises a null scrap for a multistyled edit record when an application creates the edit record. The null scrap remains throughout the life of the edit record, being disposed of when the application disposes of the edit record and release the memory associated with it.

Style Scrap

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 as follows:

- 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.

---

[4]TextEdit does not support the use of modifier keys, such as the Shift key, in conjunction with the arrow keys.

[5]The font, style, size, and colour aspects of text are collectively referred to as **character attributes**.

- You can create a style scrap record and store character attribute information in it to be applied to inserted text.

Text Alignment

The term **alignment** means the horizontal alignment of lines of text with respect to the left and right edges of the text area. Alignment can be left-aligned, right-aligned, centred, or justified (that is, aligned with both the left and right edges of the text area). Justification is achieved by spreading or compressing text to fit a given line width. TextEdit supports left-aligned, right-aligned and centred alignments.

Customising TextEdit

TextEdit may be customised by replacing the end-of-line, drawing, width-measuring, and hit test default hook routines using `TECustomHook`. You can also customise word selection, automatic scrolling, and how to determine the length of a line in order to justify it.

Primary TextEdit Data Structures

The primary data structures used by TextEdit are the **edit record** and the **dispatch record**. Additional data structures are associated with multistyled TextEdit. This section describes the primary data structures only.

The Edit Record

The edit record is the principal data structure used by TextEdit. The structure of the edit record is the same regardless of whether the text is monostyled or multistyled, although some fields are used differently for multistyled edit records. The edit record is as follows:

```
struct TERec
{
   Rect      destRect;      // Destination rectangle.
   Rect      viewRect;      // View rectangle.
   Rect      selRect;       // Selection rectangle.
   short     lineHeight;    // Used for vertical spacing of lines.
   short     fontAscent;    // Used for caret size and highlight rectangle calculation.
   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 or deactivated.
   WordBreakUPP  wordBreak; // Word break procedure.
   ClikLoopUPP   clikLoop;  // Click loop procedure.
   long      clickTime;     // (Used internally.)
   short     clickLoc;      // (Used internally.)
   long      caretTime;     // (Used internally.)
   short     caretState;    // (Used internally.)
   short     just;          // Text alignment.
   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.        // If multistyled edit record (txSize = -1),
   Style     txFace;  // Character style.  // these bytes are used as a handle
   SInt8     filler;  //                   // to a style record (TEStyleHandle).
   short     txMode;        // Pen mode.
   short     txSize;  // Value indicates font size or, if -1, multistyled edit record.
   GrafPtr   inPort;        // Pointer to grafPort for this edit record.
   HighHookUPP   highHook;     // Used for text highlighting, caret appearance.
   CaretHookUPP  caretHook;    // Used from assembly language.
   short     nLines;           // Number of lines.
   short     lineStarts[16001]; // Positions of line starts.
};

typedef struct TERec TERec,*TEPtr,**TEHandle;
```

Field Descriptions

`destRect` Destination rectangle, in local coordinates.

`viewRect` View rectangle, in local coordinates.

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 text is drawn and the **view rectangle** is that portion of the window within which the text is actually displayed. Fig 5 illustrates the relationship between the destination rectangle and the view rectangle.[6]

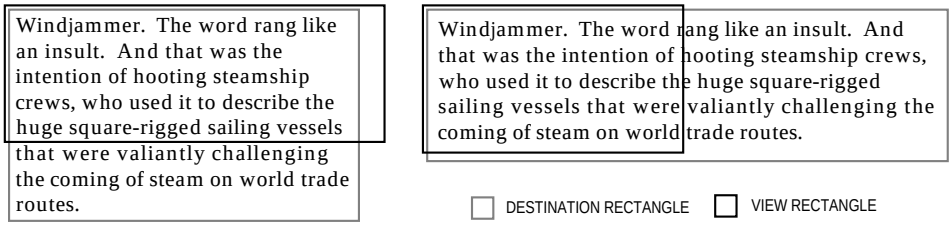

FIG 5 - DESTINATION AND VIEW RECTANGLES

Editing operations may shorten or lengthen 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, and its top determines the position of the first line.

The destination rectangle is central to the matter of scrolling text. When text is scrolled downwards, for example, you can think of the destination rectangle as being moved upwards through the view rectangle.

`selRect` The selection rectangle boundaries, in local coordinates.

`lineHeight` The vertical spacing of lines of text. In a monostyled edit record, the value specifies the fixed vertical distance from the ascent line of one line of text to the ascent line of the next line.

Multistyled Edit Record. In a multistyled edit record, this field is set to -1, which indicates that line heights are calculated independently for each line based on the maximum value for any individual character on that line.

`fontAscent` The font ascent line. For monostyled text, the value specifies how far above the baseline the pen is positioned to begin drawing the caret or highlighting. (For single-spaced text, this is the height of the text in pixels.)

Multistyled Edit Record. In a multistyled edit record, this field is set to -1, which indicates that the font ascent is calculated independently for each line based on the maximum value for any individual character on that line.

`selPoint` The point selected with the mouse, in the local coordinates of the current graphics port.

`selStart` The byte offset of the beginning of the selection range. When you create an edit record, TextEdit initialises this field to 0. (Byte offset 0 refers to the first byte in the text buffer.)

`selEnd` The byte offset of the end of the selection range. (Note that, to include that byte, this value must be one greater than the position of the last byte offset of the text.) When you

[6]Note that 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.

create an edit record, TextEdit initialises this field to 0. With both `selStart` and `selEnd` initialised to 0, the insertion point is placed at the beginning of the text.

`active` Set by TextEdit when an edit record is activated using `TEActivate` and then reset when the edit record is rendered inactive using `TEDeactivate`.

`wordBreak` Pointer to the word selection break routine, which determines, firstly, the word that is highlighted when the user double-clicks in the text and, secondly, the position at which text is wrapped at the end of the line.

`clikLoop` Pointer to the click loop routine, which is called repeatedly as long as the mouse button is held down within the text.

`just` The type of text alignment (default, left, centre, or right).

`teLength` The number of bytes in the text to be edited. The maximum allowable length is 32,767 bytes. When you create an edit record, TextEdit initialises this field to 0.

`hText` A handle to the text. When you create an edit record, TextEdit initialises this field to point to a zero-length block in the heap.

`hDispatchRec` Handle to the TextEdit dispatch record. For internal use only.

`clikStuff` TextEdit sets this field to reflect whether the most recent mouse-down event occurred on the leading or trailing edge of a glyph. Used internally by TextEdit to determine a caret position.

`crOnly` Specifies whether or not text wraps at the right edge of the destination rectangle. If the value is positive, text does wrap. If the value is negative, new lines are specified explicitly by Return characters only and text does *not* wrap at the edge of the destination rectangle.

`txFont` For a monostyled edit record, this field specifies the font 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. (If you change the value, you should also change the `lineHeight` and `fontAscent` fields as appropriate.)

Multistyled Edit Record. In a multistyled edit record, if the `txSize` field (see below) is set to -1, this field combines with `txFace` and `filler` to hold a handle to the associated style record.

`txFace` For a monostyled edit record, this field specifies the character attributes of all the text in an edit record. If you change this value, the entire text of this edit record has the new characteristic when it is redrawn. (If you change this value, you should also change the `lineHeight` and `fontAscent` fields as appropriate.)

Multistyled Edit Record. If the `txSize` field (see below) is set to -1, this field combines with `txFont` and `filler` 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 in this edit record has the new characteristic when it is redrawn.

`txSize` In a monostyled edit record, this field is set to the size of the text in points.

Multistyled Edit Record. In a multistyled edit record, this field is set to is -1, indicating that the edit record contains associated character attribute information. The `txFont`, `txFace`, and `filler` fields combine to form a handle to the style record in which this character attribute information is stored.

`inPort` Pointer to the graphics port associated with this edit record.

`highHook` Pointer to the routine which deals with text highlighting.

`caretHook` Pointer to the routine that controls the appearance of the caret.

`numLines` 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 elements internally, so do not change the elements of this array. Because this data structure grows and shrinks, the size of the edit record changes.

The Dispatch Record

The `hDispatchRec` field of the edit record stores a 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 which TextEdit uses internally to, for example, measure and draw text or determine a character's position on a line. These routines, called **hook routines**, determine the way TextEdit behaves.[7]

Monostyled TextEdit

This section describes the use of TextEdit with monostyled text, that is, text with a single typeface, style, and size. Everything in this section also applies to using TextEdit with multistyled text except where otherwise indicated.

Initialising TextEdit

Before using TextEdit, you need to initialise TextEdit using `TEInit` which, amongst other things, sets up the private scrap and allocates a handle to it. You may also need to get information about the installed version of TextEdit using `Gestalt` with the selector `gestaltTextEditVersion`[8].

Creating, and Disposing of, a Monostyled Edit Record

Creating a Monostyled Edit Record

To use TextEdit routines, you must first create an edit record using `TENew`. `TENew` returns a handle to the newly-created monostyled edit record. You typically store the returned handle in a field of a document record, the handle to which is typically stored in the application window's `refCon` field.

The required destination and view rectangles are specified in the `TENew` call. To ensure that the first and last glyphs in each line are legible in a document window, you should inset the destination rectangle at least four pixels from the left and right edges of the graphics port, making an additional allowance for scroll bars as appropriate. You typically make the view rectangle equal to the destination rectangle. (If you do not want the text to be visible, specify a view rectangle off the screen.)

When an edit record is created, TextEdit initialises the edit record's fields based on values in the current graphics port record and on the type of edit record you create.

Disposing of an Edit Record

Memory allocated for an edit record may be released by calling `TEDispose`.

---

[7] You can use a TextEdit customisation routine to replace the address of a default hook routine with the address of your own customised routine.

[8] `Gestalt` is described at Chapter 21 — Miscellany.

Setting the Text of an Edit Record

When you create an edit record, it does not contain any text until the user either enters text through the keyboard or opens an existing document. The following describes how to specify *existing* text to be edited.

There are two ways to specify existing text to be edited, namely, by using `TESetText` or by setting the `hText` field of the edit record directly.

Calling `TESetText`

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`. `TESetText` 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 you pass to `TESetText` specifies the length of the text. `TESetText` resets the `teLength` field of the edit record with this value and uses it to determine the end of the text. It also 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. Finally, `TESetText` calculates the line breaks, eliminating the necessity for your application to perform that task.

`TESetText` does not cause the text to be displayed immediately. You must call `InvalRect` to force the text to be displayed at the next update event for the active window.

Changing the `hText` Field

The second method saves memory if you have a lot of text. In this method, you bring text into an edit record by directly changing the `hText` field in 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 `numLines` values to match the new text.

Responding to Events

Activate Events — Activating / Deactivating an Edit Record

When your application receives an activate event, it must call `TEActivate` for an activate event and `TEDeactivate` for a deactivate event.

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

Typically, you include edit record activation and deactivation in that function in your application which handles window activation and deactivation. That said, it is possible to 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 that, when you use `TEClick` and `TESetSelect` (see below) to set the selection range or insertion point, the selection range is not highlighted and the blinking caret is not displayed until the edit record is activated. (However, if outline highlighting is activated[9], the text of the selection range is framed or a gray, unblinking caret is displayed.)

[9]Outline highlighting may be activated and deactivated using `TEFeatureFlag`.

Update Events — Calling TEUpdate

Your application needs to call TEUpdate every time the Event Manager reports an update event for a text editing window. In addition, you must call TEUpdate after changing any fields of the edit record which affect the appearance of the text or after any editing or scrolling operation which alters the onscreen appearance of the text.

EraseRect and TEUpdate should be called after BeginUpdate and before EndUpdate. (If you do not include the EraseRect call, the black caret may sometimes remain visible (unblinking) when the window is deactivated.)

Mouse-Down Events — Calling TEClick

When your application receives notification of a mouse-down event that it determines TextEdit should handle, it must pass the event on to TEClick. TEClick tells TextEdit that a mouse-down event has occurred. Before calling TEClick, however, your application must perform the following steps:

- Convert the mouse location passed in the event record from global coordinates to the local coordinates required by TEClick.
- Determine if the Shift key was held down at the time of the event.

TEClick repeatedly calls the click loop procedure (see below) as long as the mouse button is held down and retains control until the button is released. The behaviour of TEClick depends on whether the Shift key was down at the time of the mouse-down event and on other user actions as follows:

User's Action	Behaviour of TEClick
Shift key down.	Extend the current selection range.
Shift key not down.	Remove highlighting from current selection range. Position the insertion point as close as possible to the location of the mouse click.
Mouse dragged.	Expand or shorten the selection range a character at a time. Keep control until the user releases the mouse button.
Double-click.	Extend the selection to include the entire word where the cursor is positioned.

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.

Key-Down Events - Accepting Text Input

When your application receives a key-down event which it determines TextEdit should handle, it must call TEKey to accept the keyboard input a byte at a time (or to delete a character when the user backspaces over it). TEKey replaces the current selection range with the character passed to it and moves the insertion point just past the inserted character.

Depending on the requirements of your application, you may need to filter out certain character codes (for example, that for a Tab key press) so that they are not passed to TEKey. You should also check that the TextEdit limit of 32,767 bytes will not be exceeded by the insertion of the character before calling TEKey and you should call your scroll bar adjustment routine immediately after the insertion.

Null Events - Caret Blinking

To force the insertion point caret to blink, your application must call TEIdle whenever it receives a null event. You must also ensure that the sleep parameter in the WaitNextEvent call is set to a value equal to or less than the value stored in the low-memory global CaretTime, which determines the blinking time for the caret[10]. That value can be retrieved by a call to LMGetCaretTime.

[10]The blinking time is set by the user using the General Controls Control Panel.

If there is more than one edit record associated with an active window, you must ensure that you pass `TEIdle` the handle to the currently active edit record. You should also check that the handle to be passed to `TEIdle` does not contain `NULL` before calling the routine.

Cutting, Copying, Pasting, Inserting, and Deleting Text

Cutting, Copying, and Pasting

You can use TextEdit to cut, copy, and paste text within a single edit record, between edit records, or across applications. The relevant routines, and their effect in the case of a monostyled edit record, are as follows:

Routine	Use To	Comments
`TECut`	Cut text.	Copies the text to the TextEdit private scrap.
`TECopy`	Copy text.	Copies the text to the TextEdit private scrap.
`TEPaste`	Paste text.	Pastes from the TextEdit private scrap to the edit record. (Used for monostyled text only.)
`TEToScrap`	Copy TextEdit private scrap to the Scrap Manager's desk scrap.	Copying via the Scrap Manager's desk scrap is required if monostyled text is to be carried across applications.
`TEFromScrap`	Copy the Scrap Manager's desk scrap to TextEdit private scrap.	Copying via the Scrap Manager's desk scrap is required if monostyled text is to be carried across applications.
`TEGetScrapLength`	Determine the length of the monostyled text to be pasted.	Returns the size, in bytes, of the text in the private scrap.

If you are using `TEFromScrap` to support pasting to your application from the desk scrap, you will need to ensure that a paste will not cause the TextEdit limit of 32,767 bytes to be exceeded. One way to do this is to call the Scrap Manager's `GetScrap` routine to get the size of the text to be pasted, add this to the size of the text in the edit record, subtract the size of the selection range, and then compare the result against the maximum allowable length of the edit record.

You will need to call your vertical scroll bar adjustment routine immediately after cut and paste operations.

Inserting and Deleting Text

The following TextEdit routines are used to insert and delete monostyled text:

Routine	Use To	Comments
`TEInsert`	Insert monostyled text into the edit record immediately before the selection range or insertion point.	Does not affect the selection range. Redraws the text if necessary. Use for monostyled text only.
`TEDelete`	Remove the selected range of text from the edit record.	Does not transfer the text to either TextEdit's private scrap or the Scrap Manager's desk scrap. Useful for implementing a **Clear** command. Redraws the remaining text if necessary.

You will need to call your vertical scroll bar adjustment routine immediately after insertions and deletions. In addition, you will need to ensure that an insertion will not cause the TextEdit limit of 32,767 bytes to be exceeded.

Setting the Selection Range or Insertion Point

You can use `TESetSelect` to specify the selection range or the position of the insertion point as determined by the application. For example, you can use `TESetSelect` to position the caret at the start of a data entry field where you want the user to enter a value. `TESetSelect` modifies the `selStart` and `selEnd` fields of the edit record.

To select a range of text, you pass `TESetSelect` the handle to the edit record along with the byte offsets of the starting and ending characters. You can set the selection range (or insertion point) to any

character position corresponding to byte offsets 0 to 32767. To display a caret at the insertion point, specify the same values for the `selStart` and `selEnd` parameters.

To implement a **Select All** menu command, specify 0 for starting offset parameter and use the `teLength` field of the edit record for the ending offset parameter.

Enabling, Disabling, and Customising Automatic Scrolling

Enabling and Disabling

You can use the `TEAutoView` procedure to enable automatic scrolling (which, by default, is disabled). `TEAutoView` may also be used to disable automatic scrolling.

Customising

As previously stated, the default click loop procedure does not adjust the scroll bars as the text is scrolled, a situation which can be overcome by replacing the default click loop procedure with an application-defined click loop procedure which updates the scroll bars as it scrolls the text.

The `clikLoop` field of the edit record contains a pointer to a click loop procedure, which is called continuously as long as the mouse button is held down. Installing your custom procedure involves a call to `TESetClickLoop` to assign the address of the procedure to the edit record's `clikLoop` field.

Scrolling Text

To scroll the text when a mouse-down event occurs in a scroll bar, your application needs to first determine how far to scroll the text. The basic value for vertical scrolling of a monostyled edit record is typically the value in the `lineHeight` field of the edit record, which can be used as the number of pixels to scroll for clicks in the Up and Down scroll arrows. For clicks in the gray areas, this value is typically multiplied by the number of text lines in the view rectangle minus 1. Scrolling by dragging the scroll box involves determining the number of text lines to scroll based on the current position of the top of the destination rectangle and the control value on mouse button release.

To scroll the text, you can use either `TEScroll` or `TEPinScroll`, specifying the number of pixels to scroll. The only difference between these two routines is that `TEPinScroll` stops scrolling when the last line is scrolled into the view rectangle. The destination rectangle is offset by the amount you scroll.

Forcing the Selection Range Into the View

Your application can call `TESelView` to force the selection range to be displayed in the view rectangle. When automatic scrolling is enabled, `TESelView` scrolls the selection range into view, if necessary.

Setting Text Alignment

You can change the alignment of the entire text of an edit record by calling `TESetAlignment` (old name `TESetJust`). The following constants apply:

Constant	Description
`teFlushDefault`	Default alignment according to primary line direction of the script system. (Left for Roman script system.)
`teCenter`	Centre alignment.
`teFlushRight`	Right alignment.
`teFlushLeft`	Left alignment.

You should call the Window manager's `InvalRect` routine after you change the alignment so that the text is redrawn in the new alignment.

Saving and Opening TextEdit Documents

The demonstration program at Chapter 14 — Files demonstrates opening and saving monostyled TextEdit documents.

Multistyled TextEdit

This section addresses additional factors and considerations applying to multistyled TextEdit.

Text With Multiple Styles — Style Runs, Text Segments, Font Runs, and Character Attributes

Text which uses a variety of fonts, styles, sizes, and colours is referred to as **multistyled text**.

TextEdit organises multistyled text into **style runs**, which comprise a sets of contiguous characters which all share the same font, size, style, and colour characteristics. TextEdit tracks style runs in the data structures allocated for a multistyled edit record and uses this information to correctly display multistyled text.

The part of a style run that exists on a single line is called a **text segment**. A larger division than a style run is the **font run**, which comprises those characters which share the same font. The font, style, size, and colour aspects of text are collectively referred to as **character attributes**.

Additional TextEdit Data Structures for Multistyled Text

The edit record and the dispatch record are the only data structures associated with monostyled text. However, when you allocate a multistyled edit record, a number of additional subsidiary data structures are created to support the text styling capabilities. The additional data structures associated with a multistyled edit record are shown at Fig 6.

The Style Record

The first of these additional data structures is the **style record**, which stores the character attribute information for the text. (Recall that, when a multistyled edit record is created, the bytes at the `txFace`, `txFace`, and `filler` fields of the edit record contain a handle to the style record.) The remaining additional data structures are, in fact, elements of the style record. Those elements are as follows:

- A handle to a **style table**, which has one entry for each distinct set of character attributes in the edit record. Each element in the style table is a **style table element record**.

- A handle to the **line-height table**, which provides vertical spacing and line ascent information for the text to be edited. The line-height table comprises one **line-height element record** for each line of an edit record. A line number is a direct index into the array of line-height element records.

- A **style run table**, which is an array of **style run records**, each of which provides, for each style run, the offset of the starting character to which the character attributes stored in the style table apply and an index into the style table.

- A handle to the **null style record**, which contains a handle to the **style scrap record**. The style scrap record, which is part of the style 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 and copied. Part of the style scrap record is the **scrap style table**, which comprises one **scrap style element record** for each style run in the style scrap record. Scrap style element records hold character attribute information similar to that contained in the style table element record.

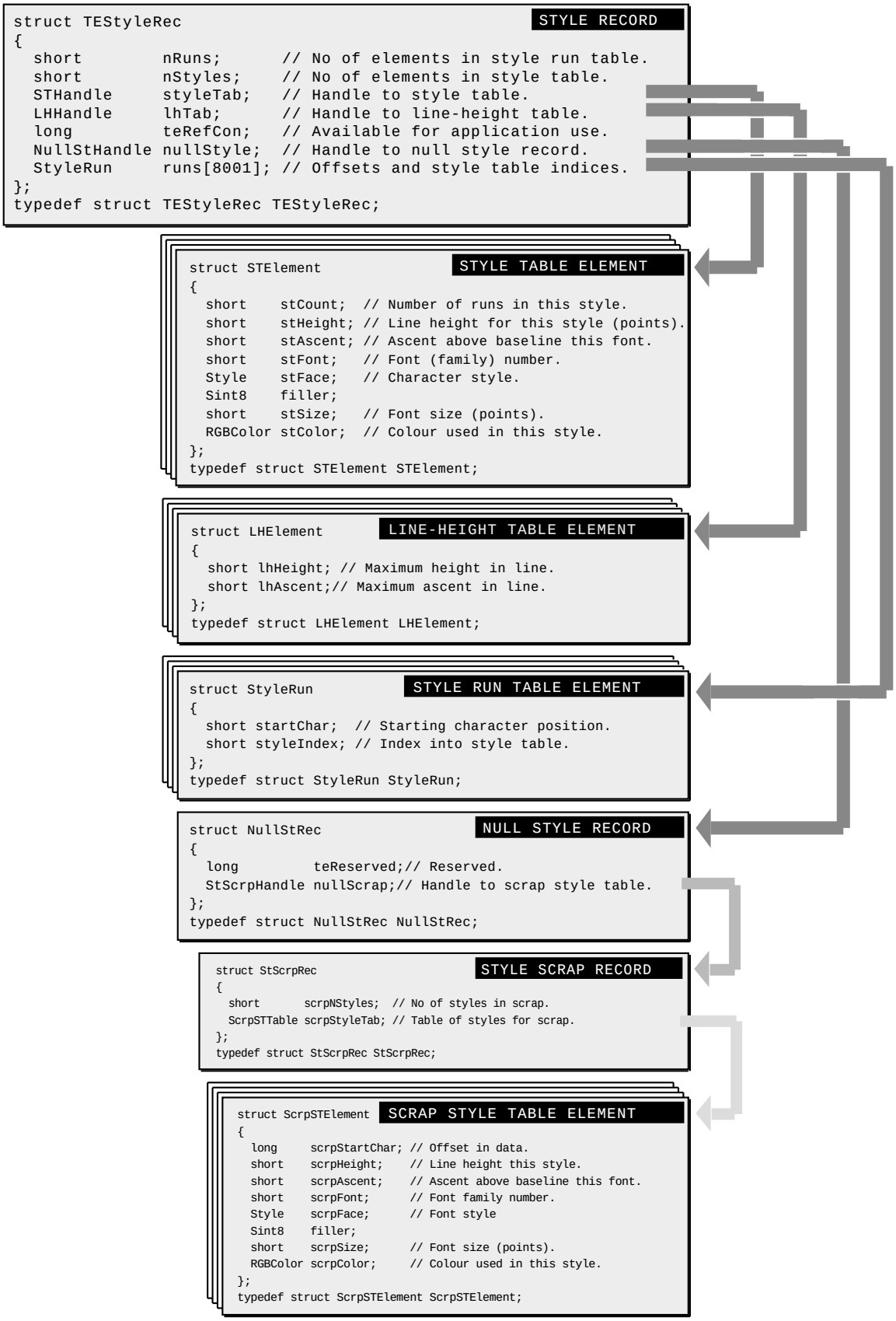

FIG 6 - THE STYLE RECORD AND SUBSIDIARY DATA STRUCTURES

Creating a Multistyled Edit Record

The multistyled edit record is created by calling `TEStyleNew` (old name `TEStylNew`).

Setting the Text

The alternative method of setting the text (that is, directly setting the `hText` field of the edit record) is somewhat more cumbersome for a multistyled text because `TECalText` does not update the style run table 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 the `selEnd` field to 32,767, and 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.

`TEKey` and Multistyled Text

When the user backspaces over characters in a multistyled edit record, `TEKey` deletes the characters (as in a monostyled edit record) but also saves the character attributes associated with the last character deleted in order to apply it to any new characters the user enters. 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.

Cutting, Copying, Pasting, Inserting, and Deleting Text

The following shows the effects of `TECut` and `TECopy` when multistyled text is involved. It also describes `TEStylePaste` (old name `TEStylPaste`), which is used for pasting multistyled text, and the additional routine (`TENumStyles`), which is involved in cutting and copying multistyled text:

Routine	Use To	Comments
`TECut` `TECopy`	Cut text. Copies text.	For multistyled text: • Copies both the text and its character attribute information to the Scrap Manager's desk scrap under scrap types `'TEXT'` and `'styl'`. • Copies the text to the TextEdit private scrap and the attributes stored in the style table to the TextEdit style scrap.
`TEStylePaste`	Paste multistyled text.	Pastes both the text and its attributes from the Scrap Manager's desk scrap to the edit record. (Use the Scrap Manager routine `GetScrap` to check the size of the text (`'TEXT'` data) to be pasted, passing `NULL` for the `hDest` parameter to avoid copying the data.)
`TENumStyles`	Determine the memory required for the style scrap before cutting or copying multistyled text.	`TENumStyles` returns the number of attribute changes contained in the range of text. Multiply this by `sizeof(ScrpSTElement)` and add two to get the number of bytes required.

The following describes `TEStyleInsert` (old name `TEStylInsert`), which is used to insert multistyled text, and the additional effects of `TEDelete` when used to delete multistyled text:

Routine	Use To	Comments
`TEStyleInsert`	Insert multistyled text into the edit record immediately before the selection range or insertion point.	Does not affect the selection range. Redraws the text if necessary. Applies the specified character attributes to the text. (You should create your own style scrap record, specifying the style attributes to be inserted and applied to the text. These attributes are copied directly into the style record's style table.)

`TEDelete`	Remove the selected range of text from the edit record.	Does not transfer the text to either TextEdit's private scrap or the Scrap Manager's desk scrap. Redraws the remaining text if necessary. For multistyled text, the character attributes are saved in the null scrap to be applied to characters after the text has been deleted. When the user clicks in some other area of the text, the attributes are removed from the null scrap. `TEDelete` can be used to implement a **Clear** command.

Scrolling Text

The number of pixels to be scrolled vertically for each line of text to be scrolled (determined from the `lineHeight` field in a monostyled edit record) is determined from the `lnHeight` field of the line height table in multistyled edit records.

Setting and Checking Text Attributes

Your application may need to check the current attributes of a range of text to determine which ones are consistent across a range of text. Your application may also need to manipulate the font, style, size, and colour of a range of text, the text selection consisting of the entire text of the edit record, a segment of text, a single character, or even an insertion point. The following routines are relevant in this regard:

Routine	Use To	Comments
`TESetStyle`	Change the font, size, style and / or colour of the text in the selection range.	Typically used to implement menu commands relating to modifying text attributes. If called for an insertion point, TextEdit stores the specified attribute information in the null scrap's style scrap record.
`TEContinuousStyle`	Examine the current selection range and determine whether a specified style attribute is continuous across the range.	Can be used as an aid in toggling styles or to determine which of the items in a **Style** menu should have a checkmark. The `mode` parameter specifies which attributes of the selected text are to be examined.

Checking Attributes in a Selection Range

The following example application-defined function shows how to determine the font, size, style and colour of the current selection range.

```
void  doGetCurrentSelection(TextStyle *textStyleRec,TEHandle editRecHdl)
{
  SInt16  mode;
  Boolean continuous;

  // doFont, doFace, doSize, and doColor are constants which specify font family number,
  // character style, type size, and colour

  mode = doFont + doFace + doSize + doColor;
  continuous = TEContinuousStyle(&mode,textStyleRec,editRecHdl);

  // When TEContinuousStyle returns, each bit in mode that was set on entry will have
  // been cleared if that style element was not continuous.  For those attributes which
  // were continuous, the text style (TextStyle) record fields will contain the actual
  // values.

  if((mode & doFont) != 0)
    // Font for selection = tsFont field of the TextStyle record.
  else
    // More than one font in selection.

  if((mode & doFace) != 0)
    // tsFace field of the TextStyle record contains the styles (or plain) common
    // to the selection.
  else
    // No text face is common to the entire selection.

  if((mode & doSize) != 0)
    // Size for selection = tsSize field of the TextStyle record.
```

```
  else
    // More than one size in selection.

  if((mode & doColor) != 0)
    // Color for selection = tsColor field of the TextStyle record.
  else
    // More than one colour in selection.
}
```

Handling a Font Menu

The following example application-defined function shows how to handle a **Font** menu item selection.

```
void  doFontMenu(WindowPtr windowPtr,TEHandle editRecHdl,SInt16 menuItem)
{
  TextStyle styleRec;
  Str255    fontName;
  SInt16    fontID;

  GetMenuItemText(GetMenuHandle(mFont),menuItem,fontName);  // Get text of menu item.
  GetFNum(fontName,&fontID);                    // Get font number matching font name.
  styleRec.tsFont = fontID;           // Assign to tsFont field of text style record.
  TESetStyle(doFont,&styleRec,true,editRecHdl); // Apply style to selected text ...
                                                // and redraw text immediately.
  doAdjustScrollBars(windowPtr,false);          // Adjust scroll bars.
}
```

Handling a Size Menu

The following example application-defined function shows how to handle a **Size** menu item selection.

```
void  doSizeMenu(WindowPtr windowPtr,TEHandle editRecHdl,SInt16 menuItem)
{
  SInt16    sizeChosen;
  TextStyle styleRec;

  doGetSize(GetMenuHandle(mSize),menuItem,sizeChosen);  // Get size from menu item.
  styleRec.tsSize = sizeChosen;                     // Assign to tsSize field of text
                                                    // style record.
  TESetStyle(doSize,&styleRec,true,editRecHdl);     // Apply size to selected text ...
                                                    // and redraw text immediately.
  doAdjustScrollBars(windowPtr,false);              // Adjust scroll bars.
}
```

Handling a Style Menu

The following example application-defined function shows how to handle a **Style** menu item selection.

```
void  doHandleStyleMenu(WindowPtr windowPtr,TEHandle editRecHdl,SInt16 menuItem)
{
  TextStyle styleRec;

  switch menuItem
  {
    case iPlain:
      styleRec.tsFace = (Style) normal;
      break;

    case iBold:
      styleRec.tsFace = (Style) bold;
      break;

    case iItalic:
      styleRec.tsFace = (Style) italic;
      break;

    case iUnderline:
      styleRec.tsFace = (Style) underline;
      break;

    case iOutline:
      styleRec.tsFace = (Style) outline;
      break;
```

```
      case iShadow:
        styleRec.tsFace = (Style) shadow;
        break;
    }

    if(menuItem != 1)                                          // If Plain not selected
      TESetStyle(doFace + doToggle,&styleRec,true,editRecHdl)  // ... include doToggle.
    else                                                       // If Plain selected
      TESetStyle(doFace,&styleRec,true,editRecHdl);            // ... delete doToggle.

    doAdjustScrollBars(windowPtr,false);
  }
```

Note that, if you call `TESetStyle` with the value of `false` for the `redraw` parameter, TextEdit does not redraw the text or recalculate line breaks, line heights or font ascents until the next update occurs. This will cause problems if you call a routine that uses any of this information before the update occurs.

The following example application-defined function checks the character attributes of the current selection range and, for each style that is continuous across the range, marks the item in the **Style** menu.

```
void  doAdjustStyleMenu(TEHandle editRecHdl)
{
  MenuHandle  styleMenuHdl;
  TextStyle   styleRec;
  SInt16      mode;

  mode = doFace;
  styleMenuHdl = GetMenuHandle(mStyle);

  // If TEContinuousStyle returns true, there is at least one style that is continuous
  // over the selection.  Note that it might be plain, which is the absence of styles.

  if(TEContinuousStyle(&mode,&styleRec,editRecHdl))          // There is a continuous
  {                                                          // style so mark all menu
    CheckItem(styleMenuHdl,iPlain,styleRec.tsFace == normal); // items appropriately.
    CheckItem(styleMenuHdl,iBold,styleRec.tsFace & bold);
    CheckItem(styleMenuHdl,iItalic,styleRec.tsFace & italic);
    // Set other items as appropriate.
  }
  else                                                       // There is no continuous
  {                                                          // style so unmark all
    CheckItem(styleMenuHdl,iPlain,false);                    // menu items.
    CheckItem(styleMenuHdl,iBold,false);
    CheckItem(styleMenuHdl,iItalic,false);
    // Set other items as appropriate.
  }
}
```

Handling the Undo Command

If you are implementing an **Undo** command for multistyled text, you need to save the character attribute information along with the text. There are a number of ways to 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 `TEGetStyleScrapHandle`, 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 `TEUseStyleScrap`. 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 applied.

Saving and Opening Multistyled TextEdit Documents

Saving a Multistyled TextEdit Document

To save the contents of a document created with a multistyled edit record, you need to save all the associated character attribute information[11] in addition to the text. Because the text attribute information in the style scrap is easier to export that the style record itself (because 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:

- Create a text file, select all the text of the edit record, and save it to the data fork.
- 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.
- Save the character attribute information in the resource fork of the file.

The following example application-defined function uses this method.

```
void doSaveAsTextEdit(TEHandle editRecHdl)
{
  StandardFileFileReply fileReply;
  StScrpHandle          styleScrapHdl;
  SInt32                dataLength;
  SInt16                dataRefNum;
  SInt16                rsrcRefNum;
  SInt16                savedStart;
  SInt16                savedEnd;
  OSErr                 osError;
  Handle                editText;

  StandardPutFile("\pSave as:","\pUntitled",&fileReply);
  if(fileReply.sfGood);
  {
    savedStart = (*editRecHdl)->selStart;             // Save current selection ...
    savedEnd = (*editRecHdl)->selEnd;                 // starting and ending offsets.

    (*editRecHdl)->selStart = 0;                      // Select all text. (Do not ..
    (*editRecHdl)->selEnd = (*editRecHdl)->teLength;  // use TESetSelect.)
    styleScrapHdl == GetStylScrap(editRecHdl);        // Get list of all attributes.
    (*editRecHdl)->selStart = savedStart;             // Reset original selection.
    (*editRecHdl)->selEnd = savedEnd;

    if(!(fileReply.sfReplacing))                      // Create file & resource ...
    {                                                 // fork if not already created.
      osError = FSpCreate(&fileReply.sfFile,'K JB','TEXT',fileReply.sfScript);
      FSpCreateResFile(fileReply,sfFile,'K JB','TEXT',fileReply.sfScript);
      osError = ResError();
    };

    osError = FSpOpenDF(&fileReply.sfFile,fsCurPerm,&dataRefNum); // Open data fork ..
    rsrcRefNum = FSpOpenResFile(&fileReply.sfFile,fsCurPerm);   // and resource fork.
    osError = ResError();

    dataLength = (*editRecHdl)->teLength;               // Write text.
    editText = (*editRecHdl)->hText;
    osError = FSWrite(dataRefNum,&dataLength,*editText);

    AddResource((Handle) styleScrapHdl,'styl',0,"\p");  // Write attributes.
    WriteResource((Handle) styleScrapHdl);
    ReleaseResource((Handle) styleScrapHdl);

    osError = FSClose(dataRefNum);                      // Close data fork ...
    CloseResFile(rsrcRefNum);                           // and resource fork.
    osError = ResError();
  }
}
```

[11] For the font, remember to save the font name, not the font number.

Notice that this function avoids using `TESetSelect` to select all of the edit record's text. `TESetSelect` sets *and* highlights the selection range that you specify. You do not want the text to be highlighted because this could mislead the user into presuming that some other action is required.[12]

Opening a Multistyled TextEdit Document

The following example application-defined function shows how to open a multistyled TextEdit document:

```
void doOpenAsTextEdit(TEHandle editRecHdl)
{
  StandardFileFileReply fileReply;
  SFFileTypes           fileTypes;
  SInt16                dataRefNum;
  SInt16                rsrcRefNum;
  Handle                textBuffer;
  SInt32                textLength;
  StScrpHandle          styleScrapHdl;
  OSErr                 osError;
  UInt8                 savedState;

  fileTypes[0] = 'TEXT';

  StandardGetFile(NULL,1,fileTypes,&fileReply);
  if(fileReply.sfGood);
  {
    osError = FSpOpenDF(&fileReply.sfFile,fsCurPerm,&dataRefNum);
    osError = SetFPos(dataRefNum,fsFromStart,0);
    osError = GetEOF(dataRefNum,&textLength);

    if(textlength > 32767)
      textLength = 32767;

    textBuffer = NewHandle((Size) textLength);            // Allocate buffer for text.
    osError = FSRead(dataRefNum,&textLength,*textBuffer); // Read text to buffer.
    LockHHi(textBuffer);
    TESetText(*buffer,textlength,editRecHdl);             // Put text in edit record.
    HUnlock(textBuffer)
    DisposeHandle(textBuffer);                            // Get rid of buffer.

    osError = FSClose(dataRefNum);                        // Close data fork.

    rsrcRefNum := FSpOpenResFile(&fileReply.sfFile,fsCurPerm);  // Open resource fork.
    osError = ResError();

    styleScrapHdl = GetResource('styl',0);                // Get style scrap.
    osError = ResError();
    if(styleScrapHdl != NULL)
    {                                                     // Apply attributes ...
      savedState = HGetState((Handle) styleScrapHdl);     // to edit record
      TEUseStyleScrap(0,textLength,styleScrapHdl,true,editRecHdl);
      HSetState((Handle) styleScrapHdl,savedState);
    }

    CloseResFile(rsrcRefNum);                             // Close resource fork.
    osError = ResError();
  }
}
```

[12]However, if you want to use `TESetSelect`, you can circumvent highlighting of the selection range if you first render the edit record inactive. Also, if you have the outline highlighting feature turned on, turn it off.

Formatting and Displaying Dates, Times, and Numbers

Preamble — The Text Utilities and International Resources

The Text Utilities

The **Text Utilities** are a collection of text-handling routines provided by the system software which allow you to specify strings for various purposes, sort strings, convert case or strip diacritical marks from text for sorting purposes, search and replace text, find word boundaries and line breaks when laying out lines of text, and format numbers, currency, dates, and times. The following is concerned only with the latter, that is, formatting numbers, currency, dates, and times.

International Resources

Many Text Utilities routines utilise the **international resources**, which define how different text elements are represented depending on the script system in use. The international resources relevant to formatting numbers, currency, dates, and times are as follows:

- **Numeric Format Resource.** The numeric format (`'itl0'`) resource contains short date and time formats, and formats for currency, numbers, and the preferred unit of measurement. It provides separators for decimals, thousands, and lists. It also contains the region code for this particular resource. Three of the several variations in short date and time formats are as follows:

System Software	Morning	Afternoon	Short Date
United States	`1:02 AM`	`1:02 PM`	`2/1/90`
Sweden	`01:02`	`13:02`	`90-01-01`
Germany	`1:02 Uhr`	`13:02 Uhr`	`2.1.1990`

- **Long Date Format Resource.** The long date format (`'itl1'`) resource specifies the long and abbreviated date formats for a particular region, including the names of days and months and the exact order of presentation of the elements. It also contains a region code for this particular resource. Three of the several variations of the long and abbreviated date formats are as follows:

System Software	Abbreviated Date	Long Date
United States	`Tue, Jan 2, 1990`	`Tuesday, January 2 1990`
French	`Mar 2 Jan 1990`	`Mardi 2 Janvier 1990`
Australian	`Tue, 2 Jan 1990`	`Tuesday, 2 January 1990`

- **Tokens Resource.** The tokens (`'itl4'`) resource contains, amongst other things, a table for formatting numbers. This table, which is called the **number parts table**, contains standard representations for the components of numbers and numeric strings. As will be seen, certain Text Utilities number formatting routines use the number parts table to create number strings in localised formats.

Date and Time

The Text Utilities routines which work with dates and times use information in the international resources to create different representations of date and time values. The Operating System provides routines that return the current date and time in numeric format. Text Utilities routines can then be used to convert these values into strings which can, in turn, be presented in the different international formats.

Date and Time Value Representations

The Operating System provides the following differing representations of date and time values:

Representation	Description
Standard date-time value.	A 32-bit integer representing the number of seconds between midnight, 1 January 1904 and the current time.

Long date-time value.	A 64-bit signed representation of data type `LongDateTime`. Allows for coverage of a longer time span than the standard date-time value, specifically, about 30,000 years.
Date-time record.	Data type `DateTimeRec`. Includes integer fields for year, month, day, hour, minute, second, and day of week.
Long date-time record	Data type `LongDateRec`. Similar to the date-time record, except that it adds several additional fields, including integer values for the era, day of the year, and week of the year. Allows for a longer time span than the date-time record.

The date-time (`DateTimeRec`) and the long date-time (`LongDateRec`) records are as follows:

```
struct DateTimeRec
}
  short  year;
  short  month;
  short  day;
  short  hour;
  short  minute;
  short  second;
  short  dayOfWeek;
};

typedef struct DateTimeRec DateTimeRec;
```

```
union LongDateRec
{
  struct
  {
    short  era;
    short  year;
    short  month;
    short  day;
    short  hour;
    short  minute;
    short  second;
    short  dayOfWeek;
    short  dayOfYear;
    short  weekOfYear;
    short  pm;
    short  res1;
    short  res2;
    short  res3;
  } ld;
  short    list[14];
  struct
  {
    short        eraAlt;
    DateTimeRec  oldDate;
  } od;
};

typedef union LongDateRec LongDateRec;
```

Obtaining Date-Time Values and Records

The Operating System Utilities provide the following two routines for obtaining date-time values and records.

Routine	Description
`GetDateTime`	Returns a standard date-time value.
`GetTime`	Returns a date-time record.

Converting Between Values and Records

The Operating System provides the following four procedures for converting between the different date and time data types:

Routine	Converts	To
`DateToSeconds`	Date-time record.	Standard date-time value.
`SecondsToDate`	Standard date-time value.	Date-time record.
`LongDateToSeconds`	Long date record.	Long date-time value.
`LongSecondsToDate`	Long date-time value.	Long date record.

Converting Date-Time Values Into Strings

The Text Utilities provide the following routines for converting from one of the numeric date-time representations to a formatted string.

Routine	Description
DateString	Converts standard date-time value to a date string formatted according to the specified international resource.
LongDateString	Converts long date-time value to a date string formatted according to the specified international resource.
TimeString	Converts standard date-time value to a time string formatted according to the specified international resource.
LongTimeString	Converts long date-time values to a time string formatted according to the specified international resource.

Output Format — Date. When you use DateString and LongDateString, you can specify, in the longFlag parameter, an output format for the resulting date string. This format can be one of the following three values of the DateForm enumerated data type:

Value	Date String Produced (Example)	Formatting Information Obtained From
shortDate	1/31/92	Numeric format resource ('itl0').
abbrevDate	Fri, Jan 31, 1992	Long date format resource ('itl1').
longDate	Friday, January 31, 1992	Long date format resource ('itl1').

Output Format — Time. When you use TimeString and LongTimeString, you can request an output format for the resulting time string by specifying either true or false in the wantSeconds parameter. true will cause seconds to be included in the string.

DateString, LongDateString, TimeString and LongTimeString use the date and time formatting information in the format resource that you specify in the resource handle (intlHandle) parameter. If you specify NULL for the value of the resource handle parameter, the appropriate format resource for the *current* script system is used.

Converting Date-Time Strings Into Internal Numeric Representation

The Text Utilities include routines which can parse date and time strings as entered by users and fill in the fields of a record with the components of the date and time, including the month, day, year, hours, minutes, and seconds, extracted from the string.

Suppose your application needs to, say, convert a date and time string typed in by the user (for example, "March 27, 1992, 08:14 p.m.") into numeric representation. The following Text Utilities routines may be used to convert the string entered by the user into a long date-time record:

Routine	Description
StringToDate	Parses an input string for a date and creates an internal numeric representation of that date. Returns a status value indicating the confidence level for the success of the conversion. Expects a date specification, in one of the formats defined by the current script system, at the beginning of the string. Recognizes date strings in many formats, for example: "September 1,1987", "1 Sept 87", "1/9/87", and "1 1987 Sept".
StringToTime	Parses an input string for a time and creates an internal numeric representation of that time. Returns a status value indicating the confidence level for the success of the conversion. Expects a time specification, in a format defined by the current script system, at the beginning of the string.

You usually call StringToDate and StringToTime sequentially to parse the date and time values from an input string and fill in these fields. Note that StringToDate assigns to its lengthUsed parameter the number of bytes that it uses to parse the date. Use this value to compute the starting location of the text that you can pass to StringToTime.

The "confidence level" value returned by both StringToDate and StringToTime is of type StringToDateStatus, a set of bit values which have been OR'd together. The higher the resultant number, the lower the confidence level. Three of the twelve StringToDateStatus values, and their meanings, are as follows:

Value	Meaning
`fataldateTime`	A fatal error occurred during the parse.
`dateTimeNotFound`	A valid date or time value could not be found in the string.
`sepNotIntlSep`	A valid date or time value was found; however, one or more of the separator characters in the string was not an expected separator character for the script system in use.

Date Cache Record. Both `StringToDate` and `StringToTime` take a **date cache record** as one of their parameters. A date cache record (a data structure of type `DateCacheRec`) stores date conversion data used by the date and time conversion routines. You must declare a data cache record in your application and initialise it by calling `InitDateCache` once, typically in your main program initialisation code.

Numbers

When you present numbers to the user, or when the user enters numbers for your application to use, you need to convert between the internal numeric representation of the number and the output (or input) format of the number. The Text Utilities provide several routines for performing these conversions.

Integers

The simplest number conversion tasks involve integer values. The following Text Utilities routines may be used to convert an integer value to a numeric string and vice versa:

Routine	Description
`NumToString`	Converts a long integer value into a string representation.
`StringToNum`	Converts a string representation of a number into a long integer value.

The range of values accommodated by these routines is -2,147,483,647 to 2,147,483,648. No comma insertion or other formatting is performed.

Number Format Specification Strings

If you are working with floating point numbers, or if you want to accommodate the possible differences in number output formats for different countries and regions of the world, you need to work with **number format specification strings**. Number format specification strings define the appearance of numeric strings in your application.

Parts. Each number format specification string contains up to three parts:

- The positive number format.
- The negative number format.
- The zero number format.

Each of these formats is applied to a numeric value of the corresponding type. When the specification string contains only one part, that part is used for all values. When in contains two parts, the first part is used for positive and zero values and the second part is used for negative values.

Elements. A number format specification string can contain the following elements:

- Number parts separators (, and .) for specifying the decimal separator and the thousands separator.
- Literals to be included in the output. (Literals can be strings or brackets, braces and parentheses, and must be enclosed in quotation marks.)
- Digit place holders. (Digit place holders that you want displayed must be indicated by digit symbols. Zero digits (`0`) add leading zeroes whenever an input digit is not present. Skipping digits (`#`) only produce output characters when an input digit is present. Padding digits (`^`) are

like zero digits except that a padding character such as a non-breaking space is used instead of leading zeros to pad the output string.)

- Quoting mechanisms for handling literals correctly.
- Symbol and sign characters.

Examples. The following shows several different number format specification strings and the output produced by each:

Number Format Specification String	Numeric Value	Output Format
`###,###.##;-###,###.##;0`	123456.78	`123,456.78`
`###,###.0##,###`	1234	`1,234.0`
`###,###.0##,###`	3.141592	`3.141,592`
`###;(000);^^^`	-1	`(001)`
`###.###`	1.234999	`1.235`
`###'CR';###'DB';''zero''`	1	`1CR`
`###'CR';###'DB';''zero''`	0	`'zero'`
`##%`	0.1	`10%`

The number formatting routines always fill in integer digits from the right and decimal places on the left. The following examples, in which a literal is included in the middle of the format strings, demonstrate this behaviour:

Number Format Specification String	Numeric Value	Output Format
`###'my'###`	1	`1`
`###'my'###`	123	`123`
`###'my'###`	1234	`1my1234`
`0.###'my'###`	0.1	`0.1`
`0.###'my'###`	0.123	`1.123`
`0.###'my'###`	0.1234	`0.123my4`

Overflow and Rounding. If the input string contains more digits than are specified in the number format specification string, an error (`formatOverflow`) will be generated. If the input string contains too many decimal places, the decimal portion is automatically rounded. For example, given the format ###.###, a value of 1234.56789 results in an error condition, and a value of 1.234999 results in the rounded-off value 1.235.

Converting Number Format Specification Strings to Internal Numeric Representations. With the required number format specification string defined, you must then convert the string into an internal numeric representation. The internal representation of format strings is stored in a `NumFormatString` record. You use the following routines to convert a number format specification string to a `NumFormatString` record and vice versa.

Routines	Description
`StringToFormatRec`	Converts a number format specification string into a `NumFormatString` record.
`FormatRecToString`	Convert a `NumFormatString` record back to a number format specification string.

Number Parts Table. The internal numeric representation allows you to map the number into different output formats. One of the parameters taken by `StringToFormatRec` is a number parts table. The number parts table specifies which characters are used for certain purposes, such as separating parts of a number[13], in the format specification string.[14] As previously stated, the number parts table is contained in the `'itl4'` resource. A handle to the `'itl4'` resource may be obtained by a call to `GetIntlResourceTable` (old name `IUGetItlTable`), specifying `iuNumberPartsTable` in the `tableCode` parameter.

---

[13] For example, a thousands separator is a comma in Australia and a decimal point in France.

[14] The `FormatRecToString` function also contains a number parts table parameter. By using a different table than was used in the call to `StringToFormatRec`, you can produce a number format specification string that specifies how numbers are formatted for a different region of the world. You use `FormatRecToString` when you want to display the number format specification string to the user for perusal or modification.

Converting Between Floating Point Numbers and Numeric Strings

Once you have a `NumFormatString` record which defines the format of numbers for a certain region of the world, you can convert floating point numbers into numeric strings and numeric strings into floating point numbers using the following routines:

Routines	Description
`StringToExtended`	Using a `NumFormatString` record and a number parts table, converts a numeric string to an 80-bit floating point value.
`ExtendedToString`	Using a `NumFormatString` record and a number parts table, converts an 80-bit floating point number to a numeric string.

`StringToFormatRec`, `FormatRecToString`, `StringToExtended`, and `ExtendedToString` return a result of type `FormatStatus`, which is an integer value. The low byte is of type `FormatResultType`. Typical examples of the returned format status are as follows:

Value	Meaning
`fFormatOK`	The format of the input value is appropriate and the conversion was successful.
`fBestGuess`	The format of the input value is questionable. The result of the conversion may or may not be correct.
`fBadPartsTable`	The parts table is not valid.

Main TextEdit Constants, Data Types and Routines

Constants

Alignment

```
teFlushDefault      = 0
teCenter            = 1
teFlushRight        = -1
teFlushLeft         = -2
```

Values for `TESetStyle`

```
doFont              = 1
doFace              = 2
doSize              = 4
doColor             = 8
doAll               = 15
addSize             = 16
```

Feature or Bit Definitions for `TEFeatureFlag feature` Parameter

```
teFAutoScroll       = 0
teFAutoScr          = 0
teFTextBuffering    = 1
teFOutlineHilite    = 2
teFInlineInput      = 3
teFUseTextServices  = 4
```

Data Types

```
typedef char Chars[32001];
typedef char *CharsPtr;
typedef CharsPtr *CharsHandle;
```

Edit Record

```
struct TERec
{
  Rect          destRect;    // Destination rectangle.
  Rect          viewRect;    // View rectangle.
  Rect          selRect;     // Selection rectangle.
```

```
  short         lineHeight;  // Used for vertical spacing of lines.
  short         fontAscent;  // Used for caret size and highlight rectangle calculation.
  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 or deactivated.
  WordBreakUPP  wordBreak;   // Word break hook procedure.
  ClikLoopUPP   clikLoop;    // Click loop hook procedure.
  long          clickTime;   // (Used internally.)
  short         clickLoc;    // (Used internally.)
  long          caretTime;   // (Used internally.)
  short         caretState;  // (Used internally.)
  short         just;        // Text alignment.
  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.   // If multistyled edit record (txSize = -1),
  Style         txFace;      // Chara style. // these bytes are used as a handle to a
  SInt8         filler;      //              // style record (TEStyleHandle).
  short         txMode;      // Pen mode.
  short         txSize;      // Value indicates font size or, if -1, multistyled edit record.
  GrafPtr       inPort;      // Pointer to grafPort for this edit record.
  HighHookUPP   highHook;    // Used for text highlighting, caret appearance.
  CaretHookUPP  caretHook;   // Used from assembly language.
  short         nLines;      // Number of lines.
  short         lineStarts[16001];  // Positions of line starts.
};

typedef struct TERec TERec,*TEPtr,**TEHandle;
```

Style Record

```
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;
```

Style Table Element

```
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.
  Sint8        filler;
  short        stSize;         // Size in points.
 RGBColor      stColor;        // absolute (RGB) color.
};

typedef struct STElement STElement;
typedef STElement TEStyleTable[1777], *STPtr, **STHandle;
```

Line Height Table Element

```
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.
```

Style Run Table Element

```
struct StyleRun
{
  short        startChar;      // Starting character position.
  short        styleIndex;     // index in style table.
};

typedef struct StyleRun StyleRun;
```

Null Style Record

```
struct NullStRec
{
  long         teReserved;     // Reserved.
  StScrpHandle nullScrap;      // Handle to scrap style table.
};
typedef struct NullStRec NullStRec;
typedef NullStRec *NullStPtr, **NullStHandle;
```

Style Scrap Record

```
struct StScrpRec
{
  short        scrpNStyles;    // Number of styles in scrap.
  ScrpSTTable  scrpStyleTab;   // Table of styles for scrap.
};
typedef struct StScrpRec StScrpRec;
typedef StScrpRec *StScrpPtr, **StScrpHandle;
```

Scrap Style Table Element

```
struct ScrpSTElement
{
  long         scrpStartChar;  // starting character position.
  short        scrpHeight;     // starting character position.
  short        scrpAscent;     // Ascent above baseline this font.
  short        scrpFont;       // Font (family) number.
  Style        scrpFace;       // Character style.
  Sint8        filler;
  short        scrpSize;       // Font size (points).
  RGBColor     scrpColor;      // Colour used this style.
};

typedef struct ScrpSTElement ScrpSTElement;
typedef ScrpSTElement ScrpSTTable[1601];  //  ARRAY [0..1600] OF ScrpSTElement.
```

Text Style Record

```
struct TextStyle
{
  short        tsFont;         // Font (family) number.
  Style        tsFace;         // Character Style.
  char         filler;
  short        tsSize;         // Size in point.
  RGBColor     tsColor;        // Absolute (RGB) color.
};

typedef struct TextStyle TextStyle;
typedef TextStyle *TextStylePtr, **TextStyleHandle;
```

Routines

Note: Some TextEdit routines can be accessed using more than one spelling of the routine's name, depending on the interface files supported by your development environment. The following reflects the newest spellings, as specified in version 2.1 of the Universal Headers.

Initialising TextEdit, Creating Edit Record, Disposing of Edit Record

```
void           TEInit(void);
TEHandle       TENew(const Rect *destRect,const Rect *viewRect);
TEHandle       TEStyleNew(const Rect *destRect,const Rect *viewRect);
void           TEDispose(TEHandle hTE);
```

Activating and Deactivating an Edit Record

```
void           TEActivate(TEHandle hTE);
void           TEDeactivate(TEHandle hTE);
```

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

```
void           TEKey(short key,TEHandle hTE);
void           TESetText(const void *text,long length,TEHandle hTE);
CharsHandle    TEGetText(TEHandle hTE);
void           TESetStyleHandle(TEStyleHandle theHandle,TEHandle hTE);
TEStyleHandle  TEGetStyleHandle(TEHandle hTE);
```

Setting the Caret and Selection Range

```
void           TEIdle(TEHandle hTE);
void           TEClick(Point pt,Boolean fExtend,TEHandle h);
void           TESetSelect(long selStart,long selEnd,TEHandle hTE);
```

Displaying and Scrolling Text

```
void           TESetAlignment(short just,TEHandle hTE);
void           TEUpdate(const Rect *rUpdate,TEHandle hTE);
void           TETextBox(const void *text,long length,const Rect *box,short just);
void           TECalText(TEHandle hTE);
long           TEGetHeight(long endLine,long startLine,TEHandle hTE);
void           TEScroll(short dh,short dv,TEHandle hTE);
void           TEPinScroll(short dh,short dv,TEHandle hTE);
void           TEAutoView(Boolean fAuto,TEHandle hTE);
void           TESelView(TEHandle hTE);
```

Modifying the Text of an Edit Record

```
void           TEDelete(TEHandle hTE);
void           TEInsert(const void *text,long length,TEHandle hTE);
void           TEStyleInsert(const void *text,long length,StScrpHandle hST,
void           TECut(TEHandle hTE);
void           TECopy(TEHandle hTE);
void           TEPaste(TEHandle hTE);
void           TEStylePaste(TEHandle hTE);
OSErr          TEFromScrap(void);
OSErr          TEToScrap(void);
```

Managing the TextEdit Private Scrap

```
#define        TEScrapHandle(); (* (Handle*) 0xAB4);
#define        TEGetScrapLength(); ((long) * (unsigned short *) 0x0AB0);
void           TESetScrapLength(long length);;
```

Checking, Setting, and Replacing Styles

```
void           TESetStyle(short mode,const TextStyle *newStyle,Boolean redraw,TEHandle hTE);
void           TEReplaceStyle(short mode,const TextStyle *oldStyle,const TextStyle *newStyle,
                Boolean fRedraw,TEHandle hTE);
Boolean        TEContinuousStyle(short *mode,TextStyle *aStyle,TEHandle hTE);
void           TEStyleInsert(const void *text,long length,StScrpHandle hST,TEHandle hTE);
TEStyleHandle  TEGetStyleHandle(TEHandle hTE);
StScrpHandle   TEGetStyleScrapHandle(TEHandle hTE);
void           TEUseStyleScrap(long rangeStart,long rangeEnd,StScrpHandle newStyles,Boolean
               fRredraw,TEHandle hTE);
long           TENumStyles(long rangeStart,long rangeEnd,TEHandle hTE);
```

Using Byte Offsets and Corresponding Points

```
short          TEGetOffset(Point pt,TEHandle hTE);
Point          TEGetPoint(short offset,TEHandle hTE);
```

Customising TextEdit

```
void           TESetClickLoop(TEClickLoopUPP clikProc,TEHandle hTE);;
void           TESetWordBreak(WordBreakProcPtr wBrkProc,TEHandle hTE);;
void           TECustomHook(TEIntHook which,ProcPtr *addr,TEHandle hTE);
```

Additional TextEdit Features

```
short          TEFeatureFlag(short feature,short action,TEHandle hTE);
```

Main Constants, Data Types and Routines Relating to Dates, Times and Numbers

Note: Some of the routines can be accessed using more than one spelling of the routine's name, depending on the interface files supported by your development environment. The new names, as specified in version 2.1 of the Universal Headers, are shown in the following. This table provides a mapping between the previous name of a routine and its new name:

New name	Previous Name		
StringToDate	String2Date		
StringToTime	String2Time		
DateString	IUDatePString	IUDateString	[1]
TimeString	IUTimePString	IUTimeString	[1]
LongDateString	IULDateString		[1]
LongTimeString	IULTimeString		[1]
StringToFormatRec	Str2Format		
FormatRecToString	Format2Str		
StringToExtended	FormatX2Str		
ExtendedToString	FormatStr2X		
DateToSeconds	Date2Secs		
SecondsToDate	Secs2Date		
LongDateToSeconds	LongDate2Secs		
LongSecondsToDate	LongSecs2Date		

[1]Version 2.1 of the Universal Headers states that, these old names are required for PowerPC builds because InterfaceLib exports the old names, not the new names.

Constants

StringToDate and StringToTime Status Values

```
fatalDateTime               = 0x8000  Mask to a fatal error.
longDateFound               = 1 Mask to long date found.
leftOverChars               = 2 Mask to warn of left over characters.
sepNotIntlSep               = 4 Mask to warn of non-standard separators.
fieldOrderNotIntl           = 8 Mask to warn of non-standard field order.
extraneousStrings           = 16  Mask to warn of unparsable strings in text.
tooManySeps                 = 32  Mask to warn of too many separators.
sepNotConsistent            = 64  Mask to warn of inconsistent separators.
tokenErr                    = 0x8100  Mask for 'tokenizer err encountered'.
cantReadUtilities           = 0x8200
dateTimeNotFound            = 0x8400
dateTimeInvalid             = 0x8800
```

FormatResultType Values for Numeric Conversion Functions

```
fFormatOK                  = 0
fBestGuess                 = 1
fOutOfSynch                = 2
fSpuriousChars             = 3
fMissingDelimiter          = 4
fExtraDecimal              = 5
fMissingLiteral            = 6
fExtraExp                  = 7
fFormatOverflow            = 8
fFormStrIsNAN              = 9
fBadPartsTable             = 10
fExtraPercent              = 11
fExtraSeparator            = 12
fEmptyFormatString         = 13
```

Data Types

```
typedef short StringToDateStatus;
typedef SInt8 DateForm;
typedef short FormatStatus;
typedef Sint8 FormatResultType;
```

Data Cache Record

```
struct DateCacheRecord
{
  short   hidden[256];   // Only for temporary use.
};

typedef struct DateCacheRecord DateCacheRecord;
typedef DateCacheRecord *DateCachePtr;
```

Number Format Specification Record

```
struct NumFormatString
{
  UInt8   fLength;
  UInt8   fVersion;
  char    data[254];     // Private data.
};

typedef struct NumFormatString NumFormatString;
```

Routines

Getting Date-Time Values and Records

```
void               GetDateTime(unsigned long *secs);
void               GetTime(DateTimeRec *d);
```

Converting Between Date-Time values and Records

```
void               DateToSeconds(const DateTimeRec *d,unsigned long *secs);
void               SecondsToDate(unsigned long secs,DateTimeRec *d);
void               LongDateToSeconds(const LongDateRec *lDate,LongDateTime *lSecs);
void               LongSecondsToDate(LongDateTime *lSecs,LongDateRec *lDate);
```

Converting Date-Time Strings Into Internal Numeric Representation

```
OSErr              InitDateCache(DateCachePtr theCache);
StringToDateStatus StringToDate(Ptr textPtr,long textLen,DateCachePtr theCache,long
                   *lengthUsed,LongDateRec *dateTime);
StringToDateStatus StringToTime(Ptr textPtr,long textLen,DateCachePtr theCache,long
                   *lengthUsed,LongDateRec *dateTime);
```

Converting Long Date and Time Values Into Strings

```
void               DateString(long dateTime,DateForm longFlag,Str255 result);
void               TimeString(long dateTime,Boolean wantSeconds,Str255 result);
void               LongDateString(LongDateTime *dateTime,DateForm longFlag,Str255
                   result,Handle intlHandle);
void               LongTimeString(LongDateTime *dateTime,Boolean wantSeconds,Str255
                   result,Handle intlHandle);
```

Converting Between Integers and Strings

```
void               StringToNum(ConstStr255Param theString,long *theNum);
void               NumToString(long theNum,Str255 theString);
```

Using Number Format Specification Strings For International Number Formatting

```
FormatStatus       StringToFormatRec(ConstStr255Param inString,const NumberParts
                   *partsTable,NumFormatString *outString)
FormatStatus       FormatRecToString(const NumFormatString *myCanonical,const NumberParts
                   *partsTable,Str255 outString,TripleInt positions)
```

Converting Between Strings and Floating Point Numbers

```
FormatStatus       ExtendedToString(extended80 x,const NumFormatString *myCanonical,const
                   NumberParts *partsTable,Str255 outString)
FormatStatus       StringToExtended(ConstStr255Param source,const NumFormatString
                   *myCanonical,const NumberParts *partsTable,extended80 *x)
```

Demonstration Program 1

```
// ##############################################################################################
// Text1.c
// ##############################################################################################
//
// This program demonstrates:
//
// •  A "bare-bones" monostyled text editor.
//
// •  A Help dialog which features the integrated scrolling of multistyled text and
//    pictures.
//
// In the monostyled text editor demonstration, a panel is displayed at the bottom of all
// opened windows.  This panel displays the edit record length, number of lines, line
// height, destination rectangle (top), scroll bar value, and scroll bar maximum value.
//
// The bulk of the source code for the Help dialog is contained in the file HelpDialog.c.
// The dialog itself displays information intended to assist the user in adapting the
// Help dialog source code and resources to the requirements of his/her own application.
//
// The program utilises the following resources:
//
// •  An 'MBAR' resource, and 'MENU' resources for Apple, File, Edit, and Help dialog
//    pop-up menus (preload, non-purgeable).
//
// •  A 'WIND' resource (purgeable) (initially visible).
//
// •  'CNTL' resources (purgeable) for the vertical scroll bars in the text editor window
//    and Help dialog, and for the pop-up menu in the Help Dialog.
//
// •  An 'ALRT' resource (purgeable), and associated 'DITL' resource (purgeable), for the
//    display of error messages.
//
// •  A 'DLOG' resource (purgeable, initially invisible) and associated 'dctb' resource
//    (purgeable) for the Help dialog.
//
// •  'DITL' resources (purgeable) for the 'ALRT' and 'DLOG' resources.
//
// •  'TEXT' and associated 'styl' resources (all purgeable) for the Help dialog.
//
// •  'PICT' resources (purgeable) for the Help dialog.
//
// •  A 'STR ' resource  (purgeable) containing error text strings.
//
// •  A 'SIZE' resource with the acceptSuspendResumeEvents, doesActivateOnFGSwitch, and
//    is32BitCompatible flags set.
//
// ##############################################################################################

// ……… includes

#include <Fonts.h>
#include <Menus.h>
#include <TextEdit.h>
#include <Dialogs.h>
#include <SegLoad.h>
#include <ToolUtils.h>
#include <Devices.h>
#include <Scrap.h>
#include <StandardFile.h>
#include <Balloons.h>
#include <LowMem.h>

// ……… defines

#define mApple          128
#define  iAbout         1
#define mFile           129
#define  iNew           1
#define  iOpen          2
#define  iClose         4
#define  iSaveAs        6
#define  iQuit          12
#define mEdit           130
#define  iUndo          1
```

```
#define  iCut           3
#define  iCopy          4
#define  iPaste         5
#define  iClear         6
#define  iSelectAll     7
#define kMaxTELength    32767
#define kTab            0x09
#define kDel            0x7F
#define kReturn         0x0D
#define rWindow         128
#define rMenubar        128
#define rVScrollbar     128
#define rErrorAlert     128
#define rErrorStrings   128
#define  eMenuBar       1
#define  eMenu          2
#define  eWindow        3
#define  eDocRecord     4
#define  eEditRecord    5
#define  eExceedChara   6
#define  eNoSpaceCut    7
#define  eNoSpacePaste  8

#define topLeft(r)      (((Point *) &(r))[0])
#define botRight(r)     (((Point *) &(r))[1])

// ……… typedefs

typedef struct
{
  TEHandle      editRecHdl;
  ControlHandle vScrollbarHdl;
} DocRec, **DocRecHandle;

// ……… global variables

Boolean   gDone;
Boolean   gInBackground;
RgnHandle gCursorRegion;
SInt16    gNumberOfWindows = 0;

// ……… function prototypes

void            main                (void);
void            doInitManagers      (void);
void            eventLoop           (void);
void            doIdle              (void);
void            doEvents            (EventRecord *);
void            doKeyEvent          (SInt8);
void            doMouseDown         (EventRecord *);
pascal void     scrollActionProc    (ControlHandle,SInt16);
void            doInContent         (EventRecord *);
void            doUpdate            (EventRecord *);
void            doActivate          (EventRecord *);
void            doActivateDocWindow (WindowPtr,Boolean);
void            doOSEvent           (EventRecord *);
WindowPtr       doNewDocWindow      (void);
pascal Boolean  customClikLoop      (void);
void            setScrollBarValue   (ControlHandle,SInt16 *);
void            doAdjustMenus       (void);
void            doMenuChoice        (SInt32);
void            doFileMenu          (SInt16);
void            doEditMenu          (SInt16);
SInt16          doGetSelectLength   (TEHandle);
void            doAdjustScrollbar   (WindowPtr);
void            doAdjustCursor      (WindowPtr,RgnHandle);
void            doCloseWindow       (WindowPtr);
void            doSaveAsFile        (TEHandle);
void            doOpenCommand       (void);
void            doOpenFile          (FSSpec);
void            doDrawDataPanel     (WindowPtr);
void            doErrorAlert        (SInt16);
void            doHelpMenu          (SInt16);

extern void     doHelp              (void);

// ########################################################################## main
```

```
152
153 void  main(void)
154 {
155   Handle      menubarHdl;
156   MenuHandle  menuHdl;
157   OSErr       osErr;
158
159   // ……initialise managers
160
161   doInitManagers();
162
163   // …… set up menu bar and menus
164
165   menubarHdl = GetNewMBar(rMenubar);
166   if(menubarHdl == NULL)
167     doErrorAlert(eMenuBar);
168   SetMenuBar(menubarHdl);
169   DrawMenuBar();
170
171   menuHdl = GetMenuHandle(mApple);
172   if(menuHdl == NULL)
173     doErrorAlert(eMenu);
174   else
175     AppendResMenu(menuHdl,'DRVR');
176
177   osErr = HMGetHelpMenuHandle(&menuHdl);
178   if(osErr == noErr)
179     AppendMenu(menuHdl,"\pText1 Help");
180   else
181     doErrorAlert(eMenu);
182
183   // …… open an untitled window
184
185   doNewDocWindow();
186
187   // …… enter eventLoop
188
189   eventLoop();
190 }
191
192 // ############################################################################ doInitManagers
193
194 void  doInitManagers(void)
195 {
196   MaxApplZone();
197   MoreMasters();
198
199   InitGraf(&qd.thePort);
200   InitFonts();
201   InitWindows();
202   InitMenus();
203   TEInit();
204   InitDialogs(NULL);
205   InitCursor();
206
207   FlushEvents(everyEvent,0);
208 }
209
210 // ################################################################################ eventLoop
211
212 void  eventLoop(void)
213 {
214   EventRecord eventRec;
215   Boolean     gotEvent;
216   SInt32      sleepTime;
217
218   gDone = false;
219   gCursorRegion = NewRgn();
220   sleepTime = LMGetCaretTime();
221
222   while(!gDone)
223   {
224     gotEvent = WaitNextEvent(everyEvent,&eventRec,sleepTime,gCursorRegion);
225
226     if(!gInBackground && gNumberOfWindows > 0)
227       doAdjustCursor(FrontWindow(),gCursorRegion);
228
```

```
    if(gotEvent)
      doEvents(&eventRec);
    else
    {
      if(gNumberOfWindows > 0)
        doIdle();
    }
  }
}

// ########################################################################### doIdle

void  doIdle(void)
{
  DocRecHandle  docRecHdl;
  WindowPtr     windowPtr;

  windowPtr = FrontWindow();

  docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));
  if(docRecHdl != NULL)
    TEIdle((*docRecHdl)->editRecHdl);
}

// ########################################################################## doEvents

void  doEvents(EventRecord *eventRecPtr)
{
  SInt8   charCode;

  switch(eventRecPtr->what)
  {
    case mouseDown:
      doMouseDown(eventRecPtr);
      break;

    case keyDown:
    case autoKey:
      charCode = eventRecPtr->message & charCodeMask;
      if((eventRecPtr->modifiers & cmdKey) != 0)
      {
        doAdjustMenus();
        doMenuChoice(MenuKey(charCode));
      }
      else
        doKeyEvent(charCode);
      break;

    case updateEvt:
      doUpdate(eventRecPtr);
      break;

    case activateEvt:
      doActivate(eventRecPtr);
      break;

    case osEvt:
      doOSEvent(eventRecPtr);
      break;
  }
}

// ########################################################################## doKeyEvent

void  doKeyEvent(SInt8 charCode)
{
  WindowPtr     windowPtr;
  DocRecHandle  docRecHdl;
  TEHandle      editRecHdl;
  SInt16        selectionLength;

  windowPtr = FrontWindow();
  docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));
  editRecHdl = (*docRecHdl)->editRecHdl;

  if(charCode == kTab)
  {
```

```
306       // Do tab key handling here if required.
307     }
308     else if(charCode == kDel)
309     {
310       selectionLength = doGetSelectLength(editRecHdl);
311       if(selectionLength == 0)
312         (*editRecHdl)->selEnd += 1;
313       TEDelete(editRecHdl);
314       doAdjustScrollbar(windowPtr);
315     }
316     else
317     {
318       selectionLength = doGetSelectLength(editRecHdl);
319       if(((*editRecHdl)->teLength - selectionLength + 1) < kMaxTELength)
320       {
321         TEKey(charCode,editRecHdl);
322         doAdjustScrollbar(windowPtr);
323       }
324       else
325         doErrorAlert(eExceedChara);
326     }
327
328     doDrawDataPanel(windowPtr);
329   }
330
331   // ############################################################################## doMouseDown
332
333   void  doMouseDown(EventRecord *eventRecPtr)
334   {
335     WindowPtr windowPtr;
336     SInt16    partCode;
337
338     partCode = FindWindow(eventRecPtr->where,&windowPtr);
339
340     switch(partCode)
341     {
342       case inMenuBar:
343         doAdjustMenus();
344         doMenuChoice(MenuSelect(eventRecPtr->where));
345         break;
346
347       case inSysWindow:
348         SystemClick(eventRecPtr,windowPtr);
349         break;
350
351       case inContent:
352         if(windowPtr != FrontWindow())
353           SelectWindow(windowPtr);
354         else
355           doInContent(eventRecPtr);
356         break;
357
358       case inDrag:
359         DragWindow(windowPtr,eventRecPtr->where,&qd.screenBits.bounds);
360         break;
361
362       case inGoAway:
363         if(TrackGoAway(windowPtr,eventRecPtr->where))
364           doCloseWindow(FrontWindow());
365         break;
366     }
367   }
368
369   // ############################################################################## scrollActionProc
370
371   pascal void  scrollActionProc(ControlHandle controlHdl,SInt16 partCode)
372   {
373     WindowPtr     windowPtr;
374     DocRecHandle  docRecHdl;
375     TEHandle      editRecHdl;
376     SInt16        linesToScroll;
377     SInt16        controlValue, controlMax;
378
379     if(partCode != 0)
380     {
381       windowPtr = (*controlHdl)->contrlOwner;
382       docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));
```

```
383       editRecHdl = (*docRecHdl)->editRecHdl;
384
385       switch(partCode)
386       {
387         case kControlUpButtonPart:
388         case kControlDownButtonPart:
389           linesToScroll = 1;
390           break;
391
392         case kControlPageUpPart:
393         case kControlPageDownPart:
394           linesToScroll = (((*editRecHdl)->viewRect.bottom - (*editRecHdl)->viewRect.top) /
395                            (*editRecHdl)->lineHeight) -1;
396           break;
397       }
398
399       if((partCode == kControlDownButtonPart) || (partCode == kControlPageDownPart))
400         linesToScroll = -linesToScroll;
401
402       controlValue = GetControlValue(controlHdl);
403       controlMax = GetControlMaximum(controlHdl);
404
405       linesToScroll = controlValue - linesToScroll;
406       if(linesToScroll < 0)
407         linesToScroll = 0;
408       else if(linesToScroll > controlMax)
409         linesToScroll = controlMax;
410
411       SetControlValue(controlHdl,linesToScroll);
412
413       linesToScroll = controlValue - linesToScroll;
414
415       if(linesToScroll != 0)
416         TEScroll(0,linesToScroll * (*editRecHdl)->lineHeight,editRecHdl);
417
418       doDrawDataPanel(windowPtr);
419     }
420   }
421
422   // ############################################################################## doInContent
423
424   void  doInContent(EventRecord *eventRecPtr)
425   {
426     WindowPtr     windowPtr;
427     DocRecHandle  docRecHdl;
428     TEHandle      editRecHdl;
429     Point         mouseXY;
430     ControlHandle controlHdl;
431     SInt16        partCode, controlValue;
432     Boolean       shiftKeyPosition = false;
433
434     windowPtr = FrontWindow();
435     docRecHdl = (DocRecHandle) GetWRefCon(windowPtr);
436     editRecHdl = (*docRecHdl)->editRecHdl;
437
438     mouseXY = eventRecPtr->where;
439     GlobalToLocal(&mouseXY);
440
441     if((partCode = FindControl(mouseXY,windowPtr,&controlHdl)) != 0)
442     {
443       switch(partCode)
444       {
445         case kControlUpButtonPart:
446         case kControlDownButtonPart:
447         case kControlPageUpPart:
448         case kControlPageDownPart:
449           TrackControl(controlHdl,mouseXY,(ControlActionUPP) scrollActionProc);
450           break;
451
452         case kControlIndicatorPart:
453           controlValue = GetControlValue(controlHdl);
454           partCode = TrackControl(controlHdl,mouseXY,NULL);
455           if(partCode != 0)
456           {
457             controlValue -= GetControlValue(controlHdl);
458             if(controlValue != 0)
459               TEScroll(0,controlValue * (*editRecHdl)->lineHeight,editRecHdl);
```

```
460           }
461           doDrawDataPanel(windowPtr);
462           break;
463       }
464     }
465     else if(PtInRect(mouseXY,&(*editRecHdl)->viewRect))
466     {
467       if((eventRecPtr->modifiers & shiftKey) != 0)
468         shiftKeyPosition = true;
469       TEClick(mouseXY,shiftKeyPosition,editRecHdl);
470     }
471   }
472
473
474   // ############################################################################## doUpdate
475
476   void  doUpdate(EventRecord *eventRecPtr)
477   {
478     WindowPtr     windowPtr;
479     DocRecHandle  docRecHdl;
480     TEHandle      editRecHdl;
481     GrafPtr       oldPort;
482
483     windowPtr = (WindowPtr) eventRecPtr->message;
484     docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));
485     editRecHdl = (*docRecHdl)->editRecHdl;
486
487     GetPort(&oldPort);
488     SetPort(windowPtr);
489
490     BeginUpdate((WindowPtr)eventRecPtr->message);
491
492     EraseRect(&windowPtr->portRect);
493     TEUpdate(&windowPtr->portRect,editRecHdl);
494     UpdateControls(windowPtr,windowPtr->visRgn);
495
496     doDrawDataPanel(windowPtr);
497
498     EndUpdate((WindowPtr)eventRecPtr->message);
499
500     SetPort(oldPort);
501   }
502
503   // ############################################################################ doActivate
504
505   void  doActivate(EventRecord *eventRecPtr)
506   {
507     WindowPtr windowPtr;
508     Boolean   becomingActive;
509
510     windowPtr = (WindowPtr) eventRecPtr->message;
511     becomingActive = ((eventRecPtr->modifiers & activeFlag) == activeFlag);
512     doActivateDocWindow(windowPtr,becomingActive);
513   }
514
515   // ######################################################################## doActivateDocWindow
516
517   void  doActivateDocWindow(WindowPtr windowPtr,Boolean becomingActive)
518   {
519     DocRecHandle  docRecHdl;
520     TEHandle      editRecHdl;
521
522     docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));
523     editRecHdl = (*docRecHdl)->editRecHdl;
524
525     if(becomingActive)
526     {
527       SetPort(windowPtr);
528
529       (*editRecHdl)->viewRect.bottom = ((((*editRecHdl)->viewRect.bottom -
530                                           (*editRecHdl)->viewRect.top) /
531                                           (*editRecHdl)->lineHeight) *
532                                           (*editRecHdl)->lineHeight) +
533                                           (*editRecHdl)->viewRect.top;
534       (*editRecHdl)->destRect.bottom = (*editRecHdl)->viewRect.bottom;
535
536       TEActivate(editRecHdl);
```

```
      HiliteControl((*docRecHdl)->vScrollbarHdl,0);
      doAdjustScrollbar(windowPtr);
    }
    else
    {
      TEDeactivate(editRecHdl);
      HiliteControl((*docRecHdl)->vScrollbarHdl,255);
    }
  }

  // ############################################################################## doOSEvent

  void  doOSEvent(EventRecord *eventRecPtr)
  {
    if(((eventRecPtr->message >>  24) & 0x000000FF) == suspendResumeMessage)
    {
      gInBackground = (eventRecPtr->message & resumeFlag) == 0;
      if(gNumberOfWindows > 0)
        doActivateDocWindow(FrontWindow(),!gInBackground);
      HiliteMenu(0);
    }
  }

  // ############################################################################ doNewDocWindow

  WindowPtr  doNewDocWindow(void)
  {
    WindowPtr     windowPtr;
    DocRecHandle  docRecHdl;
    Rect          destAndViewRect;

    if(!(windowPtr = GetNewWindow(rWindow,NULL,(WindowPtr) -1)))
    {
      doErrorAlert(eWindow);
      return(NULL);
    }

    SetPort(windowPtr);

    TextSize(10);
    TextFont(geneva);

    if(!(docRecHdl = (DocRecHandle) NewHandle(sizeof(DocRec))))
    {
      doErrorAlert(eDocRecord);
      return(NULL);
    }
    SetWRefCon(windowPtr,(SInt32) docRecHdl);

    gNumberOfWindows ++;

    (*docRecHdl)->vScrollbarHdl = GetNewControl(rVScrollbar,windowPtr);

    destAndViewRect = windowPtr->portRect;
    destAndViewRect.right -= 15;
    destAndViewRect.bottom -= 15;
    InsetRect(&destAndViewRect,2,2);

    MoveHHi((Handle) docRecHdl);
    HLock((Handle) docRecHdl);

    if(!((*docRecHdl)->editRecHdl = TENew(&destAndViewRect,&destAndViewRect)))
    {
      DisposeWindow(windowPtr);
      gNumberOfWindows --;
      DisposeHandle((Handle) docRecHdl);
      doErrorAlert(eEditRecord);
      return(NULL);
    }

    HUnlock((Handle) docRecHdl);

    TESetClickLoop((TEClickLoopUPP) customClikLoop,(*docRecHdl)->editRecHdl);
    TEAutoView(true,(*docRecHdl)->editRecHdl);
    TEFeatureFlag(teFOutlineHilite,1,(*docRecHdl)->editRecHdl);

    return(windowPtr);
```

```
}

// ############################################################################ customClikLoop

pascal Boolean  customClikLoop(void)
{
  WindowPtr     windowPtr;
  DocRecHandle  docRecHdl;
  TEHandle      editRecHdl;
  GrafPtr       oldPort;
  RgnHandle     oldClip;
  Rect          tempRect;
  Point         mouseXY;
  SInt16        linesToScroll;

  windowPtr = FrontWindow();
  docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));
  editRecHdl = (*docRecHdl)->editRecHdl;

  GetPort(&oldPort);
  SetPort(windowPtr);
  oldClip = NewRgn();
  GetClip(oldClip);
  SetRect(&tempRect,-32767,-32767,32767,32767);
  ClipRect(&tempRect);

  GetMouse(&mouseXY);
  if(mouseXY.v < windowPtr->portRect.top)
  {
    linesToScroll = 1;
    setScrollBarValue((*docRecHdl)->vScrollbarHdl,&linesToScroll);
    if(linesToScroll != 0)
      TEScroll(0,linesToScroll * ((*editRecHdl)->lineHeight),editRecHdl);
  }
  else if(mouseXY.v > windowPtr->portRect.bottom)
  {
    linesToScroll = -1;
    setScrollBarValue((*docRecHdl)->vScrollbarHdl,&linesToScroll);
    if(linesToScroll != 0)
      TEScroll(0,linesToScroll * ((*editRecHdl)->lineHeight),editRecHdl);
  }

  doDrawDataPanel(windowPtr);

  SetClip(oldClip);
  DisposeRgn(oldClip);
  SetPort(oldPort);

  return(true);
}

// ############################################################################ setScrollBarValue

void  setScrollBarValue(ControlHandle controlHdl,SInt16 *linesToScroll)
{
  SInt16  controlValue, controlMax;

  controlValue = GetControlValue(controlHdl);
  controlMax = GetControlMaximum(controlHdl);

  *linesToScroll = controlValue - *linesToScroll;
  if(*linesToScroll < 0)
    *linesToScroll = 0;
  else if(*linesToScroll > controlMax)
    *linesToScroll = controlMax;

  SetControlValue(controlHdl,*linesToScroll);
  *linesToScroll = controlValue - *linesToScroll;
}

// ############################################################################ doAdjustMenus

void  doAdjustMenus(void)
{
  MenuHandle    fileMenuHdl, editMenuHdl;
  WindowPtr     windowPtr;
  DocRecHandle  docRecHdl;
```

```
691     TEHandle      editRecHdl;
692     SInt32        scrapOffset;
693
694     fileMenuHdl = GetMenuHandle(mFile);
695     editMenuHdl = GetMenuHandle(mEdit);
696
697     if(gNumberOfWindows > 0)
698     {
699       windowPtr = FrontWindow();
700       docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));
701       editRecHdl = (*docRecHdl)->editRecHdl;
702
703       EnableItem(fileMenuHdl,iClose);
704
705       if((*editRecHdl)->selStart < (*editRecHdl)->selEnd)
706       {
707         EnableItem(editMenuHdl,iCut);
708         EnableItem(editMenuHdl,iCopy);
709         EnableItem(editMenuHdl,iClear);
710       }
711       else
712       {
713         DisableItem(editMenuHdl,iCut);
714         DisableItem(editMenuHdl,iCopy);
715         DisableItem(editMenuHdl,iClear);
716       }
717
718       if(GetScrap(NULL,'TEXT',&scrapOffset) > 0)
719         EnableItem(editMenuHdl,iPaste);
720       else
721         DisableItem(editMenuHdl,iPaste);
722
723       if((*editRecHdl)->teLength > 0)
724       {
725         EnableItem(fileMenuHdl,iSaveAs);
726         EnableItem(editMenuHdl,iSelectAll);
727       }
728       else
729       {
730         DisableItem(fileMenuHdl,iSaveAs);
731         DisableItem(editMenuHdl,iSelectAll);
732       }
733     }
734     else
735     {
736       DisableItem(fileMenuHdl,iClose);
737       DisableItem(fileMenuHdl,iSaveAs);
738       DisableItem(editMenuHdl,iClear);
739       DisableItem(editMenuHdl,iSelectAll);
740     }
741
742     DrawMenuBar();
743   }
744
745   // ############################################################################## doMenuChoice
746
747   void  doMenuChoice(SInt32 menuChoice)
748   {
749     SInt16  menuID, menuItem;
750     Str255  itemName;
751     SInt16  daDriverRefNum;
752
753     menuID   = HiWord(menuChoice);
754     menuItem = LoWord(menuChoice);
755
756     if(menuID == 0)
757       return;
758
759     switch(menuID)
760     {
761       case mApple:
762         if(menuItem == iAbout)
763           SysBeep(10);
764         else
765         {
766           GetMenuItemText(GetMenuHandle(mApple),menuItem,itemName);
767           daDriverRefNum = OpenDeskAcc(itemName);
```

```
768         }
769         break;
770
771       case mFile:
772         doFileMenu(menuItem);
773         break;
774
775       case mEdit:
776         doEditMenu(menuItem);
777         break;
778
779       case kHMHelpMenuID:
780         if(FrontWindow())
781           doActivateDocWindow(FrontWindow(),false);
782         doHelpMenu(menuItem);
783         break;
784     }
785
786     HiliteMenu(0);
787   }
788
789   // ############################################################################## doFileMenu
790
791   void  doFileMenu(SInt16 menuItem)
792   {
793     WindowPtr     windowPtr;
794     DocRecHandle  docRecHdl;
795     TEHandle      editRecHdl;
796
797     switch(menuItem)
798     {
799       case iNew:
800         if(windowPtr = doNewDocWindow())
801           ShowWindow(windowPtr);
802         break;
803
804       case iOpen:
805         doOpenCommand();
806         break;
807
808       case iClose:
809         doCloseWindow(FrontWindow());
810         break;
811
812       case iSaveAs:
813         docRecHdl = (DocRecHandle) (GetWRefCon(FrontWindow()));
814         editRecHdl = (*docRecHdl)->editRecHdl;
815         doSaveAsFile(editRecHdl);
816         break;
817
818       case iQuit:
819         gDone = true;
820         break;
821     }
822   }
823
824   // ############################################################################## doEditMenu
825
826   void  doEditMenu(SInt16 menuItem)
827   {
828     WindowPtr     windowPtr;
829     DocRecHandle  docRecHdl;
830     TEHandle      editRecHdl;
831     SInt32        totalSize, contigSize, newSize, scrapOffset;
832     SInt16        selectionLength;
833
834     windowPtr = FrontWindow();
835     docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));
836     editRecHdl = (*docRecHdl)->editRecHdl;
837
838     switch(menuItem)
839     {
840       case iUndo:
841         break;
842
843       case iCut:
844         if(ZeroScrap() == noErr)
```

```
      {
        PurgeSpace(&totalSize,&contigSize);
        selectionLength = doGetSelectLength(editRecHdl);
        if(selectionLength > contigSize)
          doErrorAlert(eNoSpaceCut);
        else
        {
          TECut(editRecHdl);
          doAdjustScrollbar(windowPtr);
          if(TEToScrap() != noErr)
            ZeroScrap();
        }
      }
      break;

    case iCopy:
      if(ZeroScrap() == noErr)
      {
        TECopy(editRecHdl);
        if(TEToScrap() != noErr)
          ZeroScrap();
      }
      break;

    case iPaste:
      newSize = (*editRecHdl)->teLength + GetScrap(NULL,'TEXT',&scrapOffset);
      if(newSize > kMaxTELength)
        doErrorAlert(eNoSpacePaste);
      else
      {
        if(TEFromScrap() == noErr)
        {
          TEPaste(editRecHdl);
          doAdjustScrollbar(windowPtr);
        }
      }
      break;

    case iClear:
      TEDelete(editRecHdl);
      doAdjustScrollbar(windowPtr);
      break;

    case iSelectAll:
      TESetSelect(0,(*editRecHdl)->teLength,editRecHdl);
      break;
  }

  doDrawDataPanel(windowPtr);
}

// ########################################################################## doGetSelectLength

SInt16  doGetSelectLength(TEHandle editRecHdl)
{
  SInt16  selectionLength;

  selectionLength = (*editRecHdl)->selEnd - (*editRecHdl)->selStart;
  return(selectionLength);
}

// ########################################################################## doAdjustScrollbar

void  doAdjustScrollbar(WindowPtr windowPtr)
{
  DocRecHandle  docRecHdl;
  TEHandle      editRecHdl;
  SInt16        numberOfLines, controlMax, controlValue;

  docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));
  editRecHdl = (*docRecHdl)->editRecHdl;

  numberOfLines = (*editRecHdl)->nLines;
  if(*(*(*editRecHdl)->hText + (*editRecHdl)->teLength - 1) == kReturn)
    numberOfLines += 1;

  controlMax = numberOfLines - (((*editRecHdl)->viewRect.bottom -
```

```
                (*editRecHdl)->viewRect.top) /
                (*editRecHdl)->lineHeight);
  if(controlMax < 0)
    controlMax = 0;
  SetControlMaximum((*docRecHdl)->vScrollbarHdl,controlMax);

  controlValue = ((*editRecHdl)->viewRect.top - (*editRecHdl)->destRect.top) /
                  (*editRecHdl)->lineHeight;
  if(controlValue < 0)
    controlValue = 0;
  else if(controlValue > controlMax)
    controlValue = controlMax;

  SetControlValue((*docRecHdl)->vScrollbarHdl,controlValue);

  TEScroll(0,((*editRecHdl)->viewRect.top - (*editRecHdl)->destRect.top) -
             (GetControlValue((*docRecHdl)->vScrollbarHdl) *
             (*editRecHdl)->lineHeight),editRecHdl);
}

// ########################################################################## doAdjustCursor

void  doAdjustCursor(WindowPtr windowPtr,RgnHandle mouseRegion)
{
  GrafPtr   oldPort;
  RgnHandle arrowRegion, iBeamRegion;
  Rect      cursorRect;
  Point     mouseXY;

  if(gInBackground)
  {
    SetCursor(&qd.arrow);
    return;
  }

  GetPort(&oldPort);
  SetPort(windowPtr);

  arrowRegion = NewRgn();
  iBeamRegion = NewRgn();
  SetRectRgn(arrowRegion,-32768,-32768,32766,32766);

  cursorRect = windowPtr->portRect;
  cursorRect.bottom -= 15;
  cursorRect.right  -= 15;
  LocalToGlobal(&topLeft(cursorRect));
  LocalToGlobal(&botRight(cursorRect));

  RectRgn(iBeamRegion,&cursorRect);
  DiffRgn(arrowRegion,iBeamRegion,arrowRegion);

  GetMouse(&mouseXY);
  LocalToGlobal(&mouseXY);

  if(PtInRgn(mouseXY,iBeamRegion))
  {
    SetCursor(*(GetCursor(iBeamCursor)));
    CopyRgn(iBeamRegion,mouseRegion);
  }
  else
  {
    SetCursor(&qd.arrow);
    CopyRgn(arrowRegion,mouseRegion);
  }

  DisposeRgn(arrowRegion);
  DisposeRgn(iBeamRegion);

  SetPort(oldPort);
}

// ########################################################################## doCloseWindow

void  doCloseWindow(WindowPtr windowPtr)
{
  DocRecHandle  docRecHdl;

```

```
  docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));

  DisposeControl((*docRecHdl)->vScrollbarHdl);
  TEDispose((*docRecHdl)->editRecHdl);
  DisposeHandle((Handle) docRecHdl);
  DisposeWindow(windowPtr);

  gNumberOfWindows --;
}

// ########################################################################## doSaveAsFile

void  doSaveAsFile(TEHandle editRecHdl)
{
  StandardFileReply fileReply;
  WindowPtr         windowPtr;
  SInt16            fileRefNum;
  SInt32            dataLength;
  Handle            editTextHdl;

  StandardPutFile("\pSave as:","\pUntitled",&fileReply);
  if(fileReply.sfGood)
  {
    windowPtr = FrontWindow();
    SetWTitle(windowPtr,fileReply.sfFile.name);

    if(!(fileReply.sfReplacing))
      FSpCreate(&fileReply.sfFile,' KJB','TEXT',fileReply.sfScript);

    FSpOpenDF(&fileReply.sfFile,fsCurPerm,&fileRefNum);

    dataLength = (*editRecHdl)->teLength;
    editTextHdl = (*editRecHdl)->hText;
    FSWrite(fileRefNum,&dataLength,*editTextHdl);

    FSClose(fileRefNum);
  }
}

// ########################################################################## doOpenCommand

void  doOpenCommand(void)
{
  StandardFileReply fileReply;
  SFTypeList        fileTypes;

  fileTypes[0] = 'TEXT';

  StandardGetFile(NULL,1,fileTypes,&fileReply);
  if(fileReply.sfGood)
    doOpenFile(fileReply.sfFile);
}

// ########################################################################## doOpenFile

void  doOpenFile(FSSpec fileSpec)
{
  WindowPtr     windowPtr;
  DocRecHandle  docRecHdl;
  TEHandle      editRecHdl;
  SInt16        fileRefNum;
  SInt32        textLength;
  Handle        textBuffer;

  if((windowPtr = doNewDocWindow()) == NULL)
    return;

  docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));
  editRecHdl = (*docRecHdl)->editRecHdl;

  SetWTitle(windowPtr,fileSpec.name);

  FSpOpenDF(&fileSpec,fsCurPerm,&fileRefNum);

  SetFPos(fileRefNum,fsFromStart,0);
  GetEOF(fileRefNum,&textLength);

```

```
1076    if(textLength > 32767)
1077      textLength = 32767;
1078
1079    textBuffer = NewHandle((Size) textLength);
1080
1081    FSRead(fileRefNum,&textLength,*textBuffer);
1082
1083    MoveHHi(textBuffer);
1084    HLock(textBuffer);
1085
1086    TESetText(*textBuffer,textLength,editRecHdl);
1087
1088    HUnlock(textBuffer);
1089    DisposeHandle(textBuffer);
1090
1091    FSClose(fileRefNum);
1092
1093    (*editRecHdl)->selStart = 0;
1094    (*editRecHdl)->selEnd = 0;
1095
1096    ShowWindow(windowPtr);
1097  }
1098
1099  // ########################################################################## doDrawDataPanel
1100
1101  void  doDrawDataPanel(WindowPtr windowPtr)
1102  {
1103    DocRecHandle  docRecHdl;
1104    TEHandle      editRecHdl;
1105    ControlHandle controlHdl;
1106    Rect          panelRect;
1107    Str255        textString;
1108
1109    docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));
1110    editRecHdl = (*docRecHdl)->editRecHdl;
1111    controlHdl = (*docRecHdl)->vScrollbarHdl;
1112
1113    TextFont(geneva);
1114    TextSize(9);
1115
1116    MoveTo(3,249);
1117    DrawString("\pteLength           nLines        lineHeight");
1118
1119    MoveTo(193,249);
1120    DrawString("\pdestRect.top           controlValue        contrlMax");
1121
1122    MoveTo(0,238);
1123    LineTo(432,238);
1124    MoveTo(191,239);
1125    LineTo(191,252);
1126    MoveTo(283,239);
1127    LineTo(283,252);
1128    MoveTo(363,239);
1129    LineTo(363,252);
1130
1131    SetRect(&panelRect,42,239,76,252);
1132    EraseRect(&panelRect);
1133    SetRect(&panelRect,106,239,128,252);
1134    EraseRect(&panelRect);
1135    SetRect(&panelRect,174,239,190,252);
1136    EraseRect(&panelRect);
1137    SetRect(&panelRect,250,239,282,252);
1138    EraseRect(&panelRect);
1139    SetRect(&panelRect,342,239,361,252);
1140    EraseRect(&panelRect);
1141    SetRect(&panelRect,412,239,433,252);
1142    EraseRect(&panelRect);
1143
1144    NumToString((SInt32) (*editRecHdl)->teLength,textString);
1145    MoveTo(45,249);
1146    DrawString(textString);
1147
1148    NumToString((SInt32) (*editRecHdl)->nLines,textString);
1149    MoveTo(108,249);
1150    DrawString(textString);
1151
1152    NumToString((SInt32) (*editRecHdl)->lineHeight,textString);
```

```
1153    MoveTo(176,249);
1154    DrawString(textString);
1155
1156    NumToString((SInt32) (*editRecHdl)->destRect.top,textString);
1157    MoveTo(251,249);
1158    DrawString(textString);
1159
1160    NumToString((SInt32) GetControlValue(controlHdl),textString);
1161    MoveTo(344,249);
1162    DrawString(textString);
1163
1164    NumToString((SInt32) GetControlMaximum(controlHdl),textString);
1165    MoveTo(414,249);
1166    DrawString(textString);
1167
1168    TextSize(10);
1169  }
1170
1171  // ############################################################################## doErrorAlert
1172
1173  void  doErrorAlert(SInt16 errorCode)
1174  {
1175    Str255  errorString;
1176
1177    GetIndString(errorString,rErrorStrings,errorCode);
1178    ParamText(errorString,NULL,NULL,NULL);
1179
1180    if(errorCode < eWindow)
1181    {
1182      StopAlert(rErrorAlert,NULL);
1183      ExitToShell();
1184    }
1185    else
1186      CautionAlert(rErrorAlert,NULL);
1187  }
1188
1189  // ################################################################################ doHelpMenu
1190
1191  void  doHelpMenu(SInt16 menuItem)
1192  {
1193    MenuHandle  helpMenuHdl;
1194    SInt16      origHelpItems, numItems;
1195
1196    HMGetHelpMenuHandle(&helpMenuHdl);
1197
1198    numItems = CountMItems(helpMenuHdl);
1199    origHelpItems = numItems - 1;
1200
1201    if(menuItem > origHelpItems)
1202      doHelp();
1203  }
1204
1205  // ##############################################################################################
1206
1207  // ##############################################################################################
1208  // HelpDialog.c
1209  // ##############################################################################################
1210
1211  // ..... includes
1212
1213  #include <Dialogs.h>
1214  #include <ToolUtils.h>
1215  #include <Resources.h>
1216
1217  // ..... defines
1218
1219  #define rHelpModal            129
1220  #define  iOK                  1
1221  #define  iTextUserItem        2
1222  #define  iScrollBar           3
1223  #define  iPopupMenu           4
1224  #define rErrorAlert           128
1225  #define  eHelpDialog          9
1226  #define  eHelpDocRecord       10
1227  #define  eHelpText            11
1228  #define  eHelpPicture         12
1229  #define kTextInset            4
```

```
#define kReturn              0x0D
#define kEnter               0x03

#define rTextIntroduction    128
#define rTextCreatingText    129
#define rTextModifyHelp      130
#define rPictIntroductionBase 128
#define rPictCreatingTextBase 129

// ……… typedefs

typedef struct
{
  Rect          bounds;
  PicHandle     pictureHdl;
} pictInfoRec;

typedef struct
{
  TEHandle      editRecHdl;
  ControlHandle scrollbarHdl;
  SInt16        pictCount;
  pictInfoRec   *pictInfoRecPtr;
} docRecord, ** docRecordHandle;

// ……… global variables

SInt16      gTextResourceID;
SInt16      gPictResourceBaseID;
RgnHandle   gSavedClipRgn = NULL;

// ……… function prototypes

void            doHelp              (void);
void            closeHelp           (DialogPtr,GrafPtr);
pascal void     drawHelp            (DialogPtr,SInt16);
Boolean         getText             (DialogPtr,SInt16,Rect);
Boolean         getPictureInfo      (DialogPtr,SInt16);
void            handleScrollBar     (DialogPtr,SInt16,Point);
pascal void     actionProcedure     (ControlHandle,SInt16);
void            scrollTextAndPicts  (DialogPtr);
void            drawPictures        (DialogPtr,Rect *);
pascal Boolean  helpDialogFilter    (DialogPtr,EventRecord *,SInt16 *);

extern void     doUpdate            (EventRecord *);
extern void     doErrorAlert        (SInt16);

// ########################################################################## doHelp

void  doHelp(void)
{
  DialogPtr       modalDlgPtr;
  docRecordHandle docRecHdl;
  GrafPtr         oldPort;
  SInt16          itemType,itemHit, menuItem;
  Handle          itemHdl;
  Rect            userItemRect, destRect, viewRect, itemRect;

  if(!(modalDlgPtr = GetNewDialog(rHelpModal,NULL,(WindowPtr) -1)))
  {
    doErrorAlert(eHelpDialog);
    return;
  }

  if(!(docRecHdl = (docRecordHandle) NewHandle(sizeof(docRecord))))
  {
    doErrorAlert(eHelpDocRecord);
    DisposeDialog(modalDlgPtr);
    return;
  }

  SetWRefCon(modalDlgPtr,(SInt32) docRecHdl);

  GetPort(&oldPort);
  SetPort(modalDlgPtr);

  GetDialogItem(modalDlgPtr,iTextUserItem,&itemType,&itemHdl,&userItemRect);
```

```
1307    SetDialogItem(modalDlgPtr,iTextUserItem,itemType,(Handle) &drawHelp,&userItemRect);
1308
1309    GetDialogItem(modalDlgPtr,iScrollBar,&itemType,&itemHdl,&itemRect);
1310    (*docRecHdl)->scrollbarHdl = (ControlHandle) itemHdl;
1311
1312    InsetRect(&userItemRect,kTextInset,kTextInset / 2);
1313    destRect = viewRect = userItemRect;
1314    (*docRecHdl)->editRecHdl = TEStyleNew(&destRect,&viewRect);
1315
1316    (*docRecHdl)->pictInfoRecPtr = NULL;
1317
1318    gTextResourceID     = rTextIntroduction;
1319    gPictResourceBaseID = rPictIntroductionBase;
1320
1321    if(!(getText(modalDlgPtr,gTextResourceID,viewRect)))
1322    {
1323      closeHelp(modalDlgPtr,oldPort);
1324      return;
1325    }
1326    if(!(getPictureInfo(modalDlgPtr,gPictResourceBaseID)))
1327    {
1328      closeHelp(modalDlgPtr,oldPort);
1329      return;
1330    }
1331
1332    gSavedClipRgn = NewRgn();
1333
1334    ShowWindow(modalDlgPtr);
1335
1336    do
1337    {
1338      ModalDialog((ModalFilterUPP) &helpDialogFilter,&itemHit);
1339
1340      if(itemHit == iPopupMenu)
1341      {
1342        SetControlValue((*docRecHdl)->scrollbarHdl,0);
1343
1344        GetDialogItem(modalDlgPtr,iPopupMenu,&itemType,&itemHdl,&itemRect);
1345        menuItem = GetControlValue((ControlHandle) itemHdl);
1346
1347        switch(menuItem)
1348        {
1349          case 1:
1350            gTextResourceID     = rTextIntroduction;
1351            gPictResourceBaseID = rPictIntroductionBase;
1352            break;
1353
1354          case 2:
1355            gTextResourceID     = rTextCreatingText;
1356            gPictResourceBaseID = rPictCreatingTextBase;
1357            break;
1358
1359          case 3:
1360            gTextResourceID     = rTextModifyHelp;
1361            break;
1362        }
1363
1364        if(!(getText(modalDlgPtr,gTextResourceID,viewRect)))
1365        {
1366          closeHelp(modalDlgPtr,oldPort);
1367          return;
1368        }
1369        if(!(getPictureInfo(modalDlgPtr,gPictResourceBaseID)))
1370        {
1371          closeHelp(modalDlgPtr,oldPort);
1372          return;
1373        }
1374
1375        drawPictures(modalDlgPtr,&viewRect);
1376      }
1377
1378    } while(itemHit != iOK);
1379
1380    closeHelp(modalDlgPtr,oldPort);
1381
1382    return;
1383  }
```

```
1384
1385  // ############################################################################## closeHelp
1386
1387  void  closeHelp(DialogPtr modalDlgPtr,GrafPtr oldPort)
1388  {
1389    docRecordHandle docRecHdl;
1390    TEHandle        editRecHdl;
1391    SInt16          a;
1392
1393    docRecHdl = (docRecordHandle) GetWRefCon(modalDlgPtr);
1394    editRecHdl = (*docRecHdl)->editRecHdl;
1395
1396    if(gSavedClipRgn)
1397      DisposeRgn(gSavedClipRgn);
1398
1399    if((*docRecHdl)->editRecHdl)
1400      TEDispose((*docRecHdl)->editRecHdl);
1401
1402    if((*docRecHdl)->pictInfoRecPtr)
1403    {
1404      for(a=0;a<(*docRecHdl)->pictCount;a++)
1405        ReleaseResource((Handle) (*docRecHdl)->pictInfoRecPtr[a].pictureHdl);
1406      DisposePtr((Ptr) (*docRecHdl)->pictInfoRecPtr);
1407    }
1408
1409    DisposeHandle((Handle) docRecHdl);
1410    DisposeDialog(modalDlgPtr);
1411
1412    SetPort(oldPort);
1413  }
1414
1415  // ################################################################################ drawHelp
1416
1417  pascal void  drawHelp(DialogPtr modalDlgPtr,SInt16 theItem)
1418  {
1419    PenState        oldPenState;
1420    Handle          itemHdl;
1421    Rect            itemRect, viewRect;
1422    SInt16          itemType, buttonOval;
1423    docRecordHandle docRecHdl;
1424    TEHandle        editRecHdl;
1425
1426    GetPenState(&oldPenState);
1427
1428    GetDialogItem(modalDlgPtr,iTextUserItem,&itemType,&itemHdl,&itemRect);
1429    InsetRect(&itemRect,1,1);
1430
1431    BackColor(whiteColor);
1432    FillRect(&itemRect,&qd.white);
1433    InsetRect(&itemRect,-1,-1);
1434    FrameRect(&itemRect);
1435
1436    TextFont(0);
1437    MoveTo(13,309);
1438    DrawString("\pTopic:");
1439
1440    GetDialogItem(modalDlgPtr,iOK,&itemType,&itemHdl,&itemRect);
1441    InsetRect(&itemRect,-4,-4);
1442
1443    buttonOval = (itemRect.bottom - itemRect.top) / 2 + 2;
1444
1445    ForeColor(blackColor);
1446    PenPat(&qd.black);
1447    PenSize(3,3);
1448    FrameRoundRect(&itemRect,buttonOval,buttonOval);
1449
1450    docRecHdl = (docRecordHandle) GetWRefCon(modalDlgPtr);
1451    editRecHdl = (*docRecHdl)->editRecHdl;
1452    viewRect = (*editRecHdl)->viewRect;
1453
1454    TEUpdate(&viewRect,editRecHdl);
1455    drawPictures(modalDlgPtr,&viewRect);
1456
1457    SetPenState(&oldPenState);
1458  }
1459
1460  // ################################################################################ getText
```

```
1461
1462  Boolean  getText(DialogPtr modalDlgPtr,SInt16 textResourceID,Rect viewRect)
1463  {
1464    docRecordHandle docRecHdl;
1465    TEHandle        editRecHdl;
1466    Handle          helpTextHdl;
1467    StScrpHandle    stylScrpRecHdl;
1468    SInt16          numberOfLines, heightOfText, heightToScroll;
1469
1470    docRecHdl = (docRecordHandle) GetWRefCon(modalDlgPtr);
1471    editRecHdl = (*docRecHdl)->editRecHdl;
1472
1473    TESetSelect(0,32767,editRecHdl);
1474    TEDelete(editRecHdl);
1475
1476    (*editRecHdl)->destRect = (*editRecHdl)->viewRect;
1477    SetControlValue((*docRecHdl)->scrollbarHdl,0);
1478
1479    helpTextHdl = GetResource('TEXT',textResourceID);
1480    if(helpTextHdl == NULL)
1481    {
1482      doErrorAlert(eHelpText);
1483      return false;
1484    }
1485
1486    stylScrpRecHdl = (StScrpHandle) GetResource('styl',textResourceID);
1487    if(stylScrpRecHdl == NULL)
1488    {
1489      doErrorAlert(eHelpText);
1490      return false;
1491    }
1492
1493    TEStyleInsert(*helpTextHdl,GetHandleSize(helpTextHdl),stylScrpRecHdl,editRecHdl);
1494
1495    ReleaseResource(helpTextHdl);
1496    ReleaseResource((Handle) stylScrpRecHdl);
1497
1498    numberOfLines = (*editRecHdl)->nLines;
1499    heightOfText = TEGetHeight((SInt32) numberOfLines,1,editRecHdl);
1500
1501    if(heightOfText >  (viewRect.bottom - viewRect.top))
1502    {
1503      heightToScroll = TEGetHeight((SInt32) numberOfLines,1,editRecHdl) -
1504                                  (viewRect.bottom - viewRect.top);
1505      SetControlMaximum((*docRecHdl)->scrollbarHdl,heightToScroll);
1506      HiliteControl((*docRecHdl)->scrollbarHdl,0);
1507    }
1508    else
1509    {
1510      HiliteControl((*docRecHdl)->scrollbarHdl,255);
1511    }
1512
1513    return true;
1514  }
1515
1516  // ########################################################################## getPictureInfo
1517
1518  Boolean  getPictureInfo(DialogPtr modalDlgPtr,SInt16 firstPictID)
1519  {
1520    docRecordHandle docRecHdl;
1521    TEHandle        editRecHdl;
1522    Handle          textHdl;
1523    SInt32          offset, textSize;
1524    SInt16          numberOfPicts, a, lineHeight, fontAscent;
1525    SInt8           optionSpace[1] = "\xCA";
1526    pictInfoRec     *pictInfoPtr;
1527    Point           picturePoint;
1528    TextStyle       whatStyle;
1529
1530    docRecHdl = (docRecordHandle) GetWRefCon(modalDlgPtr);
1531
1532    if((*docRecHdl)->pictInfoRecPtr != NULL)
1533    {
1534      for(a=0;a<(*docRecHdl)->pictCount;a++)
1535        ReleaseResource((Handle) (*docRecHdl)->pictInfoRecPtr[a].pictureHdl);
1536
1537      DisposePtr((Ptr) (*docRecHdl)->pictInfoRecPtr);
```

```
      (*docRecHdl)->pictInfoRecPtr = NULL;
    }

    (*docRecHdl)->pictCount = 0;

    editRecHdl = (*docRecHdl)->editRecHdl;
    textHdl = (*editRecHdl)->hText;

    textSize = GetHandleSize(textHdl);
    offset = 0;
    numberOfPicts = 0;

    HLock(textHdl);

    offset = Munger(textHdl,offset,optionSpace,1,NULL,0);
    while((offset >= 0) && (offset <= textSize))
    {
      numberOfPicts++;
      offset++;
      offset = Munger(textHdl,offset,optionSpace,1,NULL,0);
    }

    if(numberOfPicts == 0)
    {
      HUnlock(textHdl);
      return true;
    }

    pictInfoPtr = (pictInfoRec *) NewPtr(sizeof(pictInfoRec) * numberOfPicts);
    (*docRecHdl)->pictInfoRecPtr = pictInfoPtr;

    offset = 0L;

    for(a=0;a<numberOfPicts;a++)
    {
      pictInfoPtr[a].pictureHdl = GetPicture(firstPictID + a);
      if(pictInfoPtr[a].pictureHdl == NULL)
      {
        doErrorAlert(eHelpPicture);
        return false;
      }

      offset = Munger(textHdl,offset,optionSpace,1,NULL,0);
      picturePoint = TEGetPoint((SInt16)offset,editRecHdl);

      TEGetStyle(offset,&whatStyle,&lineHeight,&fontAscent,editRecHdl);
      picturePoint.v -= lineHeight;
      offset++;
      pictInfoPtr[a].bounds = (**pictInfoPtr[a].pictureHdl).picFrame;

      OffsetRect(&pictInfoPtr[a].bounds,
                 (((*editRecHdl)->destRect.right + (*editRecHdl)->destRect.left) -
                 (pictInfoPtr[a].bounds.right + pictInfoPtr[a].bounds.left) ) / 2,
                 - pictInfoPtr[a].bounds.top + picturePoint.v);
    }

    (*docRecHdl)->pictCount = a;

    HUnlock(textHdl);

    return true;
  }

  // ############################################################################ handleScrollBar

  void  handleScrollBar(DialogPtr modalDlgPtr,SInt16 thePart,Point mouseXY)
  {
    docRecordHandle   docRecHdl;

    docRecHdl = (docRecordHandle) GetWRefCon(modalDlgPtr);

    if(thePart == kControlIndicatorPart)
    {
      if(TrackControl((*docRecHdl)->scrollbarHdl,mouseXY,NULL))
        scrollTextAndPicts(modalDlgPtr);
    }
    else
```

```
    TrackControl((*docRecHdl)->scrollbarHdl,mouseXY,(ControlActionUPP) &actionProcedure);
}

// ########################################################################## actionProcedure

pascal void  actionProcedure(ControlHandle scrollbarHdl,SInt16 partCode)
{
  docRecordHandle docRecHdl;
  DialogPtr       modalDlgPtr;
  TEHandle        editRecHdl;
  SInt16          delta, oldValue, offset, lineHeight, fontAscent;
  Point           thePoint;
  Rect            viewRect;
  TextStyle       style;

  if(partCode)
  {
    modalDlgPtr = (*scrollbarHdl)->contrlOwner;
    docRecHdl = (docRecordHandle) GetWRefCon(modalDlgPtr);
    editRecHdl = (*docRecHdl)->editRecHdl;
    viewRect = (*editRecHdl)->viewRect;
    thePoint.h = viewRect.left + kTextInset;

    switch(partCode)
    {
      case kControlUpButtonPart:
        thePoint.v = viewRect.top - 4;
        offset = TEGetOffset(thePoint,editRecHdl);
        thePoint = TEGetPoint(offset,editRecHdl);
        TEGetStyle(offset,&style,&lineHeight,&fontAscent,editRecHdl);
        delta = thePoint.v - lineHeight - viewRect.top;
        break;

      case kControlDownButtonPart:
        thePoint.v = viewRect.bottom + 2;
        offset = TEGetOffset(thePoint,editRecHdl);
        thePoint = TEGetPoint(offset,editRecHdl);
        delta = thePoint.v - viewRect.bottom;
        break;

      case kControlPageUpPart:
        thePoint.v = viewRect.top + 2;
        offset = TEGetOffset(thePoint,editRecHdl);
        thePoint = TEGetPoint(offset,editRecHdl);
        TEGetStyle(offset,&style,&lineHeight,&fontAscent,editRecHdl);
        thePoint.v += lineHeight - fontAscent;
        thePoint.v -= viewRect.bottom - viewRect.top;
        offset = TEGetOffset(thePoint,editRecHdl);
        thePoint = TEGetPoint(offset,editRecHdl);
        TEGetStyle(offset,&style,&lineHeight,&fontAscent,editRecHdl);
        delta = thePoint.v - viewRect.top;
        if(offset == 0)
          delta -= lineHeight;
        break;

      case kControlPageDownPart:
        thePoint.v = viewRect.bottom - 2;
        offset = TEGetOffset(thePoint,editRecHdl);
        thePoint = TEGetPoint(offset,editRecHdl);
        TEGetStyle(offset,&style,&lineHeight,&fontAscent,editRecHdl);
        thePoint.v -= fontAscent;
        thePoint.v += viewRect.bottom - viewRect.top;
        offset = TEGetOffset(thePoint,editRecHdl);
        thePoint = TEGetPoint(offset,editRecHdl);
        TEGetStyle(offset,&style,&lineHeight,&fontAscent,editRecHdl);
        delta =  thePoint.v - lineHeight - viewRect.bottom;
        if(offset == (**editRecHdl).teLength)
          delta += lineHeight;
        break;
    }

    oldValue = GetControlValue(scrollbarHdl);

    if(((delta < 0) && (oldValue > 0)) || ((delta > 0) &&
       (oldValue < GetControlMaximum(scrollbarHdl))))
    {
      GetClip(gSavedClipRgn);
```

```
1692        ClipRect(&modalDlgPtr->portRect);
1693
1694        SetControlValue(scrollbarHdl,oldValue + delta);
1695        SetClip(gSavedClipRgn);
1696      }
1697
1698      scrollTextAndPicts(modalDlgPtr);
1699    }
1700  }
1701
1702  // ########################################################################## scrollTextAndPicts
1703
1704  void  scrollTextAndPicts(DialogPtr modalDlgPtr)
1705  {
1706    docRecordHandle docRecHdl;
1707    TEHandle        editRecHdl;
1708    SInt16          scrollDistance, oldScroll;
1709    Rect            updateRect;
1710
1711    docRecHdl = (docRecordHandle) GetWRefCon(modalDlgPtr);
1712    editRecHdl = (*docRecHdl)->editRecHdl;
1713
1714    oldScroll = (*editRecHdl)->viewRect.top -(**editRecHdl).destRect.top;
1715    scrollDistance = oldScroll - GetControlValue((*docRecHdl)->scrollbarHdl);
1716    if(scrollDistance == 0)
1717      return;
1718
1719    TEScroll(0,scrollDistance,editRecHdl);
1720
1721    if((*docRecHdl)->pictCount == 0)
1722      return;
1723
1724    updateRect = (*editRecHdl)->viewRect;
1725
1726    if(scrollDistance > 0)
1727    {
1728      if(scrollDistance < (updateRect.bottom - updateRect.top))
1729        updateRect.bottom = updateRect.top + scrollDistance;
1730    }
1731    else
1732    {
1733      if( - scrollDistance < (updateRect.bottom - updateRect.top))
1734        updateRect.top = updateRect.bottom + scrollDistance;
1735    }
1736
1737    drawPictures(modalDlgPtr,&updateRect);
1738  }
1739
1740  // ############################################################################## drawPictures
1741
1742  void  drawPictures(DialogPtr modalDlgPtr,Rect *updateRect)
1743  {
1744    docRecordHandle docRecHdl;
1745    TEHandle        editRecHdl;
1746    SInt16          pictCount, pictIndex, vOffset;
1747    PicHandle       thePictHdl;
1748    Rect            pictLocRect, dummyRect;
1749
1750    docRecHdl = (docRecordHandle) GetWRefCon(modalDlgPtr);
1751    editRecHdl = (*docRecHdl)->editRecHdl;
1752
1753    vOffset = (*editRecHdl)->destRect.top - (*editRecHdl)->viewRect.top - kTextInset;
1754    pictCount = (*docRecHdl)->pictCount;
1755
1756    for(pictIndex = 0;pictIndex < pictCount;pictIndex++)
1757    {
1758      pictLocRect = (*docRecHdl)->pictInfoRecPtr[pictIndex].bounds;
1759      OffsetRect(&pictLocRect,0,vOffset);
1760
1761      if(!SectRect(&pictLocRect,updateRect,&dummyRect))
1762        continue;
1763
1764      thePictHdl = (*docRecHdl)->pictInfoRecPtr[pictIndex].pictureHdl;
1765
1766      LoadResource((Handle) thePictHdl);
1767      HLock((Handle) thePictHdl);
1768
```

```
1769      GetClip(gSavedClipRgn);
1770      ClipRect(updateRect);
1771      DrawPicture(thePictHdl,&pictLocRect);
1772
1773      SetClip(gSavedClipRgn);
1774      HUnlock((Handle) thePictHdl);
1775    }
1776  }
1777
1778  // ########################################################################## helpDialogFilter
1779
1780  pascal  Boolean helpDialogFilter(DialogPtr modalDlgPtr, EventRecord *eventRecPtr,
1781                                   SInt16 *itemHit)
1782  {
1783    docRecordHandle docRecHdl;
1784    SInt16          handledEvent, itemType, thePart;
1785    SInt8           charCode;
1786    Rect            itemRect;
1787    Handle          itemHdl;
1788    Point           mouseXY;
1789    SInt32          finalTicks;
1790    ControlHandle   controlHdl;
1791
1792    handledEvent = false;
1793
1794    if((eventRecPtr->what == updateEvt) && ((WindowPtr) eventRecPtr->message != modalDlgPtr))
1795    {
1796      doUpdate(eventRecPtr);
1797    }
1798    else
1799    {
1800      switch(eventRecPtr->what)
1801      {
1802        case keyDown:
1803        case autoKey:
1804          charCode = eventRecPtr->message & charCodeMask;
1805          if((charCode == (SInt8) kReturn) || (charCode == (SInt8) kEnter))
1806          {
1807            GetDialogItem(modalDlgPtr,iOK,&itemType,&itemHdl,&itemRect);
1808            HiliteControl((ControlHandle) itemHdl,inButton);
1809            Delay(10,&finalTicks);
1810            HiliteControl((ControlHandle) itemHdl,0);
1811            handledEvent = true;
1812            *itemHit = iOK;
1813          }
1814          break;
1815
1816        case mouseDown:
1817          mouseXY = eventRecPtr->where;
1818          GlobalToLocal(&mouseXY);
1819          thePart = FindControl(mouseXY,modalDlgPtr,&controlHdl);
1820          docRecHdl = (docRecordHandle) GetWRefCon(modalDlgPtr);
1821          if(controlHdl == (*docRecHdl)->scrollbarHdl)
1822          {
1823            handleScrollBar(modalDlgPtr,thePart,mouseXY);
1824            *itemHit = iScrollBar;
1825            handledEvent = true;
1826          }
1827          break;
1828      }
1829    }
1830
1831    return handledEvent;
1832  }
1833
1834  //#############################################################################################
```

Demonstration Program 1 Comments

When this program is run, the user should explore both the text editor and the Help dialog.

Text Editor

In the text editor, the user should perform all the actions usually associated with a simple text editor, that is:

- Open a new document window, open an existing 'TEXT' file for display in a new document window, and save a document to a 'TEXT' file.
- Enter new text and use the Edit menu Cut, Copy, Paste, and Clear commands to edit the text. (Pasting between documents and other applications is supported.)
- Select text by clicking and dragging, double-clicking a word, shift-clicking, and choosing the Select All command from the Edit menu. Also select large amounts of text by clicking in the text and dragging the cursor above or below the window so as to invoke auto-scrolling.
- Scroll a large document by dragging the scroll box, clicking once in a scroll arrow or gray area, and holding the mouse down in a scroll arrow or gray area.

Whenever any action is taken, the user should observe the changes to the values displayed in the data panel at the bottom of each window. In particular, the relationship between the destination rectangle and scroll bar control value should be noted.

The user should also note that outline highlighting is activated for all windows and that the del key is supported by the application. (The del key is not supported by TextEdit.)

Help Dialog

The user should choose Text1 Help from the Help menu to open the Help dialog and then scroll through the three help topics, which may be chosen in the pop-up menu at the bottom of the dialog. The help topics contain documentation on the Help dialog which supplements the source code comments below.

Text1.c

#define

Lines 65-79 establish constants relating to menu IDs and menu item numbers. Line 80 establishes a constant for the maximum allowable number of bytes in a TextEdit edit record. Lines 81-83 establish constants representing the character codes generated by the tab, delete and return keys. Lines 84-88 establish constants relating to various resources. The constants at Lines 89-96 are used to index a 'STR#' resource so that specified error strings can be retrieved. Lines 98-99 define two common macros. The first converts the top and left fields of a Rect to a Point. The second converts the bottom and right field of a Rect to a Point.

#typedef

The DocRec data type will be used for a small document record comprising a handle to an edit record and a handle to a (vertical) scroll bar.

Global variable

The global variable gDone controls program termination. gInBackground relates to foreground/background switching. gCursorRegion is related to the WaitNextEvent's mouseRgn parameter. gNumberOfWindows will keep track of the number of windows open at any one time.

main

The main function initialises the system software managers, sets up the menus, opens a new document window and enters the event loop.

eventLoop

eventLoop is the main event loop. At line 218, the variable which controls program termination is set to false. Line 219 creates a new empty region for the cursorRgn parameter of the WaitNextEvent function. Line 220 gets the current caret blink interval and assigns it to the variable to be passed as the sleep parameter in the WaitNextEvent call.

The while loop entered at Line 222 continues until gDone is set to true. Whenever an event is retrieved, the function which changes the cursor shape is called provided the application is not in the background and at least one window is open(Lines 226-227). If the event is a null event, and provided at least one window is open, the application-defined function doIdle is called (Lines 231-234). Otherwise, the main event processing function is called.

doIdle

doIdle is invoked whenever a NULL event is received.

Line 246 gets a pointer to the front window, allowing Line 248 to attempt to retrieve a handle to that window's document record. If Line 248 was successful, Line 250 calls TEIdle to blink the insertion point.

doEvents

doEvents handles initial event processing. Note that, in the case of mouse-down events, the application-defined function doMouseDown is called and that, in the case of key events (Lines 265-275), the application-defined function doKeyEvent is called provided the event is not a Command key equivalent. Other processing is minimal consistent with the demonstration aspects of the program.

doKeyEvent

doKeyEvent handles all key-down events that are not Command key equivalents. Lines 300-302 get the pointer to the front window, a handle to the document record attached to the window, and a handle to the TextEdit edit record which forms part of the document record.

Lines 304 filters out the tab key character code. (TextEdit does not support the tab key and some applications may need to provide a tab key handler.)

Line 308 filters out the del key character code. TextEdit does not recognise this key, so Lines 310-314 provide del key support for the program. Line 310 gets the current selection length from the edit record. If this is zero (that is, there is no selection range and an insertion point is being displayed), the selEnd field is increased by one (Lines 311-312). This, in effect, creates a selection range comprising the character following the insertion point. Line 313 deletes the current selection range from the edit record. Such deletions could change the number of text lines in the edit record, requiring the vertical scroll bar to be adjusted; hence the call to the associated application-defined function at Line 314.

Processing of those character codes which have not been filtered out is initiated at Line 316. A new character must not be allowed to be inserted in the edit record if the TextEdit limit of 32,767 characters will be exceeded. Accordingly, and given that TEKey replaces the selection range with the character passed to it, the first step is to get the current selection length (Line 318). If the current length of the edit record minus the selection length plus 1 is less than 32,767, the character code is passed to TEKey (Line 321) for insertion into the edit record. In addition, and since all this could change the number of lines in the edit record, the scroll bar adjustment function is called (Line 322).

If the TextEdit limit will be exceeded by accepting the character, an alert box is invoked advising the user of the situation (Lines 324-325).

Line 328 calls the application-defined function which prints data extracted from the edit and control records at the bottom of the window.

doMouseDown

doMouseDown performs initial processing of mouse-down events. Note that, at Lines 351-356, a mouse-down in the content region of a window will result in a call to the application-defined function doInContent provided the window is the front window.

scrollActionProc

scrollActionProc is associated with the vertical scroll bar. It is the hook procedure which will be repeatedly called by TrackControl (see Line 449) while the mouse button remains down in a scroll arrow or gray area of the vertical scroll bar.

Line 381 gets the pointer to the window which "owns" the control. Lines 382-383 get a handle to the edit record associated with the window.

The purpose of the switch initiated at Line 385 is to get a value into the variable linesToScroll. If the mouse-down was in a scroll arrow, that value will be 1 (387-390). If the mouse-down was in a gray area, that value will be equivalent to one less than the number of text lines that will fit in the view rectangle (Lines 392-396. (Subtracting 1 from the total number of lines that will fit in the view rectangle ensures that the line of text at the bottom/top of the view rectangle prior to a gray area scroll will be visible at the top/bottom of the window after the scroll.)

Lines 399-400 change the value in linesToScroll to a negative value if the mouse-down occurred in either the down scroll arrow or down gray area.

Lines 402-403 assign the current control value and the current control maximum value to two variables.

Line 405-409 ensure that no scrolling action will occur if the document is currently scrolled fully up (control value equals control maximum) or fully down (control value equals 0). In either case, linesToScroll will be set to 0, meaning that the call to TEScroll at Line 416 will not occur.

Line 411 sets the control value to the value calculated at Line 405, that is, to the current control value minus the value in linesToScroll.

Line 413 sets the value in linesToScroll back to what it was before Line 405 executed. This value, multiplied by the value in the lineHeight field of the edit record, is then passed to TEScroll as the parameter which specifies the number of pixels to scroll (Line 416).

Line 418 is for demonstration purposes only. It calls the application-defined function which prints data extracted from the edit and control records at the bottom of the window.

doInContent

doInContent continues mouse-down processing. Lines 434-436 get the pointer to the front window and a handle to the edit record associated with that window.

Lines 438-439 convert the mouse-down coordinates from global to local coordinates. (Local coordinates will be required by upcoming calls to FindControl, TrackControl, and PtInRect.)

If the mouse-down was in a control (that is, the scroll bar), Line 443 initiates a switch based on the control part code returned by FindControl at Line 441. If the mouse-down was in one of the scroll arrows or gray areas, TrackControl is called (Line 449). TrackControl retains control until the mouse button is released, during which time the previously described hook procedure scrollActionProc is repeatedly called. If the mouse-down was in the control box (Line 452), the old control value is saved (Line 453) before TrackControl is called (Line 454) to retain control until the mouse button is released. If the mouse button is released with the cursor still inside the scroll box (Line 455), the new control value is retrieved and subtracted from the old control value. If the old and new control values are not the same, TEScroll is called to scroll the text by the appropriate number of pixels (Lines 457-458) and Line 461 updates the data panel.

If the mouse-down was not in a control, Line 465 checks whether it occurred in the view rectangle. (Note that the view rectangle is in local coordinates, so the mouse-down coordinates passed as the first parameter to the PtInRect call must also be in local coordinates.) If the mouse-down was in the view rectangle, a check is made of the shift key position at the time of the mouse-down. The result is passed as the second parameter in the call to TEClick call at Line 469. (TEClick's behaviour depends on the position of the shift key.)

doUpdate

doUpdate handles update events. Lines 483-485 get the window pointer and a handle to the edit record associated with the window. Lines 487-488 save the current graphics port before setting the current graphics port to that associated with the window to be updated.

Between the usual BeginUpdate and EndUpdate calls, the port rectangle is erased, TEUpdate is called to draw the text in the edit record, UpdateControls is called to draw the scroll bar, and the data panel is redrawn (Lines 492-494). Line 500 resets the saved graphics port as the current port.

doActivate

doActivate handles initial processing of activate events. It sets a flag according to whether the window in question is about to come to the foreground or be sent to the background. It then calls an application-defined function which handles window activation/deactivation.

doActivateDocWindow

doActivateDocWindow handles window activation/deactivation.

Lines 522-523 retrieve a handle to the edit record for the window.

If the window is becoming active (Line 525), its graphics port is set as the current graphics port (Line 527). The bottom of the view rectangle is then adjusted so that the height of the view rectangle is an exact multiple of the value in the lineHeight field of the edit record. (This avoids the possibility of only part of the full height of a line of text appearing at the bottom of the view rectangle.) Line 536 activates the edit record associated with the window, Line 537 activates the scroll bar, and Line 538 adjusts the scroll bar.

If the window is becoming inactive (Line 540), Line 548 deactivates the edit record associated with the window and Line 549 deactivates the scroll bar.

doOSEvent

doOSEvent handles operating system events. If the event is a suspend or resume event and at least one window is open, the application-defined function doActivateWindow is called.

doNewDocWindow

doNewDocWindow is called at program launch and when the user chooses New or Open from the File menu. It opens a new window, attaches a document record to that window, creates a vertical scroll bar, creates a monostyled edit record, installs a custom click loop procedure (see below), enables automatic scrolling, and enables outline highlighting.

Line 568 opens a new window and Line 574 sets its graphics port as the current graphics port. (Since the edit record assumes the drawing environment of the graphics port, setting the graphics port must be done before the call to TENew to create the edit record.)

Lines 576-577 set the text size and font. (These will be copied from the graphics port to the edit record when TENew is called.)

Line 579 creates a document record and Line 584 assigns a handle to that record to the window record's refCon field. Line 586 increments the global variable which keeps track of the number of open windows. Line 588 creates a vertical scroll bar and assigns a handle to it to the appropriate field of the document record. Lines 590-593 establish the view and destination rectangles two pixels inside the window's port rectangle less the scroll bar.

Lines 595-596 move the document record is high and locked it. A monostyled edit record is then created and its handle is assigned to the appropriate field of the document record (Line 598). (If this call is not successful, the window and scroll bar are disposed of, an error alert is displayed, and the function returns (Lines 600-604).) The handle to the document record is then unlocked (Line 607).

Line 609 installs the address of the custom click loop routine customClikLoop in the clikLoop field of the edit record. Line 610 enables automatic scrolling for the edit record. Line 611 enables outline highlighting for the edit record.

Line 613 returns a pointer to the newly opened window.

customClikLoop

customClikLoop replaces the default click loop routine so as to provide for scroll bar adjustment in concert with automatic scrolling. customClikLoop is thus called repeatedly by TEClick as long as the mouse button is held down within the view rectangle.

Lines 629-631 get a pointer to the front window and a handle to the edit record associated with that window. Lines 633-634 save the current graphics port and set the window's graphics port as the current port.

The window's current clip region will have been set by TextEdit to be equivalent to the view rectangle. Since the scroll bar has to be redrawn, the clipping region must be temporarily reset to include the scroll bar. Accordingly, Line 635-636 saves the current clipping region before Lines 637-638 set the clipping region to the bounds of the coordinate plane. That done, Line 640 gets the current position of the cursor.

If the cursor is above the top of the window (Line 641), the text must be scrolled downwards. Accordingly, the variable linesToScroll is set to 1 (Line 643). The subsidiary function setScrollBarValue (see below) is then called to, amongst other things, reset the scroll bar's value. Note that the value in linesToScroll may be modified by setScrollBarValue. If linesToScroll is not set to 0 by setScrollBarValue (Line 645), TEScroll is called at Line 646 to scroll the text by a number of pixels equivalent to the value in the lineHeight field of the edit record, and in a downwards direction.

If the cursor is below the bottom of the window (Line 648), the same process occurs except that the variable linesToScroll is set to -1, thus causing an upwards scroll of the text (assuming that the value in linesToScroll is not changed to 0 by setScrollBarValue).

Line 656 redraws the data panel. Line 658 restores the clipping region to that established by the view rectangle and Line 660 restores the saved graphics port. Finally, Line 662 returns true. (A return of false would cause TextEdit to stop calling customClikLoop, as if the user had released the mouse button.)

setScrollBarValue

setScrollBarValue is called from customClikLoop. Apart from setting the scroll bar's value so as to cause the scroll box to follow up automatic scrolling, the function checks whether the limits of scrolling have been reached.

Lines 671-672 get the current control value and the current control maximum value. At Lines 674-678, the value in the variable linesToScroll will be set to either 0 (if the current control value is 0) or equivalent to the control maximum value (if the current control value is equivalent to the control maximum value. If these modifications do not occur, the value in linesToScroll will remain as established at Line 674, that is, the current control value minus the value in linesToScroll as passed to the function.

Line 680 sets the control's value to the value in linesToScroll. Line 681 sets the value in linesToScroll to 0 if the limits of scrolling have already been reached, or to the value as it was when the setScrollBarValue function was entered.

doAdjustMenus

doAdjustMenus adjusts the menus. Much depends on whether any windows are currently open (Line 697).

If at least one window is open, Lines 699-701 get a handle to the edit record associated with the front window and Line 703 enables the Close item. If there is a current selection range (Line 705), the Cut, Copy, and Clear items are enabled, otherwise they are disabled. If there is any text in the desk scrap (Line 718), the Paste item is enabled, otherwise it is disabled. If there is any text in the edit record (Line 723), the SaveAs and Select All items are enabled, otherwise they are disabled.

If no windows are open (Line 734), the Close, SaveAs, Clear, and Select All items are disabled.

doMenuChoice

doMenuChoice performs initial menu choice handling. It processes Apple menu choices to completion but passes File, Edit, and Help menu choices to subsidiary functions.

doFileMenu

doFileMenu handles File menu choices, calling the appropriate application-defined functions according to the menu item chosen. In the case of SaveAs (Lines 812), a handle to the edit record associated with the front window is retrieved and passed as a parameter to the appropriate function.

(Note that, because TextEdit, rather than file operations, is the real focus of this program, the file-related code has been kept to a minimum, even to the extent of having no Save-related, as opposed to SaveAs-related, code.)

doEditMenu

doEditMenu handles choices from the Edit menu. Recall that, in the case of monostyled edit records, TECut, TECopy, and TEPaste do not copy/paste text to/from the desk scrap. This program, however, supports copying/pasting to/from the desk scrap.

Before the usual switch is entered, Lines 834-836 get a pointer to the front window and a handle to the edit record associated with that window.

Lines 843-858 handle the Cut command. Firstly, the call to ZeroScrap attempts to empty the desk scrap. If the call succeeds, Line 846 establishes the size of the largest block in the heap that would be available if a general purge were to occur. Line 847 gets the current selection length. If the selection length is greater than the available memory (Line 848), the user is advised via an error message (Line 849). Otherwise, TECut is called at Line 852 to remove the selected text from the edit record and copy it to the TextEdit private scrap. The scroll bar is adjusted (Line 853), and TEToScrap is called at Line 854 to copy the private scrap to the desk scrap. If the latter call is not successful, Line 855 cleans up as best it can by emptying the desk scrap.

Lines 860-867 handle the Copy command. If the call to empty the desk scrap (Line 861) is successful, TECopy is called to copy the selected text from the edit record to the TextEdit private scrap. Line 864 then copies the private scrap to the desk scrap.

Lines 869-881 handle the Paste command, which must not proceed if the paste would cause the TextEdit limit of 32,767 bytes to be exceeded. Line 870 establishes a value equal to the number of bytes in the edit record plus the number of bytes of the specified data type ('TEXT') in the desk scrap. If this value exceeds the TextEdit limit, the user is advised via an error message (Lines 871-872). Otherwise, Line 875 copies the desk scrap to TextEdit's private scrap, Line 877 inserts the private scrap into the edit record, and Line 878 adjusts the scroll bar.

Lines 883-886 handle the Clear command. Line 884 deletes the current selection range from the edit record and Line 885 adjusts the scroll bar.

Lines 888-890 handle the Select All command. TESetSelect sets the selection range according to the first two parameters (selStart and selEnd).

doGetSelectLength

doGetSelectLength returns a value equal to the length of the current selection.

doAdjustScrollbar

doAdjustScrollbar adjusts the vertical scroll bar.

Lines 914-915 retrieve handles to the document record and edit record associated with the window in question.

Line 917 assigns the value in the nLines field of the edit record to the numberOfLines variable. The next action (Lines 918-919) is somewhat of a refinement and is therefore not essential. If the last character in the edit record is the return character, numberOfLines is incremented by one. This will ensure that, when the document is scrolled to its end, a single blank line will appear below the last line of text.

Line 921 assigns to the variable controlMax a value equal to the number of lines in the edit record less the number of lines which will fit in the view rectangle. If this value is less than 0 (indicating that the number of lines in the edit record is less than the number of lines that will fit in the view rectangle), controlMax is set to 0. Line 926 then sets the control maximum value. If controlMax is 0, the scroll bar is automatically unhighlighted by the SetControlMaximum call.

Line 928 assigns to the variable controlValue a value equal to the number of text lines that the top of the destination rectangle is currently "above" the top of the view rectangle. If the calculation returns a value less than 0 (that is, the document has been scrolled fully down), controlValue is set to 0. If the calculation returns a value greater than the current control maximum value (that is, the document has been scrolled fully up), controlValue is set to equal that value. Line 935 sets the control value to the value in controlValue. For example, if the top of the view rectangle is 2, the top of the destination rectangle is -34 and the lineHeight field of the edit record contains the value 13, the control value will be set to 3.

With the control maximum value and the control value set, TEScroll is called at Line 937 to make sure the text is scrolled to the position indicated by the scroll box. Extending the example in the previous paragraph, the second parameter in the TEScroll call is 2 - (34 - (3 * 13)), that is, 0. In that case, no corrective scrolling actually occurs.

doAdjustCursor

doAdjustCursor adjusts the cursor to the I-Beam shape when the cursor is over the content region less the scroll bar area, and to the arrow shape when the cursor is outside that region. It is similar to the cursor adjustment function in the demonstration program at Chapter 12 — Offscreen Graphics Worlds, Pictures, Cursors, and Icons.

doCloseWindow

doCloseWindow disposes of the specified window. The associated scroll bar, the associated edit record and the associated document record are disposed of before the call to DisposeWindow.

doSaveAsFile, doOpenCommand, doOpenFile

The functions doSaveAsFile, doOpenCommand, and doOpenFile are document saving and opening functions, enabling the user to open and save 'TEXT' documents. Since the real focus of this program is TextEdit, not file operations, the code is "bare bones" and as brief as possible.

> For a complete example of opening and saving monostyled 'TEXT' documents, see the demonstration program at Chapter 14 — Files.

doDrawDataPanel

doDrawDataPanel draws the data panel at the bottom of each window. Displayed in this panel are the values in the teLength, nLines, lineHeight and destRect.top fields of the edit record and the contrlValue and contrlMax fields of the scroll bar's control record.

doErrorAlert

doErrorAlert retrieves the appropriate error message string from a 'STR#' resource and invokes either a stop alert or a caution alert depending on the severity of the error.

doHelpMenu

doHelpMenu handles a choice of the Text1 Help item added by this program to the system-managed Help Menu. The code reflects the fact that Apple reserves the right to add items to the Help menu in future versions of the system software.

At Line 1196, HMGetHelpMenuHandle gets a handle to the Help menu record. At Line 1198, the call to CountMItems returns the number of items in the Help menu. Since we know that we have added one item to this menu, Line 1199 will establish the original number of help items. If the value passed to the doHelpMenu function is greater than this (Line 1201), it must therefore represent the item number of our Text1 Help item, in which case the application-defined function which opens the help dialog is called (Line 1202).

HelpDialog.c

#define

Lines 1219-1223 establish constants relating to the dialog resource ID and items in the dialog. Lines 1224-1228 establish constants relating to the error alert resource and error strings within a 'STR#' resource.

Line 1239 establishes a constant which will be used to inset the view and destination rectangles a few pixels inside the dialog's user item rectangle.

The constants defined at Lines 1230-1231 represent the character codes for the Return and Enter keys.

Lines 1233-1237 establish constants representing the resource ID of the 'TEXT'/'styl' resources associated with each of the pop-up menu items, and the base 'PICT' resource ID for the 'PICT's associated with the first two 'TEXT'/'styl' resources.

#typedef

The data types defined at Lines 1241-1253 are for a picture information record and a document record. Note that one field in the document record is a pointer to a picture information record.

doHelp

doHelp is called when the user chooses the Text1 Help item in the Help menu. (See Lines 781-783 at Text1.c.)

Line 1288 opens the Help dialog. Lines 1294 and 1301 create a document record and assign a handle to that record to the dialog's refCon field.

Lines 1303-1304 save the current graphics port and set the dialog's graphics port as the current port. Lines 1306-1307 install an application-defined drawing function in the dialog's single user item. Lines 1309-1310 assign the handle to the scrollbar to the appropriate field of the dialog's document record.

Lines 1312-1313 make both the destination and view rectangles equal to the user item rectangle, but inset four pixels from the left and right and two pixels from the top and bottom. Line 1314 creates a multistyled edit record based on those two rectangles.

A pointer to a picture information record will eventually be assigned to a field in the document record. For the moment, that field is set to NULL (Line 1316). At Lines 1318-1319, two global variables are assigned the resource IDs relating to the first Help topic's 'TEXT'/'styl' resource and associated 'PICT' resources.

Line 1321 calls an application-defined function which, amongst other things, loads the specified 'TEXT'/'styl' resources and inserts the text and style information into the edit record. Line 1326 calls an application-defined function which, amongst other things, searches for option-space characters in the 'TEXT' resource and, if option-space characters are found, loads a like number of 'PICT' resources beginning with the specified ID.

Line 1332 creates an empty region, referenced by a global variable.

To complete the initial setting up, Line 1334 makes the dialog visible.

The do-while loop entered at Line 1336 continues until the user clicks the "Done" (OK) button or hits the Return key.

The modal dialog uses an application-defined filter function which, as will be seen, handles Return and Enter key events and mouse-down events within the scrollbar. The only other event of interest is a hit on the pop-up menu. Accordingly, if ModalDialog returns a hit on that control (Line 1340), Line 1342 sets the scroll bar value to 0, Lines 1344-1345 determine which menu item the user chose, and Lines 1347-1362 assign the appropriate 'TEXT'/'styl' and 'PICT' resource IDs to the global variables which keep track of which of those resources are to be loaded and displayed. Lines 1364 and 1369 then perform the same actions as did Lines 1321 and 1326 at start-up. Finally, Line 1375 draws any pictures that might initially be located in the view rectangle.

When the user clicks the "Done" (OK) button or hits the Return key, an application-defined function is called to close down the Help dialog.

closeHelp

closeHelp closes down the Help dialog.

Lines 1393-1394 retrieve a handle to the dialog's document record. Line 1396 disposes of the region created at Line 1332. Lines 1399-1400 dispose of the edit record. Lines 1402-1406 dispose of any 'PICT' resources currently in memory, together with the picture information record. Finally, the dialog's document record is disposed of, the dialog itself is disposed of, and the graphics port saved at Line 1303 is restored (Lines 1409-1412).

drawHelp

drawHelp is installed in the dialog's user item and will thus be called whenever the dialog receives an update event. It draws the rectangle in which the text and pictures will be displayed, draws the bold outline around the "Done" (OK) button, and causes the text and pictures to be drawn.

Line 1426 saves the current pen state. Lines 1428-1434 get the dialog's user item rectangle, inset it one pixel all around, draw the resulting rectangle in white, restore the original rectangle, and frame the rectangle in black. Lines 1436-1438 draw the word "Topic:" to the left of the pop-up menu. Lines 1440-1448 draw the bold outline around the "Done" (OK) button.

Lines 1450-1451 get a handle to the edit record and Line 1452 retrieves the view rectangle from the edit record. The call to TEUpdate at Line 1454 draws the text in the edit record in the view rectangle. The call to drawPictures at Line 1455 draws any pictures which might currently be located in the view rectangle.

Line 1457 restores the pen state saved at Line 1426.

getText

getText is called when the dialog is opened and when the user chooses a new item from the pop-up menu. Amongst other things, it loads the 'TEXT'/'styl' resources associated with the current menu item and inserts the text and style information into the edit record.

Lines 1470-1471 get a handle to the edit record.

Lines 1473-1474 set the selection range to the maximum value and then delete that selection. Line 1476 makes the destination rectangle equal to the view rectangle and Line 1477 sets the scroll bar's value to 0.

Lines 1479 and 1486 load the specified 'TEXT'/'styl' resources and Line 1493 copies the text and style information to the edit record. Lines 1495-1496 then release the 'TEXT'/'styl' resources.

Line 1498 copies the number of lines of text in the edit record to a local variable. This is used at Line 1499 to get the total height of the text in pixels.

If the height of the text is greater than the height of the view rectangle (Line 1503), the local variable heightToScroll is made equal to total height of the text minus the height of the view rectangle. This value is then used to set the scroll bar's maximum value (Line 1505). The scroll bar is then made active (Line 1506).

If the height of the text is less than the height of the view rectangle, the scroll bar is made inactive (Lines 1508-1510).

true is returned if the GetResource calls at Lines 1479 and 1486 did not return with false.

getPictureInfo

getPictureInfo is called after getText when the dialog is opened and when the user chooses a new item from the pop-up menu. Amongst other things, it searches for option-space characters in the 'TEXT' resource and, if option-space characters are found, loads a like number of 'PICT' resources beginning with the specified ID.

Line 1530 gets a handle to the dialog's document record. If the picInfoRecPtr field of the document record does not contain NULL (Line 1532), Lines 1533-1538 release the currently loaded 'PICT' resources, dispose of the picture information records, and set the picInfoRecPtr field of the document record to NULL. Line 1541 sets to 0 the field of the document record which keeps track of the number of picures associated with the current 'TEXT' resource.

Lines 1543-1544 get a handle to the edit record, then a handle to the block containing the actual text. This latter is then used at Line 1546 to assign the size of that block to a local variable. After two local variables are initialised at Lines 1547-1548, the block containing the text is locked (Line 1550).

Lines 1552-1558 count the number of option-space characters in the text block. If there are no option-space characters in the block, the block is unlocked and the function returns (Lines 1560-1564).

Line 1566 allocates a nonrelocatable block large enough to accommodate a number of picture information records equal to the number of option-space characters found. The pointer to the block is then assigned to the appropriate field of the dialog's document record (Line 1567).

Line 1569 resets the offset value to 0.

The for loop entered at Line 1571 repeats for each of the option-space characters found. Line 1573 loads the specified 'PICT' resource (the resource ID being incremented from the base ID at each pass through the loop) and assigns the handle to appropriate field of the relevant picture information record. Line 1580 finds the offset to the next option-space character and Line 1581 gets the point, based on the destination rectangle, of the bottom left of the character at that offset. Line 1583 obtains the line height of the character at the offset and this value is subtracted from the value in the point's v field (Line 1584). Line 1586 then assigns the rectangle in the picture record's picFrame field to the bounds field of the picture information record. Lines 1588-1591 then offset this rectangle so that it is centered laterally in the destination rectangle with its top offset from the top of the destination rectangle by the amount established at Line 1584.

Line 1594 assigns the number of pictures loaded to the appropriate field of the dialog's document record and Line 1596 unlocks the block containing the text.

Line 1598 returns true if Line 1577 has not previously returned with false.

handleScrollBar

handleScrollBar is called from helpDialogFilter if a mouse-down event in the scrollbar is detected.

Line 1607 gets a handle to the dialog's document record.

If the mouse-down was in the scroll box TrackControl is called to retain control until the user releases the mouse button (Lines 1609-1611). If TrackControl returns a non-zero value, the application-defined function for scrolling the text and pictures is called (Line 1612).

If the mouse down was in either the scroll arrows or gray area, TrackControl is called to retain control while the mouse button remains down. Note that an action procedure is passed in the third parameter.

actionProcedure

actionProcedure is the action procedure called by handleScrollBar. It is repeatedly called by TrackControl while the mouse button remains down (provided the cursor remains within the scroll arrow or gray area). Its ultimate purpose is to determine the new scrollbar value, move the scroll box to reflect that new value, and call a separate application-defined function to effect the actual scrolling of the text and pictures based on the new scrollbar value.

Firstly, if the cursor is still not within the scroll arrow or gray area (Line 1630) execution falls through to the bottom of the function and the action procedure exits.

Line 1632 gets a pointer to the owner of the scrollbar, allowing Line 1633 to retrieve a handle to the dialog's document record, Line 1634 to get a handle to the edit record, Line 1635 to get the view rectangle, and the h field of a point to be assigned a value equal to the left of the view rectangle plus 4 pixels.)

In the case of the Up scroll arrow (Lines 1640-1646), the variable delta is assigned a value which will ensure that, after the scroll, the top of the incoming line of text will be positioned cleanly at top of the view rectangle.

In the case of the Down scroll arrow (Lines 1648-1653), the variable delta is assigned a value which will ensure that, after the scroll, the bottom of the incoming line of text will be positioned cleanly at bottom of the view rectangle.

In the case of the Up gray area (Lines 1655-1668), the variable delta is assigned a value which will ensure that, after the scroll, the top of the top line of text will be positioned cleanly at the top of the view rectangle and the line of text which was previously at the top will still be visible at the bottom of the view rectangle.

In the case of the Down gray area (Lines 1670-1683), the variable delta is assigned a value which will ensure that, after the scroll, the bottom of the bottom line of text will be positioned cleanly at the bottom of the view rectangle and the line of text which was previously at the bottom will still be visible at the top of the view rectangle.

Line 1686 gets the pre-scroll scrollbar value. If the text is not fully scrolled up and a scroll up is called for, or if the text is not fully scrolled down and a scroll down is called for (Lines 1688-1689), Line 1691 saves the current clipping region, Line 1692 sets the current clipping region to the dialog's port rectangle Line 1694 sets the scroll bar value to the required new value, and Line 1695 restores the saved clipping region. (TextEdit may have set the clipping region to the view rectangle, so it must be changed to include the scroll bar area, otherwise the scroll bar will not be drawn.)

With the scroll bar's new value set and the scroll box redrawn in its new position, the application-defined function for scrolling the text and pictures is called at Line 1698.

scrollTextAndPicts

scrollTextAndPicts is called from actionProcedure (mouse-downs in the scroll arrows or gray area) and from handleScrollBar (mouse-downs in the scroll box). It scrolls the text within the view rectangle and calls another application-defined function to draw any picture whose rectangle intersects the "vacated" area of the view rectangle.

Lines 1711-1712 get a handle to the edit record. Line 1714 determines the difference between the top of the destination rectangle and the top of the view rectangle. Line 1715 subtracts from this value the scroll bar's new value as set at Line 1694. If the result is zero, the text must be fully scrolled in one direction or the other, so the function simply returns (Lines 1716-1717).

If the text is not already fully scrolled one way or the other, Line 1719 scrolls the text in the view rectangle by the number of pixels determined at Line 1715.

If there are no pictures associated with the 'TEXT' resource in use, the function returns immediately after the text is scrolled (Lines 1721-1722).

The next consideration is the pictures and whether any of their rectangles, as stored in the picture information record, intersect the area of the view rectangle "vacated" by the scroll. At Lines 1726-1735, a rectangle is made equal to the "vacated" area of the view rectangle, Lines 1726-1730 catering for the scrolling up case and Lines 1731-1735 catering for the scrolling down case. (Of course, if the scroll box has been dragged by the user over a large distance, the "vacated" area will equate to the whole view rectangle.) This rectangle is passed as a parameter in the call to drawPictures at Line 1742.

drawPictures

drawPictures determines whether any pictures intersect the rectangle passed to it as a formal parameter and draws any pictures that do.

Lines 1750-1751 get handles to the dialog's document record and the edit record. Line 1753 determines the difference between the top of the destination rectangle and the top of the view rectangle. This will be used later to offset the picture's rectangle from destination rectangle coordinates to view rectangle coordinates. Line 1754 determines the number of pictures associated with the current 'TEXT' resource, a value which will be used to control the number of passes through the for loop entered at Line 1756.

Within the loop, the picture's rectangle is retrieved from the picture information record (Line 1758) and offset to the coordinates of the view rectangle (Line 1759). Line 1761 determines whether this rectangle intersects the rectangle passed to drawPictures from scrollTextAndPicts. If it does not, the loop returns for the next iteration. If it does, Line 1764 retrieves the picture's handle, Line 1766 checks whether the 'PICT' resource is in memory and, if necessary, loads it, Line 1767 locks the handle, Line 1771 draws the picture, and Line 1774 unlocks the handle. Before DrawPicture is called, the clipping region is temporarily adjusted to equate to the rectangle passed to drawPictures from scrollTextAndPicts so as to limit drawing to that rectangle.

helpDialogFilter

helpDialogFilter is the application-defined filter function for the dialog.

Update events and key-down events are handled in the normal way. However, this filter function also intercepts mouse-down events (Line 1816). In the case of a mouse-down, Lines 1817-1818 determines the location of the event in local coordinates. If Lines 1820-1821 determine that the mouse-down was within the scrollbar, the application-defined function handleScrollBar is called (Line 1823) and the function reports the item hit, and the fact that it has handled the event, to the Dialog Manager (Lines 1824-1825.

Demonstration Program 2

```
1  // ##############################################################################################
2  // Text2.c
3  // ##############################################################################################
4  //
5  // This program demonstrates:
6  //
7  // •  The use of TextEdit to enter data into the various fields of a database record.
8  //
9  // •  The formatting and display of dates, times and numbers.
10 //
11 // The program opens a window in which four data entry fields are displayed.  A separate
12 // TextEdit edit record is associated with each field.  The first field requires text to
13 // be entered, the second an integer value, the third a floating point value (that is, a
14 // currency value), and the fourth a date.
15 //
16 // The user may select a field for data entry using the mouse or by cycling between
17 // fields using the Tab key.  When the Return key is pressed, some arithmetic is
18 // performed using the integer and floating point values, six months are added to the
19 // date, and the resulting data is formatted and displayed at the bottom of the window.
20 //
21 // The current date and time is displayed at the top of the window.
22 //
23 // The program utilises the following resources:
24 //
```

```
// •  An 'MBAR' resource, and 'MENU' resources for Apple, File, and Edit menus (preload,
//    non-purgeable).
//
// •  A 'WIND' resource (purgeable) (initially visible).
//
// •  A 'SIZE' resource with the acceptSuspendResumeEvents, doesActivateOnFGSwitch, and
//    is32BitCompatible flags set.
//
// ##############################################################################################

// ……… includes

#include <Fonts.h>
#include <Menus.h>
#include <TextEdit.h>
#include <Dialogs.h>
#include <SegLoad.h>
#include <ToolUtils.h>
#include <Devices.h>
#include <LowMem.h>

// ……… defines

#define mApple            128
#define  iAbout           1
#define mFile             129
#define  iQuit            12
#define mEdit             130
#define  iCut             3
#define  iCopy            4
#define  iPaste           5
#define  iClear           6
#define  iSelectAll       7
#define kTab              0x09
#define kDel              0x7F
#define kReturn           0x0D
#define kMaxCharasItem    20
#define kMaxCharasQuant   10
#define kMaxCharasValue   12
#define kMaxCharasDate    13
#define rWindow           128
#define rMenubar          128

#define topLeft(r)        (((Point *) &(r))[0])
#define botRight(r)       (((Point *) &(r))[1])

// ……… typedefs

typedef struct
{
  TEHandle  itemEditHdl;
  TEHandle  quantEditHdl;
  TEHandle  valueEditHdl;
  TEHandle  dateEditHdl;
} DocRec, **DocRecHandle;

// ……… global variables

Boolean   gDone;
Boolean   gInBackground;
RgnHandle gCursorRegion;
TEHandle  gCurrentEditRecHdl;
SInt16    gMaxCharasThisField;
Rect      gItemRect   = { 45,92, 62, 242 };
Rect      gQuantRect  = { 68,92, 85, 242 };
Rect      gValueRect  = { 91,92, 108,242 };
Rect      gDateRect   = { 114,92,131,242 };

// ……… function prototypes

void  main                    (void);
void  doInitManagers          (void);
void  doSetUpEditRecords      (WindowPtr);
void  eventLoop               (void);
void  doIdle                  (void);
void  doEvents                (EventRecord *);
void  doKeyEvent              (SInt8);
```

```
102  void  doHandleTabKey          (void);
103  void  doHandleDelKey          (void);
104  void  doMouseDown             (EventRecord *);
105  void  doInContent             (EventRecord *);
106  void  doUpdate                (EventRecord *);
107  void  doActivate              (EventRecord *);
108  void  doChangeCurrentEditRec  (TEHandle,Boolean,Point,SInt16);
109  void  doTodaysDate            (void);
110  void  doAcceptNewRecord       (void);
111  void  doAcceptItemField       (TEHandle);
112  void  doAcceptQuantField      (TEHandle);
113  void  doAcceptValueField      (TEHandle,TEHandle);
114  void  doAcceptDateField       (TEHandle);
115  void  doAdjustMenus           (void);
116  void  doMenuChoice            (SInt32);
117  void  doEditMenu              (SInt16);
118  void  doAdjustCursor          (WindowPtr,RgnHandle);
119  void  doDrawWindow            (void);
120  void  doEraseRecordDisplay    (void);
121
122  // ############################################################################## main
123
124  void  main(void)
125  {
126    Handle        menubarHdl;
127    MenuHandle    menuHdl;
128    WindowPtr     windowPtr;
129    DocRecHandle  docRecHdl;
130
131    // ……initialise managers
132
133    doInitManagers();
134
135    // …… set up menu bar and menus
136
137    menubarHdl = GetNewMBar(rMenubar);
138    if(menubarHdl == NULL)
139      ExitToShell();
140    SetMenuBar(menubarHdl);
141    DrawMenuBar();
142
143    menuHdl = GetMenuHandle(mApple);
144    if(menuHdl == NULL)
145      ExitToShell();
146    else
147      AppendResMenu(menuHdl,'DRVR');
148
149    // …… open window and attach document record
150
151    if(!(windowPtr = GetNewWindow(rWindow,NULL,(WindowPtr) -1)))
152      ExitToShell();
153
154    if(!(docRecHdl = (DocRecHandle) NewHandle(sizeof(DocRec))))
155      ExitToShell();
156
157    SetWRefCon(windowPtr,(SInt32) docRecHdl);
158    SetPort(windowPtr);
159
160    TextSize(10);
161    TextFont(geneva);
162
163    // …… set up edit records
164
165    doSetUpEditRecords(windowPtr);
166
167    // …… enter eventLoop
168
169    eventLoop();
170  }
171
172  // ########################################################################## doInitManagers
173
174  void  doInitManagers(void)
175  {
176    MaxApplZone();
177    MoreMasters();
178
```

```
  InitGraf(&qd.thePort);
  InitFonts();
  InitWindows();
  InitMenus();
  TEInit();
  InitDialogs(NULL);
  InitCursor();

  FlushEvents(everyEvent,0);
}

// ########################################################################## doSetUpEditRecords

void  doSetUpEditRecords(WindowPtr windowPtr)
{
  Rect          aRect;
  DocRecHandle  docRecHdl;

  docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));

  aRect = gItemRect;
  InsetRect(&aRect,2,2);
  (*docRecHdl)->itemEditHdl = TENew(&aRect,&aRect);

  aRect = gQuantRect;
  InsetRect(&aRect,2,2);
  (*docRecHdl)->quantEditHdl = TENew(&aRect,&aRect);

  aRect = gValueRect;
  InsetRect(&aRect,2,2);
  (*docRecHdl)->valueEditHdl = TENew(&aRect,&aRect);

  aRect = gDateRect;
  InsetRect(&aRect,2,2);
  (*docRecHdl)->dateEditHdl = TENew(&aRect,&aRect);

  gCurrentEditRecHdl = (*docRecHdl)->itemEditHdl;
  gMaxCharasThisField = kMaxCharasItem;
}

// ############################################################################## eventLoop

void  eventLoop(void)
{
  EventRecord eventRec;
  Boolean     gotEvent;
  SInt32      sleepTime;

  gDone = false;
  gCursorRegion = NewRgn();
  sleepTime = LMGetCaretTime();

  while(!gDone)
  {
    gotEvent = WaitNextEvent(everyEvent,&eventRec,sleepTime,gCursorRegion);

    if(!gInBackground)
      doAdjustCursor(FrontWindow(),gCursorRegion);

    if(gotEvent)
      doEvents(&eventRec);
    else
      doIdle();
  }
}

// ################################################################################### doIdle

void  doIdle(void)
{
  static UInt32   oldRawSeconds;
  UInt32          rawSeconds;
  Str255          timeString;
  Rect            eraseRect;

  if(gCurrentEditRecHdl != NULL)
    TEIdle(gCurrentEditRecHdl);
```

```
256
257   GetDateTime(&rawSeconds);
258   if(rawSeconds > oldRawSeconds)
259   {
260     TimeString(rawSeconds,true,timeString,NULL);
261     MoveTo(285,22);
262     SetRect(&eraseRect,285,12,335,22);
263     EraseRect(&eraseRect);
264     DrawString(timeString);
265     oldRawSeconds = rawSeconds;
266   }
267 }
268
269 // ############################################################################## doEvent
270
271 void  doEvents(EventRecord *eventRecPtr)
272 {
273   SInt8   charCode;
274
275   switch(eventRecPtr->what)
276   {
277     case mouseDown:
278       doMouseDown(eventRecPtr);
279       break;
280
281     case keyDown:
282     case autoKey:
283       charCode = eventRecPtr->message & charCodeMask;
284       if((eventRecPtr->modifiers & cmdKey) != 0)
285       {
286         doAdjustMenus();
287         doMenuChoice(MenuKey(charCode));
288       }
289       else
290         doKeyEvent(charCode);
291       break;
292
293     case updateEvt:
294       doUpdate(eventRecPtr);
295       break;
296
297     case activateEvt:
298       doActivate(eventRecPtr);
299       break;
300
301     case osEvt:
302       if(((eventRecPtr->message >> 24) & 0x000000FF) == suspendResumeMessage)
303         gInBackground = (eventRecPtr->message & resumeFlag) == 0;
304       break;
305   }
306 }
307
308 // ############################################################################## doKeyEvent
309
310 void  doKeyEvent(SInt8 charCode)
311 {
312   if(charCode == kTab)
313     doHandleTabKey();
314   else if(charCode == kDel)
315     doHandleDelKey();
316   else if(charCode == kReturn)
317     doAcceptNewRecord();
318   else
319   {
320     if(((*gCurrentEditRecHdl)->teLength < gMaxCharasThisField) || (charCode < 0x20))
321       TEKey(charCode,gCurrentEditRecHdl);
322     else
323       SysBeep(10);
324   }
325 }
326
327 // ############################################################################## doHandleTabKey
328
329 void  doHandleTabKey(void)
330 {
331   WindowPtr     windowPtr;
332   DocRecHandle  docRecHdl;
```

```
  Point         dummy;

  windowPtr = FrontWindow();
  docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));

  if(gCurrentEditRecHdl == (*docRecHdl)->itemEditHdl)
    doChangeCurrentEditRec((*docRecHdl)->quantEditHdl,false,dummy,kMaxCharasQuant);
  else if(gCurrentEditRecHdl == (*docRecHdl)->quantEditHdl)
    doChangeCurrentEditRec((*docRecHdl)->valueEditHdl,false,dummy,kMaxCharasValue);
  else if(gCurrentEditRecHdl == (*docRecHdl)->valueEditHdl)
    doChangeCurrentEditRec((*docRecHdl)->dateEditHdl,false,dummy,kMaxCharasDate);
  else if(gCurrentEditRecHdl == (*docRecHdl)->dateEditHdl)
    doChangeCurrentEditRec((*docRecHdl)->itemEditHdl,false,dummy,kMaxCharasItem);
}

// ############################################################################ doHandleDelKey

void  doHandleDelKey(void)
{
  SInt16  selectionLength;

  selectionLength = (*gCurrentEditRecHdl)->selEnd - (*gCurrentEditRecHdl)->selStart;
  if(selectionLength == 0)
    (*gCurrentEditRecHdl)->selEnd += 1;
  TEDelete(gCurrentEditRecHdl);
}

// ############################################################################ doMouseDown

void  doMouseDown(EventRecord *eventRecPtr)
{
  WindowPtr windowPtr;
  SInt16    partCode;

  partCode = FindWindow(eventRecPtr->where,&windowPtr);

  switch(partCode)
  {
    case inMenuBar:
      doAdjustMenus();
      doMenuChoice(MenuSelect(eventRecPtr->where));
      break;

    case inSysWindow:
      SystemClick(eventRecPtr,windowPtr);
      break;

    case inContent:
      if(windowPtr != FrontWindow())
        SelectWindow(windowPtr);
      else
        doInContent(eventRecPtr);
      break;

    case inDrag:
      DragWindow(windowPtr,eventRecPtr->where,&qd.screenBits.bounds);
      break;

    case inGoAway:
      if(TrackGoAway(windowPtr,eventRecPtr->where))
        gDone = true;
      break;
  }
}

// ############################################################################ doInContent

void  doInContent(EventRecord *eventRecPtr)
{
  WindowPtr     windowPtr;
  DocRecHandle  docRecHdl;
  Point         mouseXY;
  Boolean       shiftKeyPosition;

  windowPtr = FrontWindow();
  docRecHdl = (DocRecHandle) GetWRefCon(windowPtr);

```

```
410     mouseXY = eventRecPtr->where;
411     GlobalToLocal(&mouseXY);
412
413     shiftKeyPosition = (eventRecPtr->modifiers & shiftKey) != 0;
414
415     if(PtInRect(mouseXY,&gItemRect))
416     {
417       if(gCurrentEditRecHdl == (*docRecHdl)->itemEditHdl)
418         TEClick(mouseXY,shiftKeyPosition,(*docRecHdl)->itemEditHdl);
419       else
420         doChangeCurrentEditRec((*docRecHdl)->itemEditHdl,true,mouseXY,kMaxCharasItem);
421     }
422     else if(PtInRect(mouseXY,&gQuantRect))
423     {
424       if(gCurrentEditRecHdl == (*docRecHdl)->quantEditHdl)
425         TEClick(mouseXY,shiftKeyPosition,(*docRecHdl)->quantEditHdl);
426       else
427         doChangeCurrentEditRec((*docRecHdl)->quantEditHdl,true,mouseXY,kMaxCharasQuant);
428     }
429     else if(PtInRect(mouseXY,&gValueRect))
430     {
431       if(gCurrentEditRecHdl == (*docRecHdl)->valueEditHdl)
432         TEClick(mouseXY,shiftKeyPosition,(*docRecHdl)->valueEditHdl);
433       else
434         doChangeCurrentEditRec((*docRecHdl)->valueEditHdl,true,mouseXY,kMaxCharasValue);
435     }
436     else if(PtInRect(mouseXY,&gDateRect))
437     {
438       if(gCurrentEditRecHdl == (*docRecHdl)->dateEditHdl)
439         TEClick(mouseXY,shiftKeyPosition,(*docRecHdl)->dateEditHdl);
440       else
441         doChangeCurrentEditRec((*docRecHdl)->dateEditHdl,true,mouseXY,kMaxCharasDate);
442     }
443   }
444
445   // ############################################################################## doUpdate
446
447   void  doUpdate(EventRecord *eventRecPtr)
448   {
449     WindowPtr     windowPtr;
450     DocRecHandle  docRecHdl;
451     GrafPtr       oldPort;
452
453     windowPtr = (WindowPtr) eventRecPtr->message;
454     docRecHdl = (DocRecHandle) GetWRefCon(windowPtr);
455
456     GetPort(&oldPort);
457     SetPort(windowPtr);
458
459     BeginUpdate((WindowPtr) eventRecPtr->message);
460
461     EraseRect(&windowPtr->portRect);
462     doDrawWindow();
463     TEUpdate(&windowPtr->portRect,(*docRecHdl)->itemEditHdl);
464     TEUpdate(&windowPtr->portRect,(*docRecHdl)->quantEditHdl);
465     TEUpdate(&windowPtr->portRect,(*docRecHdl)->valueEditHdl);
466     TEUpdate(&windowPtr->portRect,(*docRecHdl)->dateEditHdl);
467     EndUpdate((WindowPtr) eventRecPtr->message);
468
469     SetPort(oldPort);
470   }
471
472   // ############################################################################ doActivate
473
474   void  doActivate(EventRecord *eventRecPtr)
475   {
476     WindowPtr windowPtr;
477     Boolean   becomingActive;
478
479     windowPtr = (WindowPtr) eventRecPtr->message;
480
481     becomingActive = ((eventRecPtr->modifiers & activeFlag) == activeFlag);
482
483     if(becomingActive)
484       TEActivate(gCurrentEditRecHdl);
485     else
486       TEDeactivate(gCurrentEditRecHdl);
```

```
}

// ################################################################## doChangeCurrentEditRec

void  doChangeCurrentEditRec(TEHandle newEditRecHdl,Boolean teClickFlag,Point mouseXY,
                             SInt16 maxCharas)
{
  TEDeactivate(gCurrentEditRecHdl);
  TEActivate(newEditRecHdl);
  gCurrentEditRecHdl = newEditRecHdl;

  if(teClickFlag)
    TEClick(mouseXY,false,newEditRecHdl);

  TESetSelect(0,32767,gCurrentEditRecHdl);

  gMaxCharasThisField = maxCharas;
}

// ############################################################################ doTodaysDate

void  doTodaysDate(void)
{
  UInt32  rawSeconds;
  Str255  dateString;

  GetDateTime(&rawSeconds);
  DateString(rawSeconds,longDate,dateString,NULL);
  MoveTo(110,22);
  DrawString(dateString);
}

// ########################################################################## doAcceptNewRecord

void  doAcceptNewRecord(void)
{
  WindowPtr     windowPtr;
  DocRecHandle  docRecHdl;

  windowPtr = FrontWindow();
  docRecHdl = (DocRecHandle) GetWRefCon(windowPtr);

  if((*(*docRecHdl)->itemEditHdl)->teLength  == 0 ||
     (*(*docRecHdl)->quantEditHdl)->teLength == 0 ||
     (*(*docRecHdl)->valueEditHdl)->teLength == 0 ||
     (*(*docRecHdl)->dateEditHdl)->teLength  == 0)
  {
    SysBeep(10);
    return;
  }

  doEraseRecordDisplay();

  doAcceptItemField ((*docRecHdl)->itemEditHdl);
  doAcceptQuantField((*docRecHdl)->quantEditHdl);
  doAcceptValueField((*docRecHdl)->valueEditHdl,(*docRecHdl)->quantEditHdl);
  doAcceptDateField ((*docRecHdl)->dateEditHdl);

  TESetSelect(0,32767,(*docRecHdl)->itemEditHdl);
  TEDelete((*docRecHdl)->itemEditHdl);

  TESetSelect(0,32767,(*docRecHdl)->quantEditHdl);
  TEDelete((*docRecHdl)->quantEditHdl);

  TESetSelect(0,32767,(*docRecHdl)->valueEditHdl);
  TEDelete((*docRecHdl)->valueEditHdl);

  TESetSelect(0,32767,(*docRecHdl)->dateEditHdl);
  TEDelete((*docRecHdl)->dateEditHdl);

  TEDeactivate(gCurrentEditRecHdl);
  gCurrentEditRecHdl = (*docRecHdl)->itemEditHdl;
  TEActivate((*docRecHdl)->itemEditHdl);

  gMaxCharasThisField = kMaxCharasItem;
}
```

```
564  // ########################################################################## doAcceptItemField
565
566  void  doAcceptItemField(TEHandle itemEditHdl)
567  {
568    Str255  itemString;
569
570    GetDialogItemText((*itemEditHdl)->hText,itemString);
571    MoveTo(166,188);
572    DrawString(itemString);
573  }
574
575  // ########################################################################## doAcceptQuantField
576
577  void  doAcceptQuantField(TEHandle quantEditHdl)
578  {
579    Str255  quantString;
580
581    GetDialogItemText((*quantEditHdl)->hText,quantString);
582    MoveTo(166,208);
583    DrawString(quantString);
584  }
585
586  // ########################################################################## doAcceptValueField
587
588  void  doAcceptValueField(TEHandle valueEditHdl,TEHandle quantEditHdl)
589  {
590    Handle            itl4ResourceHdl;
591    SInt32            numpartsOffset;
592    SInt32            numpartsLength;
593    NumberParts       *numpartsTablePtr;
594    Str255            formatString = "\p'$'###,###,###.00;'Valueless';'Valueless'";
595    NumFormatString   formatStringRec;
596    Str255            inputNumString;
597    Str255            formattedNumString;
598    extended80        value80Bit;
599    Str255            quantityString;
600    SInt32            quantity;
601    FormatResultType  result;
602
603    GetIntlResourceTable(smSystemScript,iuNumberPartsTable,&itl4ResourceHdl,&numpartsOffset,
604                         &numpartsLength);
605    numpartsTablePtr = (NumberPartsPtr) ((SInt32) *itl4ResourceHdl + numpartsOffset);
606
607    StringToFormatRec(formatString,numpartsTablePtr,&formatStringRec);
608
609    GetDialogItemText((*valueEditHdl)->hText,inputNumString);
610
611    StringToExtended(inputNumString,&formatStringRec,numpartsTablePtr,&value80Bit);
612    ExtendedToString(&value80Bit,&formatStringRec,numpartsTablePtr,formattedNumString);
613
614    MoveTo(166,228);
615    DrawString(formattedNumString);
616
617    GetDialogItemText((*quantEditHdl)->hText,quantityString);
618    StringToNum(quantityString,&quantity);
619    value80Bit = value80Bit * quantity;
620
621    result = ExtendedToString(&value80Bit,&formatStringRec,numpartsTablePtr,
622                              formattedNumString);
623    MoveTo(166,248);
624    if(result == fFormatOverflow)
625      DrawString("\pToo large to display");
626    else
627      DrawString(formattedNumString);
628  }
629
630  // ########################################################################## doAcceptDateField
631
632  void  doAcceptDateField(TEHandle dateEditHdl)
633  {
634    DateCacheRecord dateCacheRec;
635    Ptr             textPtr;
636    SInt32          textLength, lengthUsed;
637    LongDateRec     longDateTimeRec;
638    LongDateTime    longDateTimeValue;
639    Str255          dateString;
640
```

```
641     InitDateCache(&dateCacheRec);
642     textPtr = (*(*dateEditHdl)->hText);
643     textLength = (*dateEditHdl)->teLength;
644
645     StringToDate(textPtr,textLength,&dateCacheRec,&lengthUsed,&longDateTimeRec);
646
647     LongDateToSeconds(&longDateTimeRec,&longDateTimeValue);
648     longDateTimeValue.lo += 15724800;
649     LongDateString(&longDateTimeValue,longDate,dateString,NULL);
650
651     MoveTo(166,268);
652     DrawString(dateString);
653   }
654
655   // ############################################################################## doAdjustMenus
656
657   void  doAdjustMenus(void)
658   {
659     MenuHandle  fileMenuHdl, editMenuHdl;
660
661     fileMenuHdl = GetMenuHandle(mFile);
662     editMenuHdl = GetMenuHandle(mEdit);
663
664     if((*gCurrentEditRecHdl)->selStart < (*gCurrentEditRecHdl)->selEnd)
665     {
666       EnableItem(editMenuHdl,iCut);
667       EnableItem(editMenuHdl,iCopy);
668       EnableItem(editMenuHdl,iClear);
669     }
670     else
671     {
672       DisableItem(editMenuHdl,iCut);
673       DisableItem(editMenuHdl,iCopy);
674       DisableItem(editMenuHdl,iClear);
675     }
676
677     if(TEGetScrapLength() > 0)
678       EnableItem(editMenuHdl,iPaste);
679     else
680       DisableItem(editMenuHdl,iPaste);
681
682     if((*gCurrentEditRecHdl)->teLength > 0)
683       EnableItem(editMenuHdl,iSelectAll);
684     else
685       DisableItem(editMenuHdl,iSelectAll);
686
687     DrawMenuBar();
688   }
689
690   // ############################################################################## doMenuChoice
691
692   void  doMenuChoice(SInt32 menuChoice)
693   {
694     SInt16  menuID, menuItem;
695     Str255  itemName;
696     SInt16  daDriverRefNum;
697
698     menuID   = HiWord(menuChoice);
699     menuItem = LoWord(menuChoice);
700
701     if(menuID == 0)
702       return;
703
704     switch(menuID)
705     {
706       case mApple:
707         if(menuItem == iAbout)
708           SysBeep(10);
709         else
710         {
711           GetMenuItemText(GetMenuHandle(mApple),menuItem,itemName);
712           daDriverRefNum = OpenDeskAcc(itemName);
713         }
714         break;
715
716       case mFile:
717         if(menuItem == iQuit)
```

```
718           gDone = true;
719         break;
720
721       case mEdit:
722         doEditMenu(menuItem);
723         break;
724     }
725
726     HiliteMenu(0);
727   }
728
729   // ############################################################################## doEditMenu
730
731   void  doEditMenu(SInt16 menuItem)
732   {
733     switch(menuItem)
734     {
735       case iCut:
736         TECut(gCurrentEditRecHdl);
737         break;
738
739       case iCopy:
740         TECopy(gCurrentEditRecHdl);
741         break;
742
743       case iPaste:
744         TEPaste(gCurrentEditRecHdl);
745         break;
746
747       case iClear:
748         TEDelete(gCurrentEditRecHdl);
749         break;
750
751       case iSelectAll:
752         TESetSelect(0,(*gCurrentEditRecHdl)->teLength,gCurrentEditRecHdl);
753         break;
754     }
755   }
756
757   // ############################################################################ doAdjustCursor
758
759   void  doAdjustCursor(WindowPtr windowPtr,RgnHandle mouseRegion)
760   {
761     GrafPtr   oldPort;
762     RgnHandle arrowRegion, itemRegion, quantRegion;
763     RgnHandle valueRegion, dateRegion, iBeamRegions;
764     Rect      itemRect, quantRect, valueRect, dateRect;
765     Point     mouseXY;
766
767     if(gInBackground)
768     {
769       SetCursor(&qd.arrow);
770       return;
771     }
772
773     GetPort(&oldPort);
774     SetPort(windowPtr);
775
776     arrowRegion  = NewRgn();
777     itemRegion   = NewRgn();
778     quantRegion  = NewRgn();
779     valueRegion  = NewRgn();
780     dateRegion   = NewRgn();
781     iBeamRegions = NewRgn();
782
783     SetRectRgn(arrowRegion,-32768,-32768,32766,32766);
784
785     itemRect  = gItemRect;
786     quantRect = gQuantRect;
787     valueRect = gValueRect;
788     dateRect  = gDateRect;
789
790     LocalToGlobal(&topLeft(itemRect));
791     LocalToGlobal(&botRight(itemRect));
792     RectRgn(itemRegion,&itemRect);
793     LocalToGlobal(&topLeft(quantRect));
794     LocalToGlobal(&botRight(quantRect));
```

```
  RectRgn(quantRegion,&quantRect);
  LocalToGlobal(&topLeft(valueRect));
  LocalToGlobal(&botRight(valueRect));
  RectRgn(valueRegion,&valueRect);
  LocalToGlobal(&topLeft(dateRect));
  LocalToGlobal(&botRight(dateRect));
  RectRgn(dateRegion,&dateRect);

  UnionRgn(itemRegion,quantRegion,iBeamRegions);
  UnionRgn(valueRegion,iBeamRegions,iBeamRegions);
  UnionRgn(dateRegion,iBeamRegions,iBeamRegions);

  DiffRgn(arrowRegion,iBeamRegions,arrowRegion);

  GetMouse(&mouseXY);
  LocalToGlobal(&mouseXY);

  if(PtInRgn(mouseXY,iBeamRegions))
  {
    SetCursor(*(GetCursor(iBeamCursor)));
    CopyRgn(iBeamRegions,mouseRegion);
  }
  else
  {
    SetCursor(&qd.arrow);
    CopyRgn(arrowRegion,mouseRegion);
  }

  DisposeRgn(arrowRegion);
  DisposeRgn(itemRegion);
  DisposeRgn(quantRegion);
  DisposeRgn(valueRegion);
  DisposeRgn(dateRegion);
  DisposeRgn(iBeamRegions);

  SetPort(oldPort);
}

// ############################################################################ doDrawWindow

void  doDrawWindow(void)
{
  Rect  lastItemRect;

  doTodaysDate();

  MoveTo(31,34);
  LineTo(331,34);
  MoveTo(31,142);
  LineTo(331,142);

  FrameRect(&gItemRect);
  FrameRect(&gQuantRect);
  FrameRect(&gValueRect);
  FrameRect(&gDateRect);

  SetRect(&lastItemRect,31,171,331,279);
  FrameRect(&lastItemRect);

  MoveTo(31,22);
  DrawString("\pToday's date is:");
  MoveTo(38,57);
  DrawString("\pItem Title:");
  MoveTo(45,80);
  DrawString("\pQuantity:");
  MoveTo(36,103);
  DrawString("\pUnit Value:");
  MoveTo(65,126);
  DrawString("\pDate:");
  MoveTo(248,57);
  MoveTo(118,162);
  DrawString("\pLast Record Entered:");
  MoveTo(248,57);
  DrawString("\pEg: Barometers");
  MoveTo(248,80);
  DrawString("\pEg: 34");
  MoveTo(248,103);
```

```
872     DrawString("\pEg: 135.58");
873     MoveTo(248,126);
874     DrawString("\pEg: 24 Jul 95");
875     MoveTo(110,188);
876     DrawString("\pItem Title:");
877     MoveTo(116,208);
878     DrawString("\pQuantity:");
879     MoveTo(107,228);
880     DrawString("\pUnit Value:");
881     MoveTo(104,248);
882     DrawString("\pTotal Value:");
883     MoveTo(98,268);
884     DrawString("\pReview Date:");
885   }
886
887   // ########################################################################## doEraseRecordDisplay
888
889   void  doEraseRecordDisplay(void)
890   {
891     Rect  rectToErase;
892
893     SetRect(&rectToErase,166,172,330,278);
894     EraseRect(&rectToErase);
895   }
896
897   // ############################################################################################
```

Demonstration Program 2 Comments

When this program is run, the user should enter data in the four displayed data entry fields, using the tab key or mouse clicks to select the required field and pressing the Return key when data has been entered in all fields. If all fields contain data, the Return key press causes a data record, based on the entered data, to be displayed in the bottom of the window.

> The user should note that, although, the program allows only a specific number of characters to be entered in each field, it performs no validity checks on the entered data before accepting it for calculation or display. For example, the program does not prevent the entry of alphabetic characters in the Quantity and Unit Value fields.

In order to observe number formatting effects, the user should occasionally enter very large numbers and negative numbers in the Value field. In order to observe the effects of date string parsing and formatting, the user should enter dates in a variety of formats, for example: "2 Mar 95", "2/3/95", "March 2 1995", "2 3 95", etc.

#define

Lines 48-57 establish constants relating to menu IDs and menu item numbers. Lines 58-60 establish constants for the character codes generated by the tab, del, and return keys. Lines 61-64 establish constants which will be used to limit the number of characters which can be entered in a particular field. Lines 65-66 establish constants relating to resources. Lines 68-69 define two common macros. The first converts the top and left fields of a Rect to a Point. The second converts the bottom and right field of a Rect to a Point.

#typedef

The DocRec data type will be used for a document record, which will be attached to the single window opened by the program. The four fields of this record will be assigned handles to separate TextEdit edit records.

Global Variables

gDone controls program termination. gInBackground relates to foreground/background switching. gCursorRegion relates to the cursorRgn parameter of the WaitNextEvent function. gCurrentEditHdl will be assigned a copy of the handle to the currently activated edit record. gMaxCharasThisField will be assigned a value representing the maximum number of characters allowed to be entered in the currently activated edit record. The global variables at Lines 90-93 will be used in drawing an outline of each data entry field, in establishing the view and destination rectangles for each edit record, and in establishing the I-Beam cursor regions.

main

The main function initialises the system software managers, sets up the menus, opens a window (Line 151), creates a document record and attaches it to the window (Lines 154-157), sets the window's graphics port as the current port, sets the text size and font, calls the application-defined function which sets up the TextEdit edit records (Line 165) and enters the main event loop.

doSetUpEditRecords

doSetUpEditRecords creates four edit records and assigns the handles to those records to the relevant fields of the window's document record.

Line 197 gets a handle to the window's document record.

Line 199-200 establish a view and destination rectangle size two pixels smaller all round than the rectangle which will be drawn in the window. Line 201 creates a monostyled edit record and assigns its handle to the appropriate field of the document record. This general procedure is repeated for the remaining three data entry fields (Lines 205-215).

Line 215 assigns the handle of the first-created edit record to the global variable which will keep track of the currently activated edit record. Line 216 assigns the appropriate value to the global variable which will be used to limit the number of characters that can be entered in the currently activated edit record.

eventLoop

eventLoop is the main event loop. Before the loop is entered, the global variable gDone is set to false (Line 227), a new empty region is created for the variable whose value which will be assigned to the cursorRgn parameter of the WaitNextEvent call (Line 228), and the current caret blink interval is retrieved for use in the sleep parameter of the WaitNextEvent call (Line 229).

The loop is entered at Line 231 and continues until gDone is set to true. Within the loop, the cursor adjustment function is called if the application is not in the background (Lines 235-236), and an idle function is called if the event is a null event (Lines 240-241).

doIdle

doIdle is called when WaitNextEvent returns a null event. doIdle blinks the caret and draws the current time in the top right of the window.

TEIdle is called to ensure that the caret blinks regularly in the specified edit record (Lines 254-255).

Line 257 retrieves the "raw" seconds value, as known to the system. (This is the number of seconds since 1 Jan 1904.) If that value is greater than the value retrieved the last time doIdle was called (Line 258), Line 260 converts the raw seconds value to a string containing the time formatted according to flags in the numeric format ('itl0') resource. (Since NULL is specified in the resource handle parameter, the appropriate 'itl0' resource for the current script system is used.) This string is then drawn in the window (Lines 261-264) and the retrieved raw seconds value is assigned to the static variable oldRawSeconds for use next time doIdle is called.

doEvents

doEvents handles initial event processing. Note that a non-Command key equivalent key-down event is passed to the application-defined function doKeyEvent (Lines 281-291).

doKeyEvent

doKeyEvent handles key-down and auto-key events which are not Command key equivalents.

If the character code is that generated by the tab key or the del key, handling is passed to other application-defined functions (Lines 312-315). If the Return key was pressed, control is passed to an application-defined function which accepts, formats, and displays the entered data and deletes all text from the edit records (Lines 316-317).

If the character code makes it to Line 318, TEKey is called provided that the maximum number of characters allowable in the currently activated edit record will not be exceeded or the character is one of the non-printing characters (Lines 320-321). If the number of characters entered is at the limit and a printing character key was pressed, the character is not accepted and the system alert sound is played (Lines 322-323). This arrangement ensures that,

if the number of characters in the edit record is at the maximum allowed, the arrow and delete keys, unlike the printing character keys, will not be ignored and will have their usual effect.

doHandleTabKey

doHandleTabKey handles the tab key. Its purpose is to cycle around the edit records in response to tab key presses, deactivating the currently activated edit record and activating the next edit record in the sequence.

Lines 335-336 retrieves a handle to the document record for the window.

At Lines 338-339, if the currently activated edit record is that associated with the "Item" data entry field, an application-defined function is called to deactivate that edit record, activate the next edit record in the sequence, and set the global variable which limits the number of characters that may be entered in the new edit record.

Lines 340-345 do the same for the other three edit records.

doHandleDelKey

doHandleDelKey handles the del key, which is not supported by TextEdit. Line 354 gets the current selection length. If the selection length is zero (that is, an insertion point is being displayed), the selection is set to include the character to the right of the insertion point (Lines 355-356). Line 357 deletes the current selection range from the edit record.

doMouseDown

doMouseDown further processes mouse-down events. Note that, at Lines 383-384, a click in the content region of the front window results in a call to the application-defined function doInContent.

doInContent

doInContent handles mouse-down events in the content region of the window.

Lines 407-408 get a handle to the window's document record. Lines 410-411 get the local coordinates of the mouse position at which the mouse-down occurred. (The mouse position in local coordinates will be required by TEClick.)

Line 413 gets the position of shift key at the time of the mouse-down. (TEClick's behaviour depends on the position of the shift key.)

Lines 415-442 respond to the mouse-down provided that it occurred within one of the four data entry rectangles. Lines 415-421 are typical. If the mouse-down occurred within the "Item" rectangle (Line 415), and if the associated edit record is the currently activated edit record (Line 417), TEClick is called (Line 418) to inform TextEdit that a mouse-down has occurred and to retain control until the button is released. If the associated edit record is not the currently activated edit record (Line 419), an application defined function is called (Line 420) to deactivate the currently activated edit record, activate the edit record associated with the rectangle in which the mouse-down occurred, and then call TEClick.

doUpdate

doUpdate handles update events. Between the usual calls to BeginUpdate and EndUpdate, the contents of the window are erased, the application-defined function for drawing the window's contents (less the TextEdit edit record texts) is called, and TEUpdate is called to draw the text of all four edit records (Lines 461-466).

doActivate

doActivate handles activate events. If the window in question is becoming active, the old currently activated edit record is activated, otherwise the currently active edit record is deactivated (Lines 483-486).

doChangeCurrentEditRec

doChangeCurrentEditRec is called by both doHandleTabKey and doInContent to change the currently activated edit record.

Line 494 deactivates the currently activated edit record. Line 495 activates the edit record specified by the calling function and Line 496 copies its handle to the global variable which keeps track of the currently activated edit record.

If this function was called by doInContent (Line 498), TEClick is called (Line 499) to tell TextEdit that the mouse-down has occurred in the newly-activated edit record. Note that the second parameter is set to false, telling TextEdit, regardless of the position of the shift key, that the user is not extending a selection.

Line 501 ensures that, if there are any characters in the edit record, they will be initially highlighted.

doTodaysDate

doTodaysDate draws the date at the top of the window.

Line 513 gets the raw seconds value, as known to the system. Line 514 converts the raw seconds value to a string containing a date formatted in long date format according to flags in the numeric format ('itl0') resource. (Since NULL is specified in the resource handle parameter, the appropriate 'itl0' resource for the current script system is used.) This string is then drawn in the window at Line 516.

doAcceptNewRecord

doAcceptNewRecord is called when the return key is pressed. Assuming each edit record contains at least one character of text, it erases the record display at the bottom of the window, calls other application-defined functions to format (where necessary) and display the fields of the record, and prepares the edit records to accept new data.

Lines 526-527 get a handle to the window's document record. If any one of the edit records does not contain any text, the system alert sound is played and the function returns (Lines 529-536).

Line 538 erases the right hand side of the record display panel. Lines 540-543 call application-defined functions for formatting and displaying the fields of the record. Lines 545-555 delete the text from all edit records. Lines 557-561 deactivate the currently activated edit record, activate the edit record associated with the "Item" data entry field, and assign the maximum characters value applicable to the "Item" data entry field to the global variable which holds the current maximum characters value.

doAcceptItemField

doAcceptItemField is called by doAcceptNewRecord to draw the text in the "Item" edit record in the record panel.

Line 570 gets the text in the "Item" edit record into a Pascal string, which is then drawn in the record panel against "Item:" (Lines 570-572). (GetDialogItemText is usually used to get the text from an editable text item in a dialog box. It works here because the first parameter required by GetDialogItemText is a handle to the hText field of an edit record.)

doAcceptQuantField

doAcceptQuantField is called by doAcceptNewRecord to draw the text in the "Quantity" edit record in the record panel.

Line 581 gets the text in the "Quantity" edit record into a Pascal string, which is then drawn in the record panel against "Quantity:" (Lines 582-583).

doAcceptValueField

doAcceptValueField is called by doAcceptNewRecord to get the string from the "Value" edit record, convert it to floating point number, convert that number to a formatted number string, draw that string, get the string in the "Quantity" edit record, convert that string to an integer, multiply the floating point number by the integer to arrive at the "Total Value" value, convert the result to a formatted number string, and draw that string .

A pointer to a number parts table is required by the functions that convert between floating point numbers and strings. Accordingly, Lines 603-605 get the required pointer.

Line 607 converts the number format specification string at Line 594 into the internal numeric representation required by the functions that convert between floating point numbers and strings.

With that preparation attended to, Line 609 gets the "Value" edit record text into a Pascal string. Line 611 converts that string into a floating point number of type extended (80 bits). Line 612 converts that number back to a string, formatted according to the internal

numeric representation of the number format specification string. Line 615 draws that string against "Value:" in the record panel.

Lines 617-618 get the string from the "Quantity" edit record and convert this string to a long. The extended80 is then multiplied by the long. The 80-bit number is converted to a formatted string at Line 621 and, provided ExtendedToString does not return fFormatOverflow, this string is drawn against "Total Value:" in the record panel (Lines 623-627).

doAcceptDateField

doAcceptDateField is called by doAcceptNewRecord to create a long date-time record from the string in the "Date" edit record, add six months to the date, format the date as a string (long date format), and draw that string in the record panel.

The function which creates the long date-time record takes an initialised date cache record as a parameter. Accordingly, the first step (Line 641) is to initialise the specified date cache record.

Lines 642-643 get a pointer to the text in the "Date" edit record and the length of that text. These are passed as parameters in the StringToDate call at Line 645, which parses the input string and fills in the relevant fields of the long date-time record.

Line 647 converts the long date-time record to a long date-time value. Line 648 adds six months worth of seconds to the long date-time value. The long date-time value is passed as a parameter to LongDateString at Line 649. LongDateString converts the long date-time value to a long format date string formatted according to the specified international resource. (In this case, NULL is passed as the international resource parameter, meaning that the appropriate 'itl1' resource for the current script system is used.)

The formatted date string is drawn against "Review Date:" in the data panel (Line 652).

doAdjustMenus

doAdjustMenus adjusts the menus.

If there is a selection range in the current edit record (Line 664), the Cut, Copy, and Clear items are enabled, otherwise they are disabled. If there is any text in the TextEdit private scrap (Line 677), the Paste item is enabled, otherwise it is disabled. If there is any text in the currently activated edit record (Line 682), Select All is enabled, otherwise it is disabled.

doMenuChoice

doMenuChoice handles Apple and File menu choices to completion and passes Edit menu choices to doEditmenu.

doEditMenu

doEditMenu handles choices from the Edit menu. The appropriate TextEdit procedure is called in each case.

doAdjustCursor

doAdjustCursor is the cursor adjustment function. It is similar to the cursor adjustment functions in previous demonstration programs. However, in this case, four separate I-Beam cursor rectangles are "cut out of" the arrow cursor region.

Line 783 sets the arrow cursor region to the bounds of the coordinate plane. Lines 785-801 then create four regions coinciding with the rectangles used to draw the data field outlines in the window. Lines 803-805 then combine these four regions into one region (iBeamRegions). Line 807, in effect, cuts this four-part region from the arrow region.

doDrawWindow

doDrawWindow draws the contents of the window, less the text in the edit records.

doEraseRecordDisplay

doEraseRecordDisplay erases that part of the record display in which the record's fields are drawn.