17

Version 1.1

TEXT AND TEXTEDIT

Includes Demonstration Programs Text1Pascal and Text2Pascal

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**.

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

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 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	¶	∂	∆	÷	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 `$20` to `$7F` 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.

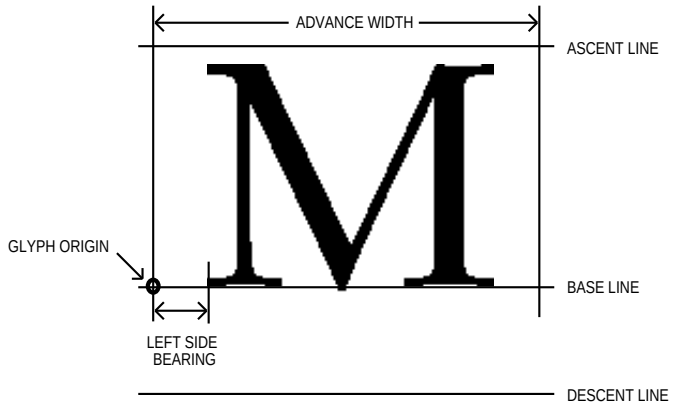

FIG 2 - FONT MEASUREMENT TERMS

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 which 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).

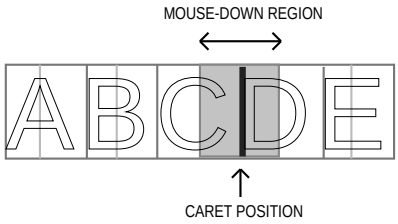

FIG 4 - INTERPRETING CARET POSITION FROM A MOUSE-DOWN EVENT

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.

---

[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**.

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.
- 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:

```
type

TERec = record
  destRect : Rect;         { Destination rectangle.}
  viewRect : Rect;         { View rectangle.}
  selRect : Rect;          { Selection rectangle.}
  lineHeight: integer;     { Used for vertical spacing of lines.}
  fontAscent: integer;     { Used for caret size & highlight rectangle calculation.}
  selPoint: Point;         { Point selected with the mouse.}
  selStart: integer;       { Start of selection range.}
  selEnd : integer;        { End of selection range.}
  active : integer;        { Set when record is activated or deactivated.}
  wordBreak : WordBreakUPP;    { Word break procedure.}
  clickLoop : TEClickLoopUPP;  { Click loop procedure.}
  clickTime : longint;     {(Used internally.)}
  clickLoc : integer;      {(Used internally.)}
  caretTime : longint;     {(Used internally.)}
  caretState : integer;    {(Used internally.)}
  just : integer;          { Text alignment.}
  teLength : integer;      { Length of text.}
  hText : Handle;          { Handle to text to be edited.}
  hDispatchRec : longint; { Handle to TextEdit dispatch record.}
  clikStuff : integer;     {(Used internally.)}
  crOnly : integer;        { If <0, new line at Return only.}
  txFont : integer;        { Text font.}
  txFace : Style;          { Character style.}
  txMode : integer;        { Pen mode.}
  txSize : integer;     { Value indicates font size or, if -1, multistyled edit record.}
```

```
    inPort : GrafPtr;       { Pointer to grafPort for this edit record.}
    highHook : HighHookUPP; { Used for text highlighting, caret appearance.}
    caretHook : CaretHookUPP;    { Used from assembly language.}
    nLines : integer;           { Number of lines.}
    lineStarts : array [0..16000] of integer; { Positions of line starts.}
    end;

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

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]

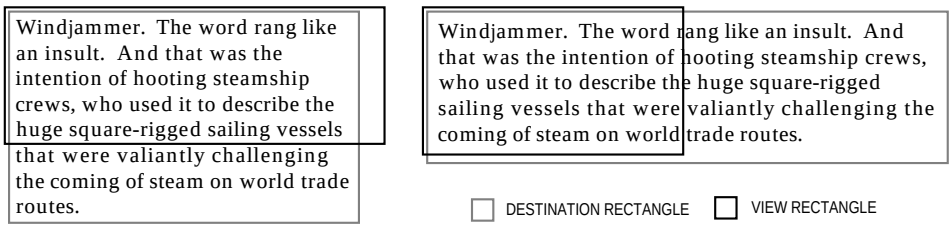

FIG 5 - DESTINATION AND VIEW RECTANGLES

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.

---

[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.

`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 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` 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` 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` and `txFace` 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.

Initializing 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.)

---

[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.

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`.

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.)

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

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

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`.

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 `nil` 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.

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

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 Customizing 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.

Customizing

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 and 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 `txFont` and `txFace` 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.

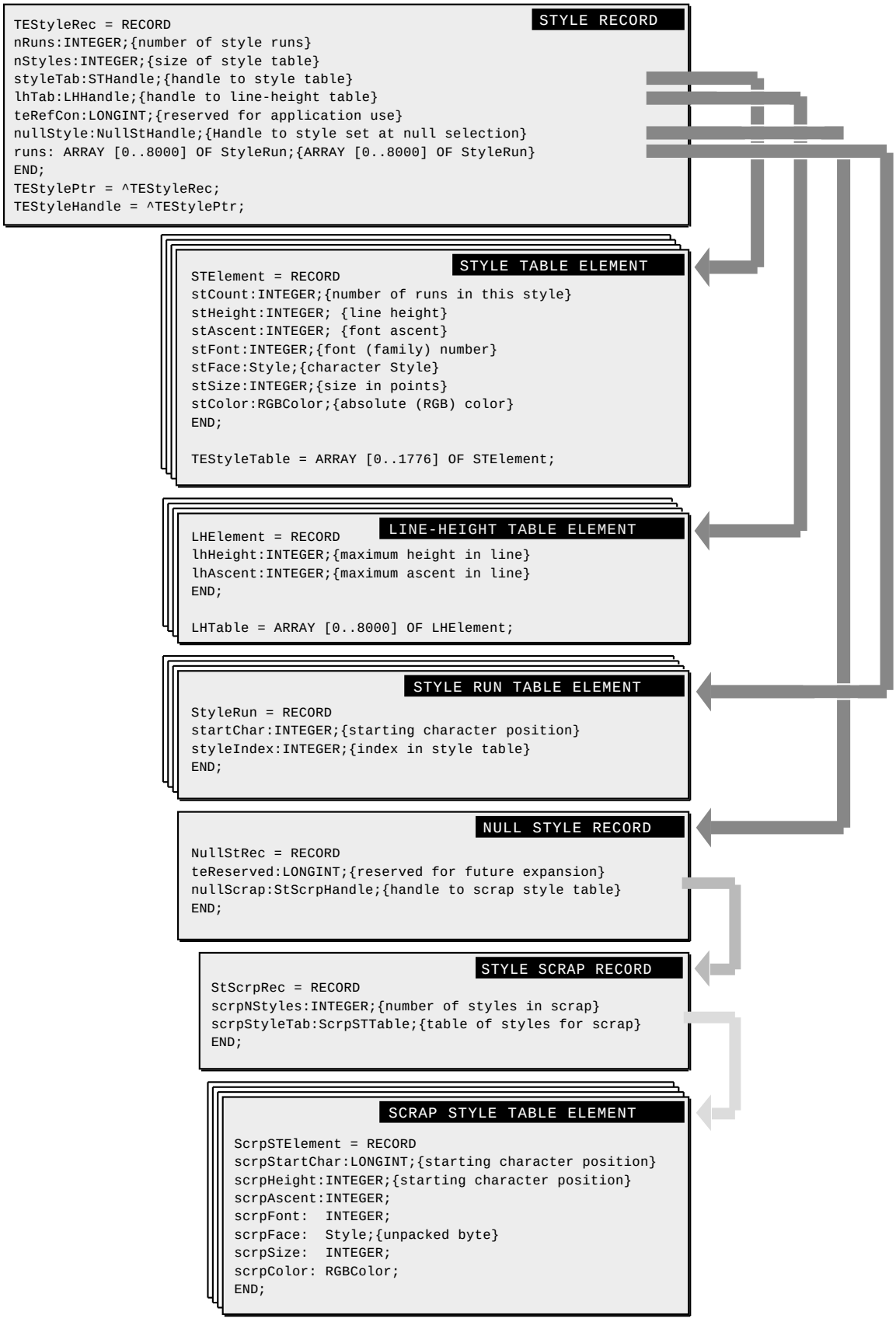

FIG 6 - THE STYLE RECORD AND SUBSIDIARY DATA STRUCTURES

- 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.

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`	Cut text.	For multistyled text:
`TECopy`	Copies 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 `nil` 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.

```
procedure DoGetCurrentSelection(var textStyleRec : TextStyle;
                                editRecHdl : TEHandle);
  var
  mode : integer;
  continuous : boolean;

  { 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 (BAnd(mode, doFont) <> 0) then
    { Font for selection = tsFont field of the TextStyle record. }
```

```
  else
    { More than one font in selection. }

  if (BAnd(mode, doFace) <> 0) then
    { 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(BAnd(mode, doSize) <> 0) then
    { Size for selection = tsSize field of the TextStyle record. }
  else
    { More than one size in selection. }

  If(BAnd(mode, doColor) <> 0) then
    { 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.

```
procedure DoFontMenu(myWindowPtr : WindowPtr; editRecHdl : TEHandle;
                     menuItem : integer);
  var
  styleRec : TextStyle;
  fontName : string;
  fontID : integer;

  begin
  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(myWindowPtr,false);                   { Adjust scroll bars.}
  end;
```

Handling a Size Menu

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

```
procedure DoSizeMenu(myWindowPtr : WindowPtr; editRecHdl : TEHandle;
                     menuItem : integer);
  var
  sizeChosen : integer;
  styleRec : TextStyle;

  begin
  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(myWindowPtr, false);                  { Adjust scroll bars.}
  end;
```

Handling a Style Menu

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

```
procedure DoHandleStyleMenu(myWindowPtr : WindowPtr; editRecHdl : TEHandle;
                            menuItem : integer);
  var
  TextStyle styleRec : integer;

  begin
  case menuItem of
```

```
    iPlain:
      begin
      styleRec.tsFace := ([normal]);
      end;

    iBold:
      begin
      styleRec.tsFace := ([bold]);
      end;

    iItalic:
      styleRec.tsFace := ([italic]);
      end;

    iUnderline:
      begin
      styleRec.tsFace := ([underline]);
      end;

    iOutline:
      begin
      styleRec.tsFace := ([outline]);
      end;

    iShadow:
      begin
      styleRec.tsFace := ([shadow]);
      end;
    end;

  if (menuItem <> 1) then                                  { If Plain not selected }
    TESetStyle(doFace + doToggle, styleRec, true, editRecHdl) { ..include doToggle.}
  else                                                      { If Plain selected }
    TESetStyle(doFace, styleRec, true, editRecHdl);        { ...delete doToggle.}

  DoAdjustScrollBars(myWindowPtr,false);
  end;
```

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.

```
procedure DoAdjustStyleMenu(editRecHdl : TEHandle);

  var
  styleMenuHdl : MenuHandle;
  styleRec : TextStyle;
  mode : integer;

  begin
  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)) then  { There is a continuous }
    begin                                                    { style so mark all }
    CheckItem(styleMenuHdl, iPlain, styleRec.tsFace = []);          { menu items }
    CheckItem(styleMenuHdl, iBold, bold in styleRec.tsFace);      { appropriately.}
    CheckItem(styleMenuHdl, iItalic, italic in styleRec.tsFace);
    { Set other items as appropriate.}
    end
  else begin                                              { There is no continuous }
    CheckItem(styleMenuHdl, iPlain, false);                  { style so unmark all }
    CheckItem(styleMenuHdl, iBold, false);                          { menu items.}
    CheckItem(styleMenuHdl, iItalic, false);
    { Set other items as appropriate.}
    end;
  end;
```

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.

```
Procedure DoSaveAsTextEdit(editRecHdl : TEHandle);

  var
  fileReply : StandardFileFileReply;
  styleScrapHdl : StScrpHandle;
  dataLength : longint;
  dataRefNum : integer;
  rsrcRefNum : integer;
  savedStart : integer;
  savedEnd : integer;
  osError, ignored : OSErr;
  editText : Handle;

  begin
  StandardPutFile('Save as:', 'Untitled', fileReply);
  if (fileReply.sfGood) then begin
    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 not (fileReply.sfReplacing) then                   { Create file & resource }
      begin                                          { fork if not already created.}
      osError := FSpCreate(fileReply.sfFile, 'K JB', 'TEXT', fileReply.sfScript);
      ignored := FSpCreateResFile(fileReply,sfFile,'K JB','TEXT',
                                  fileReply.sfScript);
```

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

```
    osError := ResError;
    end;

  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, '');          { Write attributes.}
  WriteResource(Handle(styleScrapHdl));
  ReleaseResource(Handle(styleScrapHdl));

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

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:

```
procedure DoOpenAsTextEdit(editRecHdl : TEHandle);

  var
  fileReply : StandardFileFileReply;
  fileTypes : SFFileTypes;
  dataRefNum : integer;
  rsrcRefNum : integer;
  textBuffer : Handle;
  textLength : longint;
  styleScrapHdl : StScrpHandle;
  osError : OSErr;
  savedState : UInt8;

  begin
  fileTypes[0] := 'TEXT';

  StandardGetFile(nil, 1, fileTypes, fileReply);
  if (fileReply.sfGood) then
    begin
    osError := FSpOpenDF(fileReply.sfFile, fsCurPerm, dataRefNum);
    osError := SetFPos(dataRefNum, fsFromStart, 0);
    osError := GetEOF(dataRefNum, textLength);

    if (textlength > 32767) then
      textLength := 32767;

    textBuffer := NewHandle(Size(textLength));          { Allocate buffer for text.}
    osError := FSRead(dataRefNum, textLength, textBuffer^);  { Read text to buffer.}
    LockHHi(textBuffer);
    TESetText(textBuffer^, 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;
```

[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.

```
  if (styleScrapHdl <> nil) then
    begin                                                { Apply attributes }
    savedState := HGetState(Handle(styleScrapHdl));         { to edit record}
    TEUseStyleScrap(0,textLength, styleScrapHdl, true, editRecHdl);
    HSetState(Handle(styleScrapHdl), savedState);
    end;

  CloseResFile(rsrcRefNum);                            { Close resource fork.}
  osError := ResError;
  end;
end;
```

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 which return the current date and time in numeric format. Text Utilities routines can then be

used to convert these values into strings that 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:

```
type

  DateTimeRec = record
    year:           integer;
    month:          integer;
    day:            integer;
    hour:           integer;
    minute:         integer;
    second:         integer;
    dayOfWeek:      integer;
  end;

  LongDateRec = record
    case integer of
    0: (
      era:          integer;
      year:         integer;
      month:        integer;
      day:          integer;
      hour:         integer;
      minute:       integer;
      second:       integer;
      dayOfWeek:    integer;
      dayOfYear:    integer;
      weekOfYear:   integer;
      pm:           integer;
      res1:         integer;
      res2:         integer;
      res3:         integer;
       );
    1: (
      list:           array [0..13] of integer;     {Index by LongDateField!}
       );
    2: (
      eraAlt:           integer;
      oldDate:          DateTimeRec;
       );
  end;
```

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 nil 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. Recognises 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 initialization 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.

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
```

[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.

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

```
type

  Chars = packed array [0..32000] of char;
  CharsPtr = ^Chars;
  CharsHandle = ^CharsPtr;
```

Edit Record

```
TERec = record
  destRect:     Rect;           { Destination rectangle.}
  viewRect:     Rect;           { View rectangle.}
  selRect:      Rect;           { Selection rectangle.}
  lineHeight:   integer;        { Used for vertical spacing of lines.}
  fontAscent:   integer;        { Used for caret size and highlight rectangle calculation.}
  selPoint:     Point;          { Point selected with the mouse.}
  selStart:     integer;        { Start of selection range.}
  selEnd:       integer;        { End of selection range.}
  active:       integer;        { Set when record is activated or deactivated.}
  wordBreak:    WordBreakUPP;   { Word break procedure.}
  clickLoop:    TEClickLoopUPP; { Click loop procedure.}
  clickTime:    longint;        {(Used internally.)}
  clickLoc:     integer;        {(Used internally.)}
  caretTime:    longint;        {(Used internally.)}
  caretState:   integer;        {(Used internally.)}
  just:         integer;        { Text alignment.}
  teLength:     integer;        { Length of text.}
  hText:        Handle;         { Handle to text to be edited.}
  hDispatchRec: longint;        { Handle to TextEdit dispatch record.}
  clikStuff:    integer;        {(Used internally.)}
  crOnly:       integer;        { If <0, new line at Return only.}

  txFont:       integer;        { Text font.}
  txFace:       Style;          { Character style.}
  txMode:       integer;        { Pen mode.}
  txSize:       integer;      {Value indicates font size or, if -1, multistyled edit record.}
  inPort:       GrafPtr;        { Pointer to grafPort for this edit record.}
  highHook:     HighHookUPP;    { Used for text highlighting, caret appearance.}
  caretHook:    CaretHookUPP;   { Used from assembly language.}
  nLines:       integer;        { Number of lines.}
  lineStarts:   array [0..16000] of integer; { Positions of line starts.}
  end;

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

Style Record

```
TEStyleRec = record
  nRuns:      integer;          {number of style runs}
  nStyles:    integer;          {size of style table}
  styleTab:   STHandle;         {handle to style table}
  lhTab:      LHHandle;         {handle to line-height table}
```

```
  teRefCon:   longint;          {reserved for application use}
  nullStyle:  NullStHandle;     {Handle to style set at null selection}
  runs:       array [0..8000] of StyleRun;  {array [0..8000] of StyleRun}
end;

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

Style Table Element

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

Line Height Table Element

```
LHElement = record
  lhHeight:         integer;    {maximum height in line}
  lhAscent:       integer;      {maximum ascent in line}
end;

LHTable = array [0..8000] of LHElement;

LHPtr = ^LHTable;
LHHandle = ^LHPtr;
```

Style Run Table Element

```
StyleRun = record
  startChar:        integer;    {starting character position}
  styleIndex:       integer;    {index in style table}
end;
```

Null Style Record

```
NullStRec = record
  teReserved:       longint;        {reserved for future expansion}
  nullScrap:        StScrpHandle;   {handle to scrap style table}
end;

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

Style Scrap Record

```
StScrpRec = record
  scrpNStyles:      integer;        {number of styles in scrap}
  scrpStyleTab:     ScrpSTTable;    {table of styles for scrap}
end;

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

Scrap Style Table Element

```
ScrpSTElement = record
  scrpStartChar:    longint;        {starting character position}
  scrpHeight:       integer;        {starting character position}
  scrpAscent:       integer;
  scrpFont:         integer;
  scrpFace:         Style;          {unpacked byte}
  scrpSize:         integer;
  scrpColor:        RGBColor;
end;

ScrpSTTable = array [0..1600] of ScrpSTElement;
```

Text Style Record

```
TextStyle = record
  tsFont:         integer;          {font (family) number}
  tsFace:         Style;            {character Style}
  tsSize:         integer;          {size in point}
  tsColor:        RGBColor;         {absolute (RGB) color}
end;

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

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 Interfaces.

Initializing TextEdit, Creating Edit Record, Disposing of Edit Record

```
procedure  TEInit;
function   TENew(var destRect: Rect; var viewRect: Rect): TEHandle;
function   TEStyleNew(var destRect: Rect; var viewRect: Rect): TEHandle;
procedure  TEDispose(hTE: TEHandle);
```

Activating and Deactivating an Edit Record

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

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

```
procedure  TEKey(key: CHAR; hTE: TEHandle);
procedure  TESetText(text: UNIV Ptr; length: longint; hTE: TEHandle);
function   TEGetText(hTE: TEHandle): CharsHandle;
procedure  TESetStyleHandle(theHandle: TEStyleHandle; hTE: TEHandle);
function   TEGetStyleHandle(hTE: TEHandle): TEStyleHandle;
```

Setting the Caret and Selection Range

```
procedure  TEIdle(hTE: TEHandle);
procedure  TEClick(pt: Point; fExtend: boolean; h: TEHandle);
procedure  TESetSelect(selStart: longint; selEnd: longint; hTE: TEHandle);
```

Displaying and Scrolling Text

```
procedure  TESetAlignment(just: integer; hTE: TEHandle);
procedure  TEUpdate(var rUpdate: Rect; hTE: TEHandle);
procedure  TETextBox(text: UNIV Ptr; length: longint; var box: Rect; just: integer);
procedure  TEScroll(dh: integer; dv: integer; hTE: TEHandle);
procedure  TESelView(hTE: TEHandle);
procedure  TEPinScroll(dh: integer; dv: integer; hTE: TEHandle);
procedure  TEAutoView(fAuto: boolean; hTE: TEHandle);
procedure  TECalText(hTE: TEHandle);
function   TEGetHeight(endLine: longint; startLine: longint; hTE: TEHandle): longint;
```

Modifying the Text of an Edit Record

```
procedure  TECut(hTE: TEHandle);
procedure  TECopy(hTE: TEHandle);
procedure  TEPaste(hTE: TEHandle);
procedure  TEDelete(hTE: TEHandle);
procedure  TEInsert(text: UNIV Ptr; length: longint; hTE: TEHandle);
procedure  TEStyleInsert(text: UNIV Ptr; length: longint; hST: StScrpHandle; hTE: TEHandle);
procedure  TEStylePaste(hTE: TEHandle);
function   TEFromScrap: OSErr;
function   TEToScrap: OSErr;
```

Managing the TextEdit Private Scrap

```
function   TEScrapHandle : Handle;
function   TEGetScrapLength: longint;
procedure  TESetScrapLength(length: longint);
```

Checking, Setting, and Replacing Styles

```
procedure  TESetStyle(mode: integer; var newStyle: TextStyle; fRedraw: boolean;
           hTE: TEHandle);
procedure  TEReplaceStyle(mode: integer; var oldStyle: TextStyle; var newStyle: TextStyle;
           fRedraw: boolean; hTE: TEHandle);
function   TEContinuousStyle(var mode: integer; var aStyle: TextStyle;
           hTE: TEHandle): boolean;
procedure  TEStyleInsert(text: UNIV Ptr; length: longint; hST: StScrpHandle; hTE: TEHandle);
function   TEGetStyleHandle(hTE: TEHandle): TEStyleHandle;
function   TEGetStyleScrapHandle(hTE: TEHandle): StScrpHandle;
procedure  TEUseStyleScrap(rangeStart: longint; rangeEnd: longint; newStyles: StScrpHandle;
           fRedraw: boolean; hTE: TEHandle);
function   TENumStyles(rangeStart: longint; rangeEnd: longint; hTE: TEHandle): longint;
```

Using Byte Offsets and Corresponding Points

```
function   TEGetOffset(pt: Point; hTE: TEHandle): integer;
function   TEGetPoint(offset: integer; hTE: TEHandle): Point;
```

Customizing TextEdit

```
procedure  TESetClickLoop(clikProc: TEClickLoopUPP; hTE: TEHandle);
procedure  TESetWordBreak(wBrkProc: WordBreakUPP; hTE: TEHandle);
procedure  TECustomHook(which: TEIntHook; var addr: UniversalProcPtr; hTE: TEHandle);
```

Additional TextEdit Features

```
function   TEFeatureFlag(feature: integer; action: integer; hTE: TEHandle): integer;
```

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 Interfaces, 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      = $8000  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           = $8100  Mask for 'tokenizer err encountered'.
cantReadUtilities  = $8200
```

```
dateTimeNotFound    = $8400
dateTimeInvalid     = $8800
```

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

```
StringToDateStatus = integer;
DateForm = SInt8;
FormatStatus = integer;
FormatResultType = SInt8;
```

Data Cache Record

```
DateCacheRecord = packed record
  hidden:         array [0..255] of integer;        { only for temporary use }
end;

DateCachePtr = ^DateCacheRecord;
```

Number Format Specification Record

```
NumFormatString = packed record
  fLength:        UInt8;
  fVersion:       UInt8;
  data:         packed array [0..253] of char;      { private data }
end;

NumFormatStringRec = NumFormatString;
```

Routines

Getting Date-Time Values and Records

```
procedure  GetDateTime(var secs: longint);
procedure  GetTime(var d: DateTimeRec);
```

Converting Between Date-Time values and Records

```
procedure  DateToSeconds(var d: DateTimeRec; var secs: longint);
procedure  SecondsToDate(secs: longint; var d: DateTimeRec);
procedure  LongDateToSeconds(var lDate: LongDateRec; var lSecs: LongDateTime);
procedure  LongSecondsToDate(var lSecs: LongDateTime; var lDate: LongDateRec);
```

Converting Date-Time Strings Into Internal Numeric Representation

```
function   InitDateCache(theCache: DateCachePtr): OSErr;
function   StringToDate(textPtr: Ptr; textLen: longint; theCache: DateCachePtr;
           var lengthUsed: longint; var dateTime: LongDateRec): StringToDateStatus;
function   StringToTime(textPtr: Ptr; textLen: longint; theCache: DateCachePtr;
           var lengthUsed: longint; var dateTime: LongDateRec): StringToDateStatus;
```

Converting Long Date and Time Values Into Strings

```
procedure  DateString(dateTime: longint; longFlag: ByteParameter; var result: Str255;
           intlHandle: Handle);
```

```
procedure  TimeString(dateTime: longint; wantSeconds: boolean; var result: Str255;
           intlHandle: Handle);
procedure  LongDateString(var dateTime: LongDateTime; longFlag: ByteParameter;
           var result: Str255; intlHandle: Handle);
procedure  LongTimeString(var dateTime: LongDateTime; wantSeconds: boolean;
           var result: Str255; intlHandle: Handle);
```

Converting Between Integers and Strings

```
procedure  StringToNum(theString: ConstStr255Param; var theNum: longint);
procedure  NumToString(theNum: longint; var theString: Str255);
```

Using Number Format Specification Strings For International Number Formatting

```
function   StringToFormatRec(inString: ConstStr255Param; var partsTable: NumberParts;
           var outString: NumFormatString): FormatStatus;
function   FormatRecToString(var myCanonical: NumFormatString; var partsTable: NumberParts;
           var outString: Str255; var positions: TripleInt): FormatStatus;
```

Converting Between Strings and Floating Point Numbers

```
function   ExtendedToString(var x: extended80; var myCanonical: NumFormatString;
           var partsTable: NumberParts; var outString: Str255): FormatStatus;
function   StringToExtended(source: ConstStr255Param; var myCanonical: NumFormatString;
           var partsTable: NumberParts; var x: extended80): FormatStatus;
```

Demonstration Program 1

```
{ ############################################################################################
// Text1Pascal.p
// ############################################################################################
//
// 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.
```

```
//
// ########################################################################################## }

program Text1Pascal(input, output);

{ ……… include the following Universal Interfaces }

uses

  Windows, Fonts, Menus, TextEdit, Quickdraw, Dialogs, QuickdrawText, Processes, Types,
  Memory, Events, TextUtils, ToolUtils, OSUtils, Devices, Scrap, SegLoad, StandardFile,
  Controls, Files, LowMem, Balloons,

{ ……… include the following user-defined units }

  UText1Pascal, UHelpDialogPascal;

{ ……… global variables from UText1Pascal }

var

gNumberOfWindows : integer; external;
menubarHdl : Handle; external;
menuHdl : MenuHandle; external;
ignoredWindowPtr : WindowPtr; external;
theErr : OSErr; external;

{ ################################################################ start of main program }

begin

gNumberOfWindows := 0;
ignoredWindowPtr := nil;

  { ……… initialize managers }

  DoInitManagers;

  { ……… set up menu bar and menus }

  menubarHdl := GetNewMBar(rMenubar);
  if (menubarHdl = nil) then
    DoErrorAlert(eMenuBar);
  SetMenuBar(menubarHdl);
  DrawMenuBar;

  menuHdl := GetMenuHandle(mApple);
  if (menuHdl = nil) then
    DoErrorAlert(eMenu)
  else
    AppendResMenu(menuHdl, 'DRVR');

  { ……… set up things for the help menu }

  theErr := HMGetHelpMenuHandle(menuHdl);
  if (theErr = noErr)
    then AppendMenu(menuHdl,'Text1 Help')
    else DoErrorAlert(eMenu);

  { ……… open an untitled window }

  ignoredWindowPtr := DoNewDocWindow;

  { ……… enter EventLoop }

  EventLoop;

end.
  {of main program block}

{ ########################################################################################## }



{ ##########################################################################################
// UText1Pascal.p
// ########################################################################################## }
```

```
123
124 unit UText1Pascal;
125
126 interface
127
128 uses
129
130   Windows;
131
132 { ……… define and export the following constants }
133
134 const
135
136 mApple = 128;
137  iAbout = 1;
138 mFile = 129;
139  iNew = 1;
140  iOpen = 2;
141  iClose = 4;
142  iSaveAs = 6;
143  iQuit = 12;
144 mEdit = 130;
145  iUndo = 1;
146  iCut = 3;
147  iCopy = 4;
148  iPaste = 5;
149  iClear = 6;
150  iSelectAll = 7;
151
152 kMaxTELength = 32767;
153 kTab = $09;
154 kDel = $7F;
155 kReturn = $0D;
156
157 rWindow = 128;
158 rMenubar = 128;
159 rVScrollbar = 128;
160 rErrorAlert = 128;
161 rErrorStrings = 128;
162  eMenuBar = 1;
163  eMenu = 2;
164  eWindow = 3;
165  eDocRecord = 4;
166  eEditRecord = 5;
167  eExceedChara = 6;
168  eNoSpaceCut = 7;
169  eNoSpacePaste = 8;
170
171 { ……… exported global variables }
172
173 var
174
175 gNumberOfWindows : integer;
176 menubarHdl : Handle;
177 menuHdl : MenuHandle;
178 ignoredWindowPtr : WindowPtr;
179 theErr : OSErr;
180
181 { ……… exported functions and procedures }
182
183   procedure DoErrorAlert(errorCode : integer);
184   procedure DoInitManagers;
185   function  DoNewDocWindow : WindowPtr;
186   procedure DoUpdate(var theEvent : EventRecord);
187   procedure EventLoop;
188
189
190 implementation
191
192 { ……… include the following Universal Interfaces }
193
194 uses
195
196   Windows, Fonts, Menus, TextEdit, Quickdraw, Dialogs, QuickdrawText, Processes, Types,
197   Memory, Events, TextUtils, ToolUtils, OSUtils, Devices, Scrap, SegLoad, StandardFile,
198   Controls, Files, LowMem, Balloons,
199
```

```
{ ……… include the following user-defined units }

  UHelpDialogPascal;

{ ……… user-defined types }

type

DocRec = record
  editRecHdl : TEHandle;
  vScrollbarHdl : ControlHandle;
  end;

DocRecPtr = ^DocRec;
DocRecHandle = ^DocRecPtr;

{ ……… global variables }

var

gDone : boolean;
gInBackground : boolean;
gCursorRegion : RgnHandle;

{ ……… function implementations }

{ ############################################################################ DoInitManagers }

procedure DoInitManagers;

  begin
  MaxApplZone;
  MoreMasters;

  InitGraf(@qd.thePort);
  InitFonts;
  InitWindows;
  InitMenus;
  TEInit;
  InitDialogs(nil);

  InitCursor;
  FlushEvents(everyEvent, 0);
  end;
    {of procedure DoInitManagers}

{ ############################################################################ DoErrorAlert }

procedure DoErrorAlert(errorCode : integer);

  var
  errorString : string;
  ignored : integer;

  begin
  GetIndString(errorString, rErrorStrings, errorCode);
  ParamText(errorString, '', '', '');

  if (errorCode < eWindow) then
    begin
    ignored := StopAlert(rErrorAlert, nil);
    ExitToShell;
    end
  else
    ignored := CautionAlert(rErrorAlert, nil);
  end;
    {of procedure DoErrorAlert}

{ ############################################################################ DoHelpMenu }

procedure DoHelpMenu(menuItem : integer);

  var
  helpMenuHdl : MenuHandle;
  origHelpItems, numItems : integer;
  ignored : OSErr;

```

```
  begin
  ignored := HMGetHelpMenuHandle(helpMenuHdl);

  numItems := CountMItems(helpMenuHdl);
  origHelpItems := numItems - 1;

  if (menuItem > origHelpItems) then
    DoHelp;
  end;
    {of procedure DoHelpMenu}

{ ############################################################################ DoDrawDataPanel }

procedure DoDrawDataPanel(myWindowPtr : WindowPtr);

  var
  docRecHdl : DocRecHandle;
  editRecHdl : TEHandle;
  controlHdl : ControlHandle;
  panelRect : Rect;
  textString : string;
  familyID: integer;

  begin
  docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
  editRecHdl := docRecHdl^^.editRecHdl;
  controlHdl := docRecHdl^^.vScrollbarHdl;

  GetFNum('Geneva', familyID);
  TextFont(familyID);
  TextSize(9);

  MoveTo(3,249);
  DrawString('teLength            nLines        lineHeight');

  MoveTo(193,249);
  DrawString('destRect.top            controlValue        contrlMax');

  MoveTo(0,238);
  LineTo(432,238);

  MoveTo(191,239);
  LineTo(191,252);
  MoveTo(283,239);
  LineTo(283,252);
  MoveTo(363,239);
  LineTo(363,252);

  SetRect(panelRect,42,239,76,252);
  EraseRect(panelRect);
  SetRect(panelRect,106,239,128,252);
  EraseRect(panelRect);
  SetRect(panelRect,174,239,190,252);
  EraseRect(panelRect);
  SetRect(panelRect,250,239,282,252);
  EraseRect(panelRect);
  SetRect(panelRect,342,239,361,252);
  EraseRect(panelRect);
  SetRect(panelRect,412,239,433,252);
  EraseRect(panelRect);

  NumToString(SInt32(editRecHdl^^.teLength), textString);
  MoveTo(45, 249);
  DrawString(textString);

  NumToString(SInt32(editRecHdl^^.nLines), textString);
  MoveTo(108, 249);
  DrawString(textString);

  NumToString(SInt32(editRecHdl^^.lineHeight), textString);
  MoveTo(176, 249);
  DrawString(textString);

  NumToString(SInt32(editRecHdl^^.destRect.top), textString);
  MoveTo(251, 249);
  DrawString(textString);

```

```
354     NumToString(SInt32(GetControlValue(controlHdl)), textString);
355     MoveTo(344, 249);
356     DrawString(textString);
357
358     NumToString(SInt32(GetControlMaximum(controlHdl)), textString);
359     MoveTo(414, 249);
360     DrawString(textString);
361
362     TextSize(10);
363     end;
364       {of procedure DoDrawDataPanel}
365
366   { ###################################################################### SetScrollBarValue }
367
368   procedure SetScrollBarValue(controlHdl : ControlHandle; var linesToScroll : integer);
369
370     var
371     controlValue, controlMax : integer;
372
373     begin
374     controlValue := GetControlValue(controlHdl);
375     controlMax := GetControlMaximum(controlHdl);
376
377     linesToScroll := controlValue - linesToScroll;
378     if (linesToScroll < 0) then
379       linesToScroll := 0
380     else if (linesToScroll > controlMax) then
381       linesToScroll := controlMax;
382
383     SetControlValue(controlHdl, linesToScroll);
384     linesToScroll := controlValue - linesToScroll;
385     end;
386       {of procedure SetScrollBarValue}
387
388   { ###################################################################### CustomClikLoop }
389
390   function CustomClikLoop : boolean;
391
392     var
393     myWindowPtr : WindowPtr;
394     docRecHdl : DocRecHandle;
395     editRecHdl : TEHandle;
396     oldPort : GrafPtr;
397     oldClip : RgnHandle;
398     tempRect : Rect;
399     mouseXY : Point;
400     linesToScroll : integer;
401
402     begin
403     myWindowPtr := FrontWindow;
404     docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
405     editRecHdl := docRecHdl^^.editRecHdl;
406
407     GetPort(oldPort);
408     SetPort(myWindowPtr);
409     oldClip := NewRgn;
410     GetClip(oldClip);
411
412     SetRect(tempRect, -32767, -32767, 32767, 32767);
413     ClipRect(tempRect);
414
415     GetMouse(mouseXY);
416
417     if (mouseXY.v < myWindowPtr^.portRect.top) then
418       begin
419       linesToScroll := 1;
420       SetScrollBarValue(docRecHdl^^.vScrollbarHdl, linesToScroll);
421       if (linesToScroll <> 0) then
422         TEScroll(0, linesToScroll * (editRecHdl^^.lineHeight), editRecHdl);
423       end
424
425     else if (mouseXY.v > myWindowPtr^.portRect.bottom) then
426       begin
427       linesToScroll := -1;
428       SetScrollBarValue(docRecHdl^^.vScrollbarHdl, linesToScroll);
429       if (linesToScroll <> 0) then
430         TEScroll(0, linesToScroll * (editRecHdl^^.lineHeight), editRecHdl);
```

```
    end;

  DoDrawDataPanel(myWindowPtr);

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

  CustomClikLoop := true;
  end;
    {of function CustomClikLoop}

{ ############################################################################ DoNewDocWindow }

function DoNewDocWindow : WindowPtr;

  var
  myWindowPtr : WindowPtr;
  docRecHdl : DocRecHandle;
  destAndViewRect : Rect;
  familyID : integer;
  ignored : integer;

  begin
  myWindowPtr := GetNewWindow(rWindow, nil, WindowPtr(-1));
  if (myWindowPtr = nil) then
    begin
    DoErrorAlert(eWindow);
    DoNewDocWindow := nil;
    end;

  SetPort(myWindowPtr);

  TextSize(10);
  GetFNum('Geneva', familyID);
  TextFont(familyID);

  docRecHdl := DocRecHandle(NewHandle(sizeof(DocRec)));
  if (docRecHdl = nil) then
    begin
    DoErrorAlert(eDocRecord);
    DoNewDocWindow := nil;
    end;

  SetWRefCon(myWindowPtr, longint(docRecHdl));

  gNumberOfWindows := gNumberOfWindows + 1;

  docRecHdl^^.vScrollbarHdl := GetNewControl(rVScrollbar, myWindowPtr);

  destAndViewRect := myWindowPtr^.portRect;
  destAndViewRect.right := destAndViewRect.right - 15;
  destAndViewRect.bottom := destAndViewRect.bottom - 15;
  InsetRect(destAndViewRect, 2, 2);

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

  docRecHdl^^.editRecHdl := TENew(destAndViewRect, destAndViewRect);
  if (docRecHdl^^.editRecHdl = nil) then
    begin
    DisposeWindow(myWindowPtr);
    gNumberOfWindows := gNumberOfWindows - 1;
    DisposeHandle(Handle(docRecHdl));
    DoErrorAlert(eEditRecord);
    DoNewDocWindow := nil;
    end;

  HUnlock(Handle(docRecHdl));

  TESetClickLoop(TEClickLoopUPP(@CustomClikLoop), docRecHdl^^.editRecHdl);
  TEAutoView(true, docRecHdl^^.editRecHdl);
  ignored := TEFeatureFlag(teFOutlineHilite, 1, docRecHdl^^.editRecHdl);

  DoNewDocWindow := myWindowPtr;
  end;
    {of function DoNewDocWindow}
```

```
508
509 { ############################################################################## DoOpenFile }
510
511 procedure DoOpenFile(fileSpec : FSSpec);
512
513   var
514   myWindowPtr : WindowPtr;
515   docRecHdl : DocRecHandle;
516   editRecHdl : TEHandle;
517   fileRefNum : integer;
518   textLength : longint;
519   textBufferHdl : Handle;
520   ignored : OSErr;
521
522   begin
523   myWindowPtr := DoNewDocWindow;
524   if (myWindowPtr = nil) then
525     Exit(DoOpenFile);
526
527   docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
528   editRecHdl := docRecHdl^^.editRecHdl;
529
530   SetWTitle(myWindowPtr, fileSpec.name);
531
532   ignored := FSpOpenDF(fileSpec, fsCurPerm, fileRefNum);
533
534   ignored := SetFPos(fileRefNum, fsFromStart, 0);
535   ignored := GetEOF(fileRefNum, textLength);
536
537   if (textLength > 32767) then
538     textLength := 32767;
539
540   textBufferHdl := NewHandle(Size(textLength));
541
542   ignored := FSRead(fileRefNum, textLength, textBufferHdl^);
543
544   MoveHHi(textBufferHdl);
545   HLock(textBufferHdl);
546
547   TESetText(textBufferHdl^, textLength, editRecHdl);
548
549   HUnlock(textBufferHdl);
550   DisposeHandle(textBufferHdl);
551
552   ignored := FSClose(fileRefNum);
553
554   editRecHdl^^.selStart := 0;
555   editRecHdl^^.selEnd := 0;
556
557   ShowWindow(myWindowPtr);
558   end;
559     {of procedure DoOpenFile}
560
561 { ############################################################################ DoOpenCommand }
562
563 procedure DoOpenCommand;
564
565   var
566   fileReply : StandardFileReply;
567   fileTypes : SFTypeList;
568
569   begin
570   fileTypes[0] := 'TEXT';
571
572   StandardGetFile(nil, 1, @fileTypes[0], fileReply);
573   if (fileReply.sfGood) then
574     DoOpenFile(fileReply.sfFile);
575   end;
576     {of procedure DoInitManagers}
577
578 { ############################################################################## DoSaveAsFile }
579
580 procedure DoSaveAsFile(editRecHdl : TEHandle);
581
582   var
583   fileReply : StandardFileReply;
584   myWindowPtr : WindowPtr;
```

```
585     fileRefNum : integer;
586     dataLength : longint;
587     editTextHdl : Handle;
588     ignored : OSErr;
589
590     begin
591     StandardPutFile('Save as:', 'Untitled', fileReply);
592     if (fileReply.sfGood) then
593       begin
594       myWindowPtr := FrontWindow;
595       SetWTitle(myWindowPtr, fileReply.sfFile.name);
596
597       if not(fileReply.sfReplacing) then
598         ignored := FSpCreate(fileReply.sfFile, ' KJB', 'TEXT', fileReply.sfScript);
599
600       ignored := FSpOpenDF(fileReply.sfFile, fsCurPerm, fileRefNum);
601
602       dataLength := editRecHdl^^.teLength;
603       editTextHdl := editRecHdl^^.hText;
604       ignored := FSWrite(fileRefNum, dataLength, editTextHdl^);
605
606       ignored := FSClose(fileRefNum);
607       end;
608     end;
609       {of procedure DoSaveAsFile}
610
611   { ############################################################################## DoCloseWindow }
612
613   procedure DoCloseWindow(myWindowPtr : WindowPtr);
614
615     var
616     docRecHdl : DocRecHandle;
617
618     begin
619     docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
620
621     DisposeControl(docRecHdl^^.vScrollbarHdl);
622     TEDispose(docRecHdl^^.editRecHdl);
623     DisposeHandle(Handle(docRecHdl));
624     DisposeWindow(myWindowPtr);
625
626     gNumberOfWindows := gNumberOfWindows - 1;
627     end;
628       {of procedure DoCloseWindow}
629
630   { ############################################################################# DoAdjustCursor }
631
632   procedure DoAdjustCursor(myWindowPtr : WindowPtr; mouseRegion : RgnHandle);
633
634     var
635     oldPort : GrafPtr;
636     arrowRegion, iBeamRegion : RgnHandle;
637     cursorRect : Rect;
638     mouseXY : Point;
639
640     begin
641     if (gInBackground) then
642       begin
643       SetCursor(qd.arrow);
644       Exit(DoAdjustCursor);
645       end;
646
647     GetPort(oldPort);
648     SetPort(myWindowPtr);
649
650     arrowRegion := NewRgn;
651     iBeamRegion := NewRgn;
652     SetRectRgn(arrowRegion, -32768, -32768, 32766, 32766);
653
654     cursorRect := myWindowPtr^.portRect;
655     cursorRect.bottom := cursorRect.bottom - 15;
656     cursorRect.right := cursorRect.right - 15;
657     LocalToGlobal(cursorRect.topLeft);
658     LocalToGlobal(cursorRect.botRight);
659
660     RectRgn(iBeamRegion, cursorRect);
661     DiffRgn(arrowRegion, iBeamRegion, arrowRegion);
```

```
662
663     GetMouse(mouseXY);
664     LocalToGlobal(mouseXY);
665
666     if (PtInRgn(mouseXY, iBeamRegion)) then
667       begin
668       SetCursor(GetCursor(iBeamCursor)^^);
669       CopyRgn(iBeamRegion, mouseRegion);
670       end
671     else begin
672       SetCursor(qd.arrow);
673       CopyRgn(arrowRegion, mouseRegion);
674       end;
675
676     DisposeRgn(arrowRegion);
677     DisposeRgn(iBeamRegion);
678
679     SetPort(oldPort);
680     end;
681       {of procedure DoAdjustCursor}
682
683   { ########################################################################## DoAdjustScrollbar }
684
685   procedure DoAdjustScrollbar(myWindowPtr : WindowPtr);
686
687     var
688     docRecHdl : DocRecHandle;
689     editRecHdl : TEHandle;
690     numberOfLines, controlMax, controlValue : integer;
691
692     begin
693     docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
694     editRecHdl := docRecHdl^^.editRecHdl;
695
696     numberOfLines := editRecHdl^^.nLines;
697     if (integer(Ptr(longint(@editRecHdl^^.hText^^) +
698         integer(editRecHdl^^.teLength) - 1)^) = kReturn) then
699        numberOfLines := numberOfLines + 1;
700
701     controlMax := numberOfLines - (editRecHdl^^.viewRect.bottom -
702             editRecHdl^^.viewRect.top) div editRecHdl^^.lineHeight;
703
704     if (controlMax < 0) then
705       controlMax := 0;
706     SetControlMaximum(docRecHdl^^.vScrollbarHdl, controlMax);
707
708     controlValue := (editRecHdl^^.viewRect.top - editRecHdl^^.destRect.top) div
709                     editRecHdl^^.lineHeight;
710     if (controlValue < 0) then
711       controlValue := 0
712     else if (controlValue > controlMax) then
713       controlValue := controlMax;
714
715     SetControlValue(docRecHdl^^.vScrollbarHdl, controlValue);
716
717     TEScroll(0, (editRecHdl^^.viewRect.top - editRecHdl^^.destRect.top) -
718                 (GetControlValue(docRecHdl^^.vScrollbarHdl) *
719                 editRecHdl^^.lineHeight), editRecHdl);
720     end;
721       {of procedure DoAdjustScrollbar}
722
723   { ########################################################################## DoGetSelectLength }
724
725   function DoGetSelectLength(editRecHdl : TEHandle) : integer;
726
727     var
728     selectionLength : integer;
729
730     begin
731     selectionLength := editRecHdl^^.selEnd - editRecHdl^^.selStart;
732     DoGetSelectLength := selectionLength;
733     end;
734       {of function DoGetSelectLength}
735
736   { ########################################################################## DoEditMenu }
737
738   procedure DoEditMenu(menuItem : integer);
```

```
739
740   var
741   myWindowPtr : WindowPtr;
742   docRecHdl : DocRecHandle;
743   editRecHdl : TEHandle;
744   totalSize, contigSize, newSize, scrapOffset : longint;
745   selectionLength : integer;
746   ignored : OSErr;
747
748   begin
749   myWindowPtr := FrontWindow;
750   docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
751   editRecHdl := docRecHdl^^.editRecHdl;
752
753   case(menuItem) of
754
755     iUndo: begin
756       end;
757
758     iCut: begin
759       if (ZeroScrap = noErr) then
760         begin
761         PurgeSpace(totalSize, contigSize);
762         selectionLength := DoGetSelectLength(editRecHdl);
763         if (selectionLength > contigSize) then
764           DoErrorAlert(eNoSpaceCut)
765         else begin
766           TECut(editRecHdl);
767           DoAdjustScrollbar(myWindowPtr);
768           if (TEToScrap <> noErr) then
769             ignored := ZeroScrap;
770           end;
771         end;
772       end;
773
774     iCopy: begin
775       if (ZeroScrap = noErr) then
776         begin
777         TECopy(editRecHdl);
778         if (TEToScrap <> noErr) then
779           ignored := ZeroScrap;
780         end;
781       end;
782
783     iPaste: begin
784       newSize := editRecHdl^^.teLength + GetScrap(nil, 'TEXT', scrapOffset);
785       if (newSize > kMaxTELength) then
786         DoErrorAlert(eNoSpacePaste)
787       else begin
788         if (TEFromScrap = noErr) then
789           begin
790           TEPaste(editRecHdl);
791           DoAdjustScrollbar(myWindowPtr);
792           end;
793         end;
794       end;
795
796     iClear: begin
797       TEDelete(editRecHdl);
798       DoAdjustScrollbar(myWindowPtr);
799       end;
800
801     iSelectAll: begin
802       TESetSelect(0, editRecHdl^^.teLength, editRecHdl);
803       end;
804     end;
805       {of case statement}
806
807   DoDrawDataPanel(myWindowPtr);
808   end;
809     {of procedure DoEditMenu}
810
811 { ############################################################################ DoFileMenu }
812
813 procedure DoFileMenu(menuItem : integer);
814
815   var
```

```
816     myWindowPtr : WindowPtr;
817     docRecHdl : DocRecHandle;
818     editRecHdl : TEHandle;
819
820     begin
821     case(menuItem) of
822
823       iNew: begin
824         myWindowPtr := DoNewDocWindow;
825         if (myWindowPtr <> nil) then
826           ShowWindow(myWindowPtr);
827         end;
828
829       iOpen: begin
830         DoOpenCommand;
831         end;
832
833       iClose: begin
834         DoCloseWindow(FrontWindow);
835         end;
836
837       iSaveAs: begin
838         docRecHdl := DocRecHandle(GetWRefCon(FrontWindow()));
839         editRecHdl := docRecHdl^^.editRecHdl;
840         DoSaveAsFile(editRecHdl);
841         end;
842
843       iQuit: begin
844         gDone := true;
845         end;
846       end;
847         {of case statement}
848     end;
849       {of procedure DoFileMenu}
850
851   { ##################################################################### DoActivateDocWindow }
852
853   procedure DoActivateDocWindow(myWindowPtr : WindowPtr; becomingActive : Boolean);
854
855     var
856     docRecHdl : DocRecHandle;
857     editRecHdl : TEHandle;
858
859     begin
860     docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
861     editRecHdl := docRecHdl^^.editRecHdl;
862
863     if (becomingActive) then
864       begin
865       SetPort(myWindowPtr);
866
867       editRecHdl^^.viewRect.bottom := (((editRecHdl^^.viewRect.bottom -
868           editRecHdl^^.viewRect.top) div editRecHdl^^.lineHeight) *
869           editRecHdl^^.lineHeight) + editRecHdl^^.viewRect.top;
870
871       editRecHdl^^.destRect.bottom := editRecHdl^^.viewRect.bottom;
872
873       TEActivate(editRecHdl);
874       HiliteControl(docRecHdl^^.vScrollbarHdl, 0);
875       DoAdjustScrollbar(myWindowPtr);
876       end
877
878     else begin
879       TEDeactivate(editRecHdl);
880       HiliteControl(docRecHdl^^.vScrollbarHdl, 255);
881       end;
882     end;
883       {of procedure DoActivateDocWindow}
884
885   { ############################################################################ DoMenuChoice }
886
887   procedure DoMenuChoice(menuChoice : longint);
888
889     var
890     menuID, menuItem : integer;
891     itemName : string;
892     daDriverRefNum : integer;
```

```
893
894     begin
895     menuID := HiWord(menuChoice);
896     menuItem := LoWord(menuChoice);
897
898     if (menuID = 0) then
899       Exit(DoMenuChoice);
900
901     case(menuID) of
902       mApple:
903         begin
904         if (menuItem = iAbout) then
905           SysBeep(10)
906         else begin
907           GetMenuItemText(GetMenuHandle(mApple), menuItem, itemName);
908           daDriverRefNum := OpenDeskAcc(itemName);
909           end;
910         end;
911
912       mFile:
913         begin
914         DoFileMenu(menuItem);
915         end;
916
917       mEdit:
918         begin
919         DoEditMenu(menuItem);
920         end;
921
922       kHMHelpMenuID:
923         begin
924         if (FrontWindow <> nil) then
925           DoActivateDocWindow(FrontWindow,false);
926         DoHelpMenu(menuItem);
927         end;
928       end;
929         {of case statement}
930
931     HiliteMenu(0);
932     end;
933       {of procedure DoMenuChoice}
934
935   { ############################################################################ DoAdjustMenus }
936
937   procedure DoAdjustMenus;
938
939     var
940     fileMenuHdl, editMenuHdl : MenuHandle;
941     myWindowPtr : WindowPtr;
942     docRecHdl : DocRecHandle;
943     editRecHdl : TEHandle;
944     scrapOffset : longint;
945
946     begin
947     fileMenuHdl := GetMenuHandle(mFile);
948     editMenuHdl := GetMenuHandle(mEdit);
949
950     if (gNumberOfWindows > 0) then
951       begin
952       myWindowPtr := FrontWindow;
953       docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
954       editRecHdl := docRecHdl^^.editRecHdl;
955
956       EnableItem(fileMenuHdl, iClose);
957
958       if (editRecHdl^^.selStart < editRecHdl^^.selEnd) then
959         begin
960         EnableItem(editMenuHdl, iCut);
961         EnableItem(editMenuHdl, iCopy);
962         EnableItem(editMenuHdl, iClear);
963         end
964       else begin
965         DisableItem(editMenuHdl, iCut);
966         DisableItem(editMenuHdl, iCopy);
967         DisableItem(editMenuHdl, iClear);
968         end;
969
```

```
      if (GetScrap(nil, 'TEXT', scrapOffset) > 0) then
        EnableItem(editMenuHdl, iPaste)
      else
        DisableItem(editMenuHdl, iPaste);

      if (editRecHdl^^.teLength > 0) then
        begin
        EnableItem(fileMenuHdl, iSaveAs);
        EnableItem(editMenuHdl, iSelectAll);
        end
      else begin
        DisableItem(fileMenuHdl, iSaveAs);
        DisableItem(editMenuHdl, iSelectAll);
        end;
      end
    else begin
      DisableItem(fileMenuHdl, iClose);
      DisableItem(fileMenuHdl, iSaveAs);
      DisableItem(editMenuHdl, iClear);
      DisableItem(editMenuHdl, iSelectAll);
      end;

    DrawMenuBar;
    end;
      {of procedure DoAdjustMenus}

  { ############################################################################## DoOpSysEvent }

  procedure DoOpSysEvent(var theEvent : EventRecord);

    begin
    if (BAnd(BSR(theEvent.message, 24), $000000FF) = suspendResumeMessage) then
      begin
      if  (BAnd(theEvent.message, resumeFlag) = 0)
        then gInBackground := true
        else gInBackground := false;

      if (gNumberOfWindows > 0) then
        DoActivateDocWindow(FrontWindow, not gInBackground);
      HiliteMenu(0);
      end;
    end;
      {of procedure DoOpSysEvent}

  { ############################################################################## DoActivate }

  procedure DoActivate(var theEvent : EventRecord);

    var
    myWindowPtr : WindowPtr;
    becomingActive : boolean;

    begin
    myWindowPtr := WindowPtr(theEvent.message);
    if (BAnd(theEvent.modifiers, activeFlag) = activeFlag) then
      becomingActive := true
    else
      becomingActive := false;
    DoActivateDocWindow(myWindowPtr, becomingActive);
    end;
      {of procedure DoActivate}

  { ########################################################################## ScrollActionProc }

  procedure ScrollActionProc(controlHdl : ControlHandle; partCode : integer);

    var
    myWindowPtr : WindowPtr;
    docRecHdl : DocRecHandle;
    editRecHdl : TEHandle;
    linesToScroll : integer;
    controlValue, controlMax : integer;

    begin
    if (partCode <> 0) then
      begin
      myWindowPtr := controlHdl^^.contrlOwner;
```

```
1047      docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
1048      editRecHdl := docRecHdl^^.editRecHdl;
1049
1050      case (partCode) of
1051
1052        kControlUpButtonPart, kControlDownButtonPart:
1053          begin
1054          linesToScroll := 1;
1055          end;
1056
1057        kControlPageUpPart, kControlPageDownPart:
1058          begin
1059          linesToScroll := ((editRecHdl^^.viewRect.bottom -
1060                editRecHdl^^.viewRect.top) div editRecHdl^^.lineHeight) - 1;
1061          end;
1062        end;
1063          {of case statement}
1064
1065      if ((partCode = kControlDownButtonPart) or (partCode = kControlPageDownPart)) then
1066        linesToScroll := -linesToScroll;
1067
1068      controlValue := GetControlValue(controlHdl);
1069      controlMax := GetControlMaximum(controlHdl);
1070
1071      linesToScroll := controlValue - linesToScroll;
1072      if (linesToScroll < 0) then
1073        linesToScroll := 0
1074      else if (linesToScroll > controlMax) then
1075        linesToScroll := controlMax;
1076
1077      SetControlValue(controlHdl, linesToScroll);
1078
1079      linesToScroll := controlValue - linesToScroll;
1080
1081      if (linesToScroll <> 0) then
1082        TEScroll(0, linesToScroll * editRecHdl^^.lineHeight, editRecHdl);
1083
1084      DoDrawDataPanel(myWindowPtr);
1085      end;
1086
1087    end;
1088      {of procedure ScrollActionProc}
1089
1090  { ############################################################################## DoInContent }
1091
1092  procedure DoInContent(var theEvent : EventRecord);
1093
1094    var
1095    myWindowPtr : WindowPtr;
1096    docRecHdl : DocRecHandle;
1097    editRecHdl : TEHandle;
1098    mouseXY : Point;
1099    controlHdl : ControlHandle;
1100    partCode, controlValue : integer;
1101    shiftKeyPosition : boolean;
1102    ignored : OSErr;
1103
1104    begin
1105    shiftKeyPosition := false;
1106    myWindowPtr := FrontWindow;
1107    docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
1108    editRecHdl := docRecHdl^^.editRecHdl;
1109
1110    mouseXY := theEvent.where;
1111    GlobalToLocal(mouseXY);
1112
1113    partCode := FindControl(mouseXY, myWindowPtr, controlHdl);
1114    if (partCode <> 0) then
1115      begin
1116      case (partCode) of
1117
1118        kControlUpButtonPart,kControlDownButtonPart,kControlPageUpPart,kControlPageDownPart:
1119          begin
1120          ignored := TrackControl(controlHdl, mouseXY,
1121                                  ControlActionUPP(@ScrollActionProc));
1122          end;
1123
```

```
1124        kControlIndicatorPart:
1125          begin
1126          controlValue := GetControlValue(controlHdl);
1127          partCode := TrackControl(controlHdl, mouseXY, nil);
1128          if (partCode <> 0) then
1129            begin
1130            controlValue := controlValue - GetControlValue(controlHdl);
1131            if (controlValue <> 0) then
1132              TEScroll(0, controlValue * editRecHdl^^.lineHeight, editRecHdl);
1133            end;
1134          DoDrawDataPanel(myWindowPtr);
1135          end;
1136        end;
1137          {of case statement}
1138      end
1139
1140    else if (PtInRect(mouseXY, editRecHdl^^.viewRect)) then
1141      begin
1142      if (BAnd(theEvent.modifiers, shiftKey) <> 0) then
1143        shiftKeyPosition := true;
1144      TEClick(mouseXY, shiftKeyPosition, editRecHdl);
1145      end;
1146
1147    end;
1148      {of procedure DoInContent}
1149
1150  { ############################################################################## DoMouseDown }
1151
1152  procedure DoMouseDown(var theEvent : EventRecord);
1153
1154    var
1155    myWindowPtr : WindowPtr;
1156    partCode : integer;
1157
1158    begin
1159    partCode := FindWindow(theEvent.where, myWindowPtr);
1160
1161    case (partCode) of
1162
1163      inMenuBar:
1164        begin
1165        DoAdjustMenus;
1166        DoMenuChoice(MenuSelect(theEvent.where));
1167        end;
1168
1169      inSysWindow:
1170        begin
1171        SystemClick(theEvent, myWindowPtr);
1172        end;
1173
1174      inContent:
1175        begin
1176        if (myWindowPtr <> FrontWindow) then
1177          SelectWindow(myWindowPtr)
1178        else
1179          DoInContent(theEvent);
1180        end;
1181
1182      inDrag:
1183        begin
1184        DragWindow(myWindowPtr, theEvent.where, qd.screenBits.bounds);
1185        end;
1186
1187      inGoAway:
1188        begin
1189        if (TrackGoAway(myWindowPtr, theEvent.where)) then
1190          DoCloseWindow(FrontWindow);
1191        end;
1192      end;
1193        {of case statement}
1194    end;
1195      {of procedure DoMouseDown}
1196
1197  { ############################################################################## DoKeyEvent }
1198
1199  procedure DoKeyEvent(charCode : char);
1200
```

```
1201    var
1202    myWindowPtr : WindowPtr;
1203    docRecHdl : DocRecHandle;
1204    editRecHdl : TEHandle;
1205    selectionLength : integer;
1206
1207    begin
1208    myWindowPtr := FrontWindow;
1209    docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
1210    editRecHdl := docRecHdl^^.editRecHdl;
1211
1212    if (charCode = char(kTab)) then
1213      begin
1214      { Do tab key handling here if required. }
1215      end
1216    else if (charCode = char(kDel)) then
1217      begin
1218      selectionLength := DoGetSelectLength(editRecHdl);
1219      if (selectionLength = 0) then
1220        editRecHdl^^.selEnd := editRecHdl^^.selEnd + 1;
1221      TEDelete(editRecHdl);
1222      DoAdjustScrollbar(myWindowPtr);
1223      end
1224    else begin
1225      selectionLength := DoGetSelectLength(editRecHdl);
1226      if ((editRecHdl^^.teLength - selectionLength + 1) < kMaxTELength) then
1227        begin
1228        TEKey(charCode, editRecHdl);
1229        DoAdjustScrollbar(myWindowPtr);
1230        end
1231      else
1232        DoErrorAlert(eExceedChara);
1233      end;
1234
1235    DoDrawDataPanel(myWindowPtr);
1236    end;
1237      {of procedure DoKeyEvent}
1238
1239  { ############################################################################## DoIdle }
1240
1241  procedure DoIdle;
1242
1243    var
1244    docRecHdl : DocRecHandle;
1245    myWindowPtr : WindowPtr;
1246
1247    begin
1248    myWindowPtr := FrontWindow();
1249
1250    docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
1251    if (docRecHdl <> nil) then
1252      TEIdle(docRecHdl^^.editRecHdl);
1253    end;
1254      {of procedure DoIdle}
1255
1256  { ############################################################################ DoUpdate }
1257
1258  procedure DoUpdate(var theEvent : EventRecord);
1259
1260    var
1261    myWindowPtr : WindowPtr;
1262    docRecHdl : DocRecHandle;
1263    editRecHdl : TEHandle;
1264    oldPort : GrafPtr;
1265
1266    begin
1267    myWindowPtr := WindowPtr(theEvent.message);
1268    docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
1269    editRecHdl := docRecHdl^^.editRecHdl;
1270
1271    GetPort(oldPort);
1272    SetPort(myWindowPtr);
1273
1274    BeginUpdate(WindowPtr(theEvent.message));
1275
1276    EraseRect(myWindowPtr^.portRect);
1277    TEUpdate(myWindowPtr^.portRect, editRecHdl);
```

```
  UpdateControls(myWindowPtr, myWindowPtr^.visRgn);

  DoDrawDataPanel(myWindowPtr);

  EndUpdate(WindowPtr(theEvent.message));

  SetPort(oldPort);
  end;
    {of procedure DoUpdate}

{ ############################################################################## DoEvents }

procedure DoEvents(var theEvent : EventRecord);

  var
  charCode : char;

  begin
  case (theEvent.what) of

    mouseDown:
      begin
      DoMouseDown(theEvent);
      end;

    keyDown, autoKey:
      begin
      charCode := chr(BAnd(theEvent.message, charCodeMask));
      if (BAnd(theEvent.modifiers, cmdKey) <> 0) then
        begin
        DoAdjustMenus;
        DoMenuChoice(MenuKey(charCode));
        end
      else
        DoKeyEvent(charCode);
      end;

    updateEvt:
      begin
      DoUpdate(theEvent);
      end;

    activateEvt:
      begin
      DoActivate(theEvent);
      end;

    osEvt:
      begin
      DoOpSysEvent(theEvent);
      end;

    end;
      {of case statement}

  end;
    {of procedure DoEvents}

{ ############################################################################## EventLoop }

procedure EventLoop;

  var
  theEvent : EventRecord;
  gotEvent : boolean;
  sleepTime : longint;

  begin
  gDone := false;
  gCursorRegion := NewRgn;
  sleepTime := LMGetCaretTime;

  while not (gDone) do
    begin
    gotEvent := WaitNextEvent(everyEvent, theEvent, sleepTime, gCursorRegion);

    if ((not gInBackground) and (gNumberOfWindows > 0)) then
```

```
      DoAdjustCursor(FrontWindow, gCursorRegion);

    if (gotEvent) then
      DoEvents(theEvent)
    else begin
      if (gNumberOfWindows > 0) then
        DoIdle;
      end;
    end;
  end;
    {of procedure EventLoop}

end.
  {of unit UText1Pascal}

{ ############################################################################################ }



{ ############################################################################################
// UHelpDialogPascal.p
// ############################################################################################ }

unit UHelpDialogPascal;



interface

{ ……… exported procedures }

  procedure DoHelp;



implementation

{ ……… include the following Universal Interfaces }

uses

  Dialogs, ToolUtils, Resources, SegLoad,

{ ……… include the following user-defined units }

  UText1Pascal;

{ ……… define the following constants }

const

rErrorStrings = 128;
 eWindow = 3;


rHelpModal = 129;
 iOK = 1;
 iTextUserItem = 2;
 iScrollBar = 3;
 iPopupMenu = 4;
rErrorAlert = 128;
 eHelpDialog = 9;
 eHelpDocRecord = 10;
 eHelpText = 11;
 eHelpPicture = 12;
kTextInset = 4;
kReturn = $0D;
kEnter = $03;

rTextIntroduction = 128;
rTextCreatingText = 129;
rTextModifyHelp = 130;
rPictIntroductionBase = 128;
rPictCreatingTextBase = 129;

{ ……… user-defined types }

```

```
1432  type
1433
1434  PictInfoRec = record
1435    bounds : Rect;
1436    pictureHdl : PicHandle;
1437    end;
1438  PictInfoRecPointer = ^PictInfoRec;
1439  PictInfoRecArray = array [0..0] of PictInfoRec;
1440  PictInfoRecArrayPtr = ^PictInfoRecArray;
1441
1442  DocRecord = record
1443    editRecHdl : TEHandle;
1444    scrollbarHdl : ControlHandle;
1445    pictCount : integer;
1446    pictInfoRecPtr : PictInfoRecArrayPtr;
1447    end;
1448  DocRecordPtr = ^DocRecord;
1449  DocRecordHandle = ^DocRecordPtr;
1450
1451  { ..... global variables }
1452
1453  var
1454
1455  gTextResourceID : integer;
1456  gPictResourceBaseID : integer;
1457  gSavedClipRgn : RgnHandle;
1458
1459  { ..... function definitions }
1460
1461
1462    procedure DoHelp; forward;
1463    procedure CloseHelp( modalDlgPtr : DialogPtr; oldPort : GrafPtr); forward;
1464    procedure DrawHelp(modalDlgPtr : DialogPtr; theItem : integer); forward;
1465    function GetText(modalDlgPtr : DialogPtr; textResourceID : integer;
1466          viewRect : Rect) : boolean; forward;
1467    function GetPictureInfo(modalDlgPtr : DialogPtr;
1468          firstPictID : integer) : boolean; forward;
1469    procedure HandleScrollBar(modalDlgPtr : DialogPtr; thePart : integer;
1470          mouseXY : Point); forward;
1471    procedure ActionProcedure(scrollbarHdl : ControlHandle; partCode : integer); forward;
1472    procedure ScrollTextAndPicts(modalDlgPtr : DialogPtr); forward;
1473    procedure DrawPictures(modalDlgPtr : DialogPtr; var updateRect : Rect); forward;
1474    function HelpDialogFilter(modalDlgPtr : DialogPtr; var theEvent : EventRecord;
1475          var itemHit : integer) : boolean; forward;
1476
1477
1478  { ..... function implementations }
1479
1480  { ############################################################################## DoHelp }
1481
1482  procedure DoHelp;
1483
1484    var
1485    modalDlgPtr : DialogPtr;
1486    docRecHdl : DocRecordHandle;
1487    oldPort : GrafPtr;
1488    itemType, itemHit, menuItem : integer;
1489    itemHdl : Handle;
1490    userItemRect, destRect, viewRect, itemRect : Rect;
1491
1492    begin
1493    gSavedClipRgn := nil;
1494
1495    modalDlgPtr := GetNewDialog(rHelpModal, nil, WindowPtr(-1));
1496    if (modalDlgPtr = nil) then
1497      begin
1498      DoErrorAlert(eHelpDialog);
1499      Exit(DoHelp);
1500      end;
1501
1502    docRecHdl := DocRecordHandle(NewHandle(sizeof(DocRecord)));
1503    if (docRecHdl = nil) then
1504      begin
1505      DoErrorAlert(eHelpDocRecord);
1506      DisposeDialog(modalDlgPtr);
1507      Exit(DoHelp);
1508      end;
```

```
1509
1510    SetWRefCon(modalDlgPtr, longint(docRecHdl));
1511
1512    GetPort(oldPort);
1513    SetPort(modalDlgPtr);
1514
1515    GetDialogItem(modalDlgPtr, iTextUserItem, itemType, itemHdl, userItemRect);
1516    SetDialogItem(modalDlgPtr, iTextUserItem, itemType, Handle(@DrawHelp), userItemRect);
1517
1518    GetDialogItem(modalDlgPtr, iScrollBar, itemType, itemHdl, itemRect);
1519    docRecHdl^^.scrollbarHdl := ControlHandle(itemHdl);
1520
1521    InsetRect(userItemRect, kTextInset, kTextInset div 2);
1522    destRect := userItemRect;
1523    viewRect := userItemRect;
1524    docRecHdl^^.editRecHdl := TEStyleNew(destRect, viewRect);
1525
1526    docRecHdl^^.pictInfoRecPtr := nil;
1527
1528    gTextResourceID := rTextIntroduction;
1529    gPictResourceBaseID := rPictIntroductionBase;
1530
1531    if not (GetText(modalDlgPtr, gTextResourceID, viewRect)) then
1532      begin
1533      CloseHelp(modalDlgPtr, oldPort);
1534      Exit(DoHelp);
1535      end;
1536
1537    if not(GetPictureInfo(modalDlgPtr, gPictResourceBaseID)) then
1538      begin
1539      CloseHelp(modalDlgPtr, oldPort);
1540      Exit(DoHelp);
1541      end;
1542
1543    gSavedClipRgn := NewRgn;
1544
1545    ShowWindow(modalDlgPtr);
1546
1547    repeat
1548    ModalDialog(ModalFilterUPP(@HelpDialogFilter), itemHit);
1549
1550    if (itemHit = iPopupMenu) then
1551      begin
1552      SetControlValue(docRecHdl^^.scrollbarHdl, 0);
1553
1554      GetDialogItem(modalDlgPtr, iPopupMenu, itemType, itemHdl, itemRect);
1555      menuItem := GetControlValue(ControlHandle(itemHdl));
1556
1557      case (menuItem) of
1558
1559        1: begin
1560          gTextResourceID := rTextIntroduction;
1561          gPictResourceBaseID := rPictIntroductionBase;
1562          end;
1563
1564        2: begin
1565          gTextResourceID := rTextCreatingText;
1566          gPictResourceBaseID := rPictCreatingTextBase;
1567          end;
1568
1569        3: begin
1570          gTextResourceID := rTextModifyHelp;
1571          end;
1572        end;
1573          {of case statement}
1574
1575      if not (GetText(modalDlgPtr, gTextResourceID, viewRect)) then
1576        begin
1577        CloseHelp(modalDlgPtr, oldPort);
1578        Exit(DoHelp);
1579        end;
1580
1581      if not (GetPictureInfo(modalDlgPtr, gPictResourceBaseID)) then
1582        begin
1583        CloseHelp(modalDlgPtr, oldPort);
1584        Exit(DoHelp);
1585        end;
```

```
1586
1587      DrawPictures(modalDlgPtr, viewRect);
1588      end;
1589
1590    until (itemHit = iOK);
1591
1592    CloseHelp(modalDlgPtr, oldPort);
1593
1594    Exit(DoHelp);
1595    end;
1596      {of procedure DoHelp}
1597
1598  { ############################################################################## CloseHelp }
1599
1600  procedure CloseHelp(modalDlgPtr : DialogPtr; oldPort : GrafPtr);
1601
1602    var
1603    docRecHdl : DocRecordHandle;
1604    editRecHdl : TEHandle;
1605    a : integer;
1606
1607    begin
1608    docRecHdl := DocRecordHandle(GetWRefCon(modalDlgPtr));
1609    editRecHdl := docRecHdl^^.editRecHdl;
1610
1611    if (gSavedClipRgn <> nil) then
1612      DisposeRgn(gSavedClipRgn);
1613
1614    if (docRecHdl^^.editRecHdl <> nil) then
1615      TEDispose(docRecHdl^^.editRecHdl);
1616
1617    if (docRecHdl^^.pictInfoRecPtr <> nil) then
1618      begin
1619      for a := 0 to (docRecHdl^^.pictCount - 1) do
1620        ReleaseResource(Handle(docRecHdl^^.pictInfoRecPtr^[a].pictureHdl));
1621      DisposePtr(Ptr(docRecHdl^^.pictInfoRecPtr));
1622      end;
1623    DisposeHandle(Handle(docRecHdl));
1624    DisposeDialog(modalDlgPtr);
1625
1626    SetPort(oldPort);
1627    end;
1628      {of procedure CloseHelp}
1629
1630  { ############################################################################## DrawHelp }
1631
1632  procedure DrawHelp(modalDlgPtr : DialogPtr; theItem : integer);
1633
1634    var
1635    oldPenState : PenState;
1636    itemHdl : Handle;
1637    itemRect, viewRect : Rect;
1638    itemType, buttonOval : integer;
1639    docRecHdl : DocRecordHandle;
1640    editRecHdl : TEHandle;
1641
1642    begin
1643    GetPenState(oldPenState);
1644
1645    GetDialogItem(modalDlgPtr, iTextUserItem, itemType, itemHdl, itemRect);
1646    InsetRect(itemRect, 1, 1);
1647
1648    BackColor(whiteColor);
1649    FillRect(itemRect, qd.white);
1650    InsetRect(itemRect, -1, -1);
1651    FrameRect(itemRect);
1652
1653    TextFont(0);
1654    MoveTo(13, 309);
1655    DrawString('Topic:');
1656
1657    GetDialogItem(modalDlgPtr, iOK, itemType, itemHdl, itemRect);
1658    InsetRect(itemRect, -4, -4);
1659
1660    buttonOval := (itemRect.bottom - itemRect.top) div 2 + 2;
1661
1662    ForeColor(blackColor);
```

```
1663    PenPat(qd.black);
1664    PenSize(3, 3);
1665    FrameRoundRect(itemRect, buttonOval, buttonOval);
1666
1667    docRecHdl := DocRecordHandle(GetWRefCon(modalDlgPtr));
1668    editRecHdl := docRecHdl^^.editRecHdl;
1669    viewRect := editRecHdl^^.viewRect;
1670
1671    TEUpdate(viewRect, editRecHdl);
1672    DrawPictures(modalDlgPtr, viewRect);
1673
1674    SetPenState(oldPenState);
1675    end;
1676      {of procedure DrawHelp}
1677
1678  { ############################################################################## GetText }
1679
1680  function GetText(modalDlgPtr : DialogPtr; textResourceID : integer; viewRect : Rect)
1681                                                  : boolean;
1682
1683    var
1684    docRecHdl : DocRecordHandle;
1685    editRecHdl : TEHandle;
1686    helpTextHdl : Handle;
1687    stylScrpRecHdl : StScrpHandle;
1688    numberOfLines, heightOfText, heightToScroll : integer;
1689
1690    begin
1691    docRecHdl := DocRecordHandle(GetWRefCon(modalDlgPtr));
1692    editRecHdl := docRecHdl^^.editRecHdl;
1693
1694    TESetSelect(0, 32767, editRecHdl);
1695    TEDelete(editRecHdl);
1696
1697    editRecHdl^^.destRect := editRecHdl^^.viewRect;
1698    SetControlValue(docRecHdl^^.scrollbarHdl, 0);
1699
1700    helpTextHdl := GetResource('TEXT', textResourceID);
1701    if (helpTextHdl = nil) then
1702      begin
1703      DoErrorAlert(eHelpText);
1704      GetText := false;
1705      end;
1706
1707    stylScrpRecHdl := StScrpHandle(GetResource('styl', textResourceID));
1708    if (stylScrpRecHdl = nil) then
1709      begin
1710      DoErrorAlert(eHelpText);
1711      GetText := false;
1712      end;
1713
1714    TEStyleInsert(helpTextHdl^, GetHandleSize(helpTextHdl), stylScrpRecHdl, editRecHdl);
1715
1716    ReleaseResource(helpTextHdl);
1717    ReleaseResource(Handle(stylScrpRecHdl));
1718
1719    numberOfLines := editRecHdl^^.nLines;
1720    heightOfText := TEGetHeight(longint(numberOfLines), 1, editRecHdl);
1721
1722    if (heightOfText >  (viewRect.bottom - viewRect.top)) then
1723      begin
1724      heightToScroll := TEGetHeight(longint(numberOfLines), 1, editRecHdl) -
1725                          (viewRect.bottom - viewRect.top);
1726      SetControlMaximum(docRecHdl^^.scrollbarHdl, heightToScroll);
1727      HiliteControl(docRecHdl^^.scrollbarHdl, 0);
1728      end
1729    else begin
1730      HiliteControl(docRecHdl^^.scrollbarHdl, 255);
1731      end;
1732
1733    GetText := true;
1734    end;
1735      {of function GetText}
1736
1737  { ########################################################################## GetPictureInfo }
1738
1739  function GetPictureInfo(modalDlgPtr : DialogPtr; firstPictID : integer) : boolean;
```

```
1740
1741    var
1742    docRecHdl : DocRecordHandle;
1743    editRecHdl : TEHandle;
1744    textHdl : Handle;
1745    offset, textSize : longint;
1746    numberOfPicts, a, lineHeight, fontAscent : integer;
1747    optionSpace : array [1..1] of SInt8;
1748    pictInfoPtr : PictInfoRecArrayPtr;
1749    picturePoint : Point;
1750    whatStyle : TextStyle;
1751
1752    begin
1753    optionSpace[1] := SInt8($CA);
1754    docRecHdl := DocRecordHandle(GetWRefCon(modalDlgPtr));
1755
1756    if (docRecHdl^^.pictInfoRecPtr <> nil) then
1757      begin
1758      for a := 0 to (docRecHdl^^.pictCount - 1) do
1759        ReleaseResource(Handle(docRecHdl^^.pictInfoRecPtr^[a].pictureHdl));
1760
1761      DisposePtr(Ptr(docRecHdl^^.pictInfoRecPtr));
1762      docRecHdl^^.pictInfoRecPtr := nil;
1763      end;
1764
1765    docRecHdl^^.pictCount := 0;
1766
1767    editRecHdl := docRecHdl^^.editRecHdl;
1768    textHdl := editRecHdl^^.hText;
1769
1770    textSize := GetHandleSize(textHdl);
1771    offset := 0;
1772    numberOfPicts := 0;
1773
1774    HLock(textHdl);
1775
1776    offset := Munger(textHdl, offset, @optionSpace[1], 1, nil, 0);
1777    while ((offset >= 0) and (offset <= textSize)) do
1778      begin
1779      numberOfPicts := numberOfPicts + 1;
1780      offset := offset + 1;
1781      offset := Munger(textHdl, offset, @optionSpace[1], 1, nil, 0);
1782      end;
1783
1784    if (numberOfPicts = 0) then
1785      begin
1786      HUnlock(textHdl);
1787      GetPictureInfo :=  true;
1788      end;
1789
1790    pictInfoPtr := PictInfoRecArrayPtr(NewPtr(sizeof(PictInfoRec) * numberOfPicts));
1791    docRecHdl^^.pictInfoRecPtr := pictInfoPtr;
1792
1793    offset := longint(0);
1794
1795    for a := 0 to (numberOfPicts - 1) do
1796      begin
1797      pictInfoPtr^[a].pictureHdl := GetPicture(firstPictID + a);
1798      if (pictInfoPtr^[a].pictureHdl = nil) then
1799        begin
1800        DoErrorAlert(eHelpPicture);
1801        GetPictureInfo :=  false;
1802        end;
1803
1804      offset := Munger(textHdl, offset, @optionSpace[1], 1, nil, 0);
1805      picturePoint := TEGetPoint(integer(offset), editRecHdl);
1806
1807      TEGetStyle(offset, whatStyle, lineHeight, fontAscent, editRecHdl);
1808      picturePoint.v := picturePoint.v - lineHeight;
1809      offset := offset + 1;
1810      pictInfoPtr^[a].bounds := pictInfoPtr^[a].pictureHdl^^.picFrame;
1811
1812      OffsetRect(pictInfoPtr^[a].bounds,
1813              ((editRecHdl^^.destRect.right + editRecHdl^^.destRect.left) -
1814              (pictInfoPtr^[a].bounds.right + pictInfoPtr^[a].bounds.left)) div 2,
1815              - pictInfoPtr^[a].bounds.top + picturePoint.v);
1816      end;
```

```
1817    docRecHdl^^.pictCount := a;
1818
1819    HUnlock(textHdl);
1820
1821    GetPictureInfo := true;
1822    end;
1823      {of function GetPictureInfo}
1824
1825  { ############################################################################ HandleScrollBar }
1826
1827  procedure HandleScrollBar(modalDlgPtr : DialogPtr; thePart : integer; mouseXY : Point);
1828
1829    var
1830    docRecHdl : DocRecordHandle;
1831    ignored : OSErr;
1832
1833    begin
1834    docRecHdl := DocRecordHandle(GetWRefCon(modalDlgPtr));
1835
1836    if (thePart = kControlIndicatorPart) then
1837      begin
1838      if (TrackControl(docRecHdl^^.scrollbarHdl, mouseXY, nil) = noErr) then
1839        ScrollTextAndPicts(modalDlgPtr);
1840      end
1841    else
1842      ignored := TrackControl(docRecHdl^^.scrollbarHdl, mouseXY,
1843            ControlActionUPP(@ActionProcedure));
1844    end;
1845      {of function GetPictureInfo}
1846
1847  { ############################################################################ ActionProcedure }
1848
1849  procedure ActionProcedure(scrollbarHdl : ControlHandle; partCode : integer);
1850
1851    var
1852    docRecHdl : DocRecordHandle;
1853    modalDlgPtr : DialogPtr;
1854    editRecHdl : TEHandle;
1855    delta, oldValue, offset, lineHeight, fontAscent : integer;
1856    thePoint : Point;
1857    viewRect : Rect;
1858    style : TextStyle;
1859
1860    begin
1861    if (partCode <> 0) then
1862      begin
1863      modalDlgPtr := scrollbarHdl^^.contrlOwner;
1864      docRecHdl := DocRecordHandle(GetWRefCon(modalDlgPtr));
1865      editRecHdl := docRecHdl^^.editRecHdl;
1866      viewRect := editRecHdl^^.viewRect;
1867      thePoint.h := viewRect.left + kTextInset;
1868
1869      case (partCode) of
1870
1871        kControlUpButtonPart: begin
1872          thePoint.v := viewRect.top - 4;
1873          offset := TEGetOffset(thePoint, editRecHdl);
1874          thePoint := TEGetPoint(offset, editRecHdl);
1875          TEGetStyle(offset, style, lineHeight, fontAscent, editRecHdl);
1876          delta := thePoint.v - lineHeight - viewRect.top;
1877          end;
1878
1879        kControlDownButtonPart: begin
1880          thePoint.v := viewRect.bottom + 2;
1881          offset := TEGetOffset(thePoint, editRecHdl);
1882          thePoint := TEGetPoint(offset, editRecHdl);
1883          delta := thePoint.v - viewRect.bottom;
1884          end;
1885
1886        kControlPageUpPart: begin
1887          thePoint.v := viewRect.top + 2;
1888          offset := TEGetOffset(thePoint, editRecHdl);
1889          thePoint := TEGetPoint(offset, editRecHdl);
1890          TEGetStyle(offset, style, lineHeight, fontAscent, editRecHdl);
1891          thePoint.v := thePoint.v + lineHeight - fontAscent;
1892          thePoint.v := thePoint.v - viewRect.bottom - viewRect.top;
1893          offset := TEGetOffset(thePoint, editRecHdl);
```

```
1894          thePoint := TEGetPoint(offset, editRecHdl);
1895          TEGetStyle(offset, style, lineHeight, fontAscent, editRecHdl);
1896          delta := thePoint.v - viewRect.top;
1897          if (offset = 0) then
1898            delta := delta - lineHeight;
1899          end;
1900
1901        kControlPageDownPart: begin
1902          thePoint.v := viewRect.bottom - 2;
1903          offset := TEGetOffset(thePoint, editRecHdl);
1904          thePoint := TEGetPoint(offset, editRecHdl);
1905          TEGetStyle(offset, style, lineHeight, fontAscent, editRecHdl);
1906          thePoint.v := thePoint.v - fontAscent;
1907          thePoint.v := thePoint.v + viewRect.bottom - viewRect.top;
1908          offset := TEGetOffset(thePoint, editRecHdl);
1909          thePoint := TEGetPoint(offset, editRecHdl);
1910          TEGetStyle(offset, style, lineHeight, fontAscent, editRecHdl);
1911          delta :=  thePoint.v - lineHeight - viewRect.bottom;
1912          if (offset = editRecHdl^^.teLength) then
1913            delta := delta + lineHeight;
1914          end;
1915        end;
1916          {of case statement}
1917
1918      oldValue := GetCtlValue(scrollbarHdl);
1919
1920      if (((delta < 0) and (oldValue > 0)) or
1921            ((delta > 0) and (oldValue < GetControlMaximum(scrollbarHdl)))) then
1922        begin
1923        GetClip(gSavedClipRgn);
1924        ClipRect(modalDlgPtr^.portRect);
1925
1926        SetControlValue(scrollbarHdl, oldValue + delta);
1927        SetClip(gSavedClipRgn);
1928        end;
1929
1930      ScrollTextAndPicts(modalDlgPtr);
1931      end;
1932    end;
1933      {of procedure ActionProcedure}
1934
1935  { ##################################################################### ScrollTextAndPicts }
1936
1937  procedure ScrollTextAndPicts(modalDlgPtr : DialogPtr);
1938
1939    var
1940    docRecHdl : DocRecordHandle;
1941    editRecHdl : TEHandle;
1942    scrollDistance, oldScroll : integer;
1943    updateRect : Rect;
1944
1945    begin
1946    docRecHdl := DocRecordHandle(GetWRefCon(modalDlgPtr));
1947    editRecHdl := docRecHdl^^.editRecHdl;
1948
1949    oldScroll := editRecHdl^^.viewRect.top - editRecHdl^^.destRect.top;
1950    scrollDistance := oldScroll - GetCtlValue(docRecHdl^^.scrollbarHdl);
1951    if (scrollDistance = 0) then
1952      Exit(ScrollTextAndPicts);
1953
1954    TEScroll(0, scrollDistance, editRecHdl);
1955
1956    if (docRecHdl^^.pictCount = 0) then
1957      Exit(ScrollTextAndPicts);
1958
1959    updateRect := editRecHdl^^.viewRect;
1960
1961    if (scrollDistance > 0) then
1962      begin
1963      if (scrollDistance < (updateRect.bottom - updateRect.top)) then
1964        updateRect.bottom := updateRect.top + scrollDistance;
1965      end
1966    else begin
1967      if ( - scrollDistance < (updateRect.bottom - updateRect.top)) then
1968        updateRect.top := updateRect.bottom + scrollDistance;
1969      end;
1970
```

```
1971    DrawPictures(modalDlgPtr, updateRect);
1972    end;
1973      {of procedure ScrollTextAndPicts}
1974
1975  { ############################################################################ DrawPictures }
1976
1977  procedure DrawPictures(modalDlgPtr : DialogPtr; var updateRect : Rect);
1978
1979    var
1980    docRecHdl : DocRecordHandle;
1981    editRecHdl : TEHandle;
1982    pictCount, pictIndex, vOffset : integer;
1983    thePictHdl : PicHandle;
1984    pictLocRect, dummyRect : Rect;
1985
1986    begin
1987    docRecHdl := DocRecordHandle(GetWRefCon(modalDlgPtr));
1988    editRecHdl := docRecHdl^^.editRecHdl;
1989
1990    vOffset := editRecHdl^^.destRect.top - editRecHdl^^.viewRect.top - kTextInset;
1991    pictCount := docRecHdl^^.pictCount;
1992
1993    for pictIndex := 0 to (pictCount - 1) do
1994      begin
1995      pictLocRect := docRecHdl^^.pictInfoRecPtr^[pictIndex].bounds;
1996        OffsetRect(pictLocRect, 0, vOffset);
1997
1998      if (SectRect(pictLocRect, updateRect, dummyRect)) then
1999        begin
2000
2001        thePictHdl := docRecHdl^^.pictInfoRecPtr^[pictIndex].pictureHdl;
2002
2003        LoadResource(Handle(thePictHdl));
2004        HLock(Handle(thePictHdl));
2005
2006        GetClip(gSavedClipRgn);
2007        ClipRect(updateRect);
2008        DrawPicture(thePictHdl, pictLocRect);
2009
2010        SetClip(gSavedClipRgn);
2011        HUnlock(Handle(thePictHdl));
2012        end;
2013      end;
2014
2015    end;
2016      {of procedure DrawPictures}
2017
2018  { ######################################################################## HelpDialogFilter }
2019
2020  function HelpDialogFilter(modalDlgPtr : DialogPtr; var theEvent : EventRecord;
2021                        var itemHit : integer) : boolean;
2022
2023    var
2024    docRecHdl : DocRecordHandle;
2025    handledEvent : boolean;
2026    itemType, thePart : integer;
2027    charCode : char;
2028    itemRect : Rect;
2029    itemHdl : Handle;
2030    mouseXY : Point;
2031    finalTicks : longint;
2032    controlHdl : ControlHandle;
2033
2034    begin
2035    handledEvent := false;
2036
2037    if ((theEvent.what = updateEvt) and (WindowPtr(theEvent.message) <> modalDlgPtr)) then
2038      begin
2039      DoUpdate(theEvent);
2040      end
2041    else begin
2042      case (theEvent.what) of
2043
2044        keyDown, autoKey:
2045          begin
2046          charCode := char(BAnd(theEvent.message, charCodeMask));
2047          if ((charCode = char(kReturn)) or (charCode = char(kEnter))) then
```

```
            begin
            GetDialogItem(modalDlgPtr, iOK, itemType, itemHdl, itemRect);
            HiliteControl(ControlHandle(itemHdl), kControlButtonPart);
            Delay(10, finalTicks);
            HiliteControl(ControlHandle(itemHdl), 0);
            handledEvent := true;
            itemHit := iOK;
            end;
          end;

        mouseDown:
          begin
          mouseXY := theEvent.where;
          GlobalToLocal(mouseXY);
          thePart := FindControl(mouseXY, modalDlgPtr, controlHdl);
          docRecHdl := DocRecordHandle(GetWRefCon(modalDlgPtr));
          if (controlHdl = docRecHdl^^.scrollbarHdl) then
            begin
            HandleScrollBar(modalDlgPtr, thePart, mouseXY);
            itemHit := iScrollBar;
            handledEvent := true;
            end;
          end;
        end;
          {of case statement}
      end;

    HelpDialogFilter := handledEvent;
    end;
      {of function HelpDialogFilter}

end.
  {of unit HelpDialogPascal}

{ ############################################################################################ }
```

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.

Text1Pascal.p

The main program block

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

UText1Pascal.p

The interface section

The constant declaration block

Lines 136-150 establish constants relating to menu IDs and menu item numbers. Line 152 establishes a constant for the maximum allowable number of bytes in a TextEdit edit record. Lines 153-155 establish constants representing the character codes generated by the tab, delete and return keys. Lines 157-161 establish constants relating to various resources. The constants at Lines 162-169 are used to index a 'STR#' resource so that specified error strings can be retrieved.

Exported global variables

gNumberOfWindows will keep track of the number of windows open at any one time.

The implementation section

Internal global variables

The global variable gDone controls program termination. gInBackground relates to foreground/background switching. gCursorRegion is related to the WaitNextEvent's mouseRgn parameter.

The type declaration block

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.

The procedure 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.

The procedure 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 278, HMGetHelpMenuHandle gets a handle to the Help menu record. At Line 280, 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 281 will establish the original number of help items. If the value passed to the doHelpMenu function is greater than this (Line 283), 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 284).

The procedure 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.

The procedure 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 374-375 get the current control value and the current control maximum value. At Lines 378-381, 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 377, that is, the current control value minus the value in linesToScroll as passed to the function.

Line 383 sets the control's value to the value in linesToScroll. Line 384 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 procedure was entered.

The function 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 403-405 get a pointer to the front window and a handle to the edit record associated with that window. Lines 407-408 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 409-410 saves the current clipping region before Lines 412-413 set the clipping region to the bounds of the coordinate plane. That done, Line 415 gets the current position of the cursor.

If the cursor is above the top of the window (Line 417), the text must be scrolled downwards. Accordingly, the variable linesToScroll is set to 1 (Line 419). The subsidiary procedure 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 421), TEScroll is called at Line 422 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 425), 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 433 redraws the data panel. Line 435 restores the clipping region to that established by the view rectangle and Line 437 restores the saved graphics port. Finally, Line 439 returns true. (A return of false would cause TextEdit to stop calling CustomClikLoop, as if the user had released the mouse button.)

The function 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 455 opens a new window and Line 462 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 464-466 set the text size and font. (These will be copied from the graphics port to the edit record when TENew is called.)

Line 468 creates a document record and Line 475 assigns a handle to that record to the window record's refCon field. Line 477 increments the global variable which keeps track of the number of open windows. Line 479 creates a vertical scroll bar and assigns a handle to it to the appropriate field of the document record. Lines 481-484 establish the view and destination rectangles two pixels inside the window's port rectangle less the scroll bar.

Lines 486-487 move the document record high and lock it. A monostyled edit record is then created and its handle is assigned to the appropriate field of the document record (Line 489). (If this call is not successful, the window and scroll bar are disposed of, an error alert is displayed, and the function returns (Lines 492-496).) The handle to the document record is then unlocked (Line 499).

Line 501 installs the address of the custom click loop routine CustomClikLoop in the clikLoop field of the edit record. Line 502 enables automatic scrolling for the edit record. Line 503 enables outline highlighting for the edit record.

Line 505 returns a pointer to the newly opened window.

The procedures DoOpenFile, DoOpenCommand, and DoSaveAsFile

The procedures DoOpenFile, DoOpenCommand, and DoSaveAsFile 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.

The procedure 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.

The procedure 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.

The procedure DoAdjustScrollbar

DoAdjustScrollbar adjusts the vertical scroll bar.

Lines 693-694 retrieve handles to the document record and edit record associated with the window in question.

Line 696 assigns the value in the nLines field of the edit record to the numberOfLines variable. The next action (Lines 697-699) 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 701 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 706 then sets the control maximum value. If controlMax is 0, the scroll bar is automatically unhighlighted by the SetControlMaximum call.

Line 708 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 715 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 717 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.

The function DoGetSelectLength

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

The procedure 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 749-751 get a pointer to the front window and a handle to the edit record associated with that window.

Lines 758-772 handle the Cut command. Firstly, the call to ZeroScrap attempts to empty the desk scrap. If the call succeeds, Line 761 establishes the size of the largest block in the heap that would be available if a general purge were to occur. Line 762 gets the current selection length. If the selection length is greater than the available memory (Line 763), the user is advised via an error message (Line 764). Otherwise, TECut is called at Line 766 to remove the selected text from the edit record and copy it to the TextEdit private scrap. The scroll bar is adjusted (Line 767), and TEToScrap is called at Line 768 to copy the private scrap to the desk scrap. If the latter call is not successful, Line 769 cleans up as best it can by emptying the desk scrap.

Lines 774-781 handle the Copy command. If the call to empty the desk scrap (Line 775) is successful, TECopy is called to copy the selected text from the edit record to the TextEdit private scrap. Line 777 then copies the private scrap to the desk scrap.

Lines 783-794 handle the Paste command, which must not proceed if the paste would cause the TextEdit limit of 32,767 bytes to be exceeded. Line 784 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 785-786). Otherwise, Line 788 copies the desk scrap to TextEdit's private scrap, Line 790 inserts the private scrap into the edit record, and Line 791 adjusts the scroll bar.

Lines 796-799 handle the Clear command. Line 797 deletes the current selection range from the edit record and Line 798 adjusts the scroll bar.

Lines 801-803 handle the Select All command. TESetSelect sets the selection range according to the first two parameters (selStart and selEnd).

The procedure DoFileMenu

DoFileMenu handles File menu choices, calling the appropriate application-defined functions according to the menu item chosen. In the case of SaveAs (Lines 837), 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.)

The procedure DoActivateDocWindow

DoActivateDocWindow handles window activation/deactivation.

Lines 860-861 retrieve a handle to the edit record for the window.

If the window is becoming active (Line 863), its graphics port is set as the current graphics port (Line 865). 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 873 activates the edit record associated with the window, Line 874 activates the scroll bar, and Line 875 adjusts the scroll bar.

If the window is becoming inactive (Line 878), Line 879 deactivates the edit record associated with the window and Line 880 deactivates the scroll bar.

The procedure 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.

The procedure DoAdjustMenus

DoAdjustMenus adjusts the menus. Much depends on whether any windows are currently open (Line 950).

If at least one window is open, Lines 952-954 get a handle to the edit record associated with the front window and Line 956 enables the Close item. If there is a current selection range (Line 958), the Cut, Copy, and Clear items are enabled, otherwise they are disabled. If there is any text in the desk scrap (Line 970), the Paste item is enabled, otherwise it is disabled. If there is any text in the edit record (Line 975), the SaveAs and Select All items are enabled, otherwise they are disabled.

If no windows are open (Line 985), the Close, SaveAs, Clear, and Select All items are disabled.

The procedure DoOpSysEvent

DoOpSysEvent 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.

The procedure 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.

The procedure ScrollActionProc

ScrollActionProc is associated with the vertical scroll bar. It is the hook procedure which will be repeatedly called by TrackControl (see Line 1120) while the mouse button remains down in a scroll arrow or gray area of the vertical scroll bar.

Line 1046 gets the pointer to the window which "owns" the control. Lines 1047-1048 get a handle to the edit record associated with the window.

The purpose of the branch initiated at Line 1050 is to get a value into the variable linesToScroll. If the mouse-down was in a scroll arrow, that value will be 1 (1052-1055). 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 1057-1061. (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 1065-1066 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 1068-1069 assign the current control value and the current control maximum value to two variables.

Line 1071-1075 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 1082 will not occur.

Line 1077 sets the control value to the value calculated at Line 1071, that is, to the current control value minus the value in linesToScroll.

Line 1079 sets the value in linesToScroll back to what it was before Line 1071 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 1082).

Line 1084 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.

The procedure DoInContent

DoInContent continues mouse-down processing. Lines 1106-1108 get the pointer to the front window and a handle to the edit record associated with that window.

Lines 1110-1111 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 1116 initiates a branch based on the control part code returned by FindControl at Line 1113. If the mouse-down was in one of the scroll arrows or gray areas, TrackControl is called (Line 1120). 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 1124), the old control value is saved (Line 1126) before TrackControl is called (Line 1127) to retain control until the mouse button is released. If the mouse button is released with the cursor still inside the scroll box (Line 1128), 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 1131-1132) and Line 1134 updates the data panel.

If the mouse-down was not in a control, Line 1140 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 1144. (TEClick's behaviour depends on the position of the shift key.)

The procedure DoMouseDown

DoMouseDown performs initial processing of mouse-down events. Note that, at Lines 1175-1180, 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.

The procedure DoKeyEvent

DoKeyEvent handles all key-down events that are not Command key equivalents. Lines 1208-1210 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 1212 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 1216 filters out the del key character code. TextEdit does not recognise this key, so Lines 1218-1222 provide del key support for the program. Line 1218 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 1219-1220). This, in effect, creates a selection range comprising the character following the insertion point. Line 1221 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 1222.

Processing of those character codes which have not been filtered out is initiated at Line 1225. 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 1225). 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 1228) 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 1229).

If the TextEdit limit will be exceeded by accepting the character, an alert box is invoked advising the user of the situation (Line 1232).

Line 1235 calls the application-defined function which prints data extracted from the edit and control records at the bottom of the window.

The procedure DoIdle

DoIdle is invoked whenever a NULL event is received.

Line 1248 gets a pointer to the front window, allowing Line 1250 to attempt to retrieve a handle to that window's document record. If Line 1250 was successful, Line 1252 calls TEIdle to blink the insertion point.

The procedure DoUpdate

DoUpdate handles update events. Lines 1267-1269 get the window pointer and a handle to the edit record associated with the window. Lines 1271-1272 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 1276-1280). Line 1284 resets the saved graphics port as the current port.

The procedure DoEvents

DoEvents handles initial event processing. Note that, in the case of mouse-down events, the application-defined procedure DoMouseDown is called and that, in the case of key events (Lines 1303-1313), the application-defined procedure DoKeyEvent is called provided the event is not a Command key equivalent. Other processing is minimal consistent with the demonstration aspects of the program.

The procedure EventLoop

EventLoop is the main event loop. At line 1346, the variable which controls program termination is set to false. Line 1347 creates a new empty region for the cursorRgn parameter

of the WaitNextEvent function. Line 1348 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 1350 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 1354-1355). If the event is a null event, and provided at least one window is open, the application-defined procedure DoIdle is called (Lines 1359-1361). Otherwise, the main event processing procedure is called.

UHelpDialogPascal.p

The interface section

The constant declaration block

Lines 1410-1414 establish constants relating to the dialog resource ID and items in the dialog. Lines 1415-1419 establish constants relating to the error alert resource and error strings within a 'STR#' resource.

Line 1420 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 1421-1422 represent the character codes for the Return and Enter keys.

Lines 1424-1428 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.

The type declaration block

The data types defined at Lines 1434-1449 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.

The implementation section

The procedure DoHelp

DoHelp is called when the user chooses the Text1 Help item in the Help menu. (See Lines 283-284 at UText1Pascal.p.)

Line 1495 opens the Help dialog. Lines 1502 and 1510 create a document record and assign a handle to that record to the dialog's refCon field.

Lines 1512-1513 save the current graphics port and set the dialog's graphics port as the current port. Lines 1515-1516 install an application-defined drawing function in the dialog's single user item. Lines 1518-1519 assign the handle to the scrollbar to the appropriate field of the dialog's document record.

Lines 1522-1523 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 1524 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 nil (Line 1526). At Lines 1528-1529, two global variables are assigned the resource IDs relating to the first Help topic's 'TEXT'/'styl' resource and associated 'PICT' resources.

Line 1531 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 1537 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 1543 creates an empty region, referenced by a global variable.

To complete the initial setting up, Line 1545 makes the dialog visible.

The repeat-until loop entered at Line 1547 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 1550), Line 1552 sets the scroll bar value to 0, Lines 1554-1555 determine which menu item the user chose, and Lines 1559-1571 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 1575 and 1581 then perform the same actions as did Lines 1531 and 1537 at start-up. Finally, Line 1587 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.

The procedure CloseHelp

CloseHelp closes down the Help dialog.

Lines 1608-1609 retrieve a handle to the dialog's document record. Line 1612 disposes of the region created at Line 1543. Lines 1614-1615 dispose of the edit record. Lines 1617-1622 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 1512 is restored (Lines 1623-1626).

The procedure 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 1643 saves the current pen state. Lines 1645-1651 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 1653-1655 draw the word "Topic:" to the left of the pop-up menu. Lines 1657-1665 draw the bold outline around the "Done" (OK) button.

Lines 1667-1668 get a handle to the edit record and Line 1669 retrieves the view rectangle from the edit record. The call to TEUpdate at Line 1671 draws the text in the edit record in the view rectangle. The call to DrawPictures at Line 1672 draws any pictures which might currently be located in the view rectangle.

Line 1674 restores the pen state saved at Line 1643.

The function 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 1691-1692 get a handle to the edit record.

Lines 1694-1695 set the selection range to the maximum value and then delete that selection. Line 1697 makes the destination rectangle equal to the view rectangle and Line 1698 sets the scroll bar's value to 0.

Lines 1700 and 1707 load the specified 'TEXT'/'styl' resources and Line 1714 copies the text and style information to the edit record. Lines 1716-1717 then release the 'TEXT'/'styl' resources.

Line 1719 copies the number of lines of text in the edit record to a local variable. This is used at Line 1720 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 1722), 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 1726). The scroll bar is then made active (Line 1727).

If the height of the text is less than the height of the view rectangle, the scroll bar is made inactive (Lines 1729-1731).

true is returned if the GetResource calls at Lines 1700 and 1707 did not return with false.

The function 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 1754 gets a handle to the dialog's document record. If the picInfoRecPtr field of the document record does not contain nil (Line 1756), Lines 1758-1762 release the currently loaded 'PICT' resources, dispose of the picture information records, and set the picInfoRecPtr field of the document record to nil. Line 1765 sets to 0 the field of the document record which keeps track of the number of picures associated with the current 'TEXT' resource.

Lines 1767-1768 get a handle to the edit record, then a handle to the block containing the actual text. This latter is then used at Line 1770 to assign the size of that block to a local variable. After two local variables are initialised at Lines 1771-1772, the block containing the text is locked (Line 1774).

Lines 1776-1782 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 1784-1788).

Line 1790 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 1791).

Line 1793 resets the offset value to 0.

The for loop entered at Line 1795 repeats for each of the option-space characters found. Line 1797 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 1804 finds the offset to the next option-space character and Line 1805 gets the point, based on the destination rectangle, of the bottom left of the character at that offset. Line 1807 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 1808). Line 1810 then assigns the rectangle in the picture record's picFrame field to the bounds field of the picture information record. Lines 1812-1815 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 1806.

Line 1817 assigns the number of pictures loaded to the appropriate field of the dialog's document record and Line 1819 unlocks the block containing the text.

Line 1821 returns true if Line 1801 has not previously returned with false.

The procedure HandleScrollBar

HandleScrollBar is called from HelpDialogFilter if a mouse-down event in the scrollbar is detected.

Line 1834 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 1836-1840). If TrackControl returns a non-zero value, the application-defined function for scrolling the text and pictures is called (Line 1839).

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.

The procedure 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 1861) execution falls through to the bottom of the function and the action procedure exits.

Line 1863 gets a pointer to the owner of the scrollbar, allowing Line 1864 to retrieve a handle to the dialog's document record, Line 1865 to get a handle to the edit record, Line

1866 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 1871-1877), 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 1879-1884), 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 1886-1899), 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 1901-1914), 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 1918 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 1920-1921), Line 1923 saves the current clipping region, Line 1924 sets the current clipping region to the dialog's port rectangle Line 1926 sets the scroll bar value to the required new value, and Line 1927 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 1930.

The procedure 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 procedure to draw any picture whose rectangle intersects the "vacated" area of the view rectangle.

Lines 1946-1947 get a handle to the edit record. Line 1949 determines the difference between the top of the destination rectangle and the top of the view rectangle. Line 1950 subtracts from this value the scroll bar's new value as set at Line 1949. If the result is zero, the text must be fully scrolled in one direction or the other, so the function simply returns (Lines 1951-1952).

If the text is not already fully scrolled one way or the other, Line 1954 scrolls the text in the view rectangle by the number of pixels determined at Line 1950.

If there are no pictures associated with the 'TEXT' resource in use, the function returns immediately after the text is scrolled (Lines 1956-1957).

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 1961-1969, a rectangle is made equal to the "vacated" area of the view rectangle, Lines 1962-1965 catering for the scrolling up case and Lines 1966-1969 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 1971.

The procedure DrawPictures

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

Lines 1987-1988 get handles to the dialog's document record and the edit record. Line 1990 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 1991 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 1993.

Within the loop, the picture's rectangle is retrieved from the picture information record (Line 1995) and offset to the coordinates of the view rectangle (Line 1996). Line 1998 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 2001 retrieves the picture's handle, Line 2003 checks whether the 'PICT' resource is in memory and, if necessary, loads it, Line 2004 locks the handle, Line 2008 draws the picture, and Line 2011 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.

The function 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 2058). In the case of a mouse-down, Lines 2060-2061 determines the location of the event in local coordinates. If Lines 2062-2064 determine that the mouse-down was within the scrollbar, the application-defined function HandleScrollBar is called (Line 2066) and the function reports the item hit, and the fact that it has handled the event, to the Dialog Manager (Lines 2067-2068.

Demonstration Program 2

```
{ ############################################################################################
// Text2Pascal.p
// ############################################################################################
//
// This program demonstrates:
//
// •  The use of TextEdit to enter data into the various fields of a database record.
//
// •  The formatting and display of dates, times and numbers.
//
// The program opens a window in which four data entry fields are displayed.  A separate
// TextEdit edit record is associated with each field.  The first field requires text to
// be entered, the second an integer value, the third a floating point value (that is, a
// currency value), and the fourth a date.
//
// The user may select a field for data entry using the mouse or by cycling between
// fields using the Tab key.  When the Return key is pressed, some arithmetic is
// performed using the integer and floating point values, six months are added to the
// date, and the resulting data is formatted and displayed at the bottom of the window.
//
// The current date and time is displayed at the top of the window.
//
// The program utilizes the following resources:
//
// •  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 and doesActivateOnFGSwitch
//    flags set.
//
// ############################################################################################ }

program Text2Pascal(input, output);

{ ……… include the following Universal Interfaces }

uses

  Windows, Fonts, Menus, TextEdit, Quickdraw, Dialogs, QuickdrawText, Processes, Types,
  Memory, Events, TextUtils, ToolUtils, OSUtils, Devices, SegLoad, IntlResources,
  Script, LowMem;

{ ……… define the following constants }

const

mApple = 128;
 iAbout = 1;
mFile = 129;
 iQuit = 12;
mEdit = 130;
 iCut = 3;
```

```
 iCopy = 4;
 iPaste = 5;
 iClear = 6;
 iSelectAll = 7;

kTab = $09;
kDel = $7F;
kReturn = $0D;
kMaxCharasItem = 20;
kMaxCharasQuant = 10;
kMaxCharasValue = 12;
kMaxCharasDate = 13;

rWindow = 128;
rMenubar = 128;

{ ……… user-defined types }

type

DocRec = record
  itemEditHdl : TEHandle;
  quantEditHdl : TEHandle;
  valueEditHdl : TEHandle;
  dateEditHdl : TEHandle;
  end;
DocRecPointer = ^DocRec;
DocRecHandle = ^DocRecPointer;

{ ……… global variables }

var

gDone : Boolean;
gInBackground : Boolean;
gCursorRegion : RgnHandle;
gCurrentEditRecHdl : TEHandle;
gMaxCharasThisField : integer;
gItemRect : Rect;
gQuantRect : Rect;
gValueRect : Rect;
gDateRect : Rect;
gOldRawSeconds : UInt32;

menubarHdl : Handle;
menuHdl : MenuHandle;
myWindowPtr : WindowPtr;
docRecHdl : DocRecHandle;

{ ########################################################################## DoInitManagers }

procedure DoInitManagers;

  begin
  MaxApplZone;
  MoreMasters;

  InitGraf(@qd.thePort);
  InitFonts;
  InitWindows;
  InitMenus;
  TEInit;
  InitDialogs(nil);

  InitCursor;
  FlushEvents(everyEvent, 0);
  end;
    {of procedure DoInitManagers}

{ ######################################################################## DoSetUpEditRecords }

procedure DoSetUpEditRecords(myWindowPtr : WindowPtr);

  var
  aRect : Rect;
  docRecHdl : DocRecHandle;
```

```
  begin
  docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));

  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;
  end;
    {of procedure DoSetUpEditRecords}

{ ############################################################################ DoTodaysDate }

procedure DoTodaysDate;

  var
  rawSeconds : UInt32;
  theDateString : string;

  begin
  GetDateTime(rawSeconds);
  DateString(rawSeconds, longDate, theDateString, nil);
  MoveTo(110, 22);
  DrawString(theDateString);
  end;
    {of procedure DoTodaysDate}

{ ###################################################################### DoEraseRecordDisplay }

procedure DoEraseRecordDisplay;

  var
  rectToErase : Rect;

  begin
  SetRect(rectToErase, 166, 172, 330, 278);
  EraseRect(rectToErase);
  end;
    {of procedure DoEraseRecordDisplay}

{ ############################################################################ DoDrawWindow }

procedure DoDrawWindow;

  var
  lastItemRect : Rect;

  begin
  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('Today''s date is:');
    MoveTo(38, 57);
    DrawString('Item Title:');
    MoveTo(45, 80);
    DrawString('Quantity:');
    MoveTo(36, 103);
    DrawString('Unit Value:');
    MoveTo(65, 126);
    DrawString('Date:');
    MoveTo(248, 57);
    MoveTo(118, 162);
    DrawString('Last Record Entered:');
    MoveTo(248, 57);
    DrawString('Eg: Barometers');
    MoveTo(248, 80);
    DrawString('Eg: 34');
    MoveTo(248, 103);
    DrawString('Eg: 135.58');
    MoveTo(248, 126);
    DrawString('Eg: 24 Jul 95');
    MoveTo(110, 188);
    DrawString('Item Title:');
    MoveTo(116, 208);
    DrawString('Quantity:');
    MoveTo(107, 228);
    DrawString('Unit Value:');
    MoveTo(104, 248);
    DrawString('Total Value:');
    MoveTo(98, 268);
    DrawString('Review Date:');
    end;
      {of procedure DoDrawWindow}

{ ############################################################################ DoAdjustCursor }

procedure DoAdjustCursor(myWindowPtr : WindowPtr; mouseRegion : RgnHandle);

    var
    oldPort : GrafPtr;
    arrowRegion, itemRegion, quantRegion : RgnHandle;
    valueRegion, dateRegion, iBeamRegions : RgnHandle;
    itemRect, quantRect, valueRect, dateRect : Rect;
    mouseXY : Point;

    begin
    if (gInBackground) then
      begin
      SetCursor(qd.arrow);
      Exit(DoAdjustCursor);
      end;

    GetPort(oldPort);
    SetPort(myWindowPtr);

    arrowRegion  := NewRgn;
    itemRegion   := NewRgn;
    quantRegion  := NewRgn;
    valueRegion  := NewRgn;
    dateRegion   := NewRgn;
    iBeamRegions := NewRgn;

    SetRectRgn(arrowRegion, -32768, -32768, 32766, 32766);

    itemRect  := gItemRect;
    quantRect := gQuantRect;
    valueRect := gValueRect;
    dateRect  := gDateRect;

    LocalToGlobal(itemRect.topLeft);
    LocalToGlobal(itemRect.botRight);
    RectRgn(itemRegion, itemRect);
    LocalToGlobal(quantRect.topLeft);
    LocalToGlobal(quantRect.botRight);
    RectRgn(quantRegion, quantRect);
    LocalToGlobal(valueRect.topLeft);
    LocalToGlobal(valueRect.botRight);
    RectRgn(valueRegion, valueRect);
```

```
286     LocalToGlobal(dateRect.topLeft);
287     LocalToGlobal(dateRect.botRight);
288     RectRgn(dateRegion, dateRect);
289
290     UnionRgn(itemRegion, quantRegion, iBeamRegions);
291     UnionRgn(valueRegion, iBeamRegions, iBeamRegions);
292     UnionRgn(dateRegion, iBeamRegions, iBeamRegions);
293
294     DiffRgn(arrowRegion, iBeamRegions, arrowRegion);
295
296     GetMouse(mouseXY);
297     LocalToGlobal(mouseXY);
298
299     if (PtInRgn(mouseXY, iBeamRegions)) then
300       begin
301       SetCursor(Cursor(GetCursor(iBeamCursor)^^));
302       CopyRgn(iBeamRegions, mouseRegion);
303       end
304     else begin
305       SetCursor(qd.arrow);
306       CopyRgn(arrowRegion, mouseRegion);
307       end;
308
309     DisposeRgn(arrowRegion);
310     DisposeRgn(itemRegion);
311     DisposeRgn(quantRegion);
312     DisposeRgn(valueRegion);
313     DisposeRgn(dateRegion);
314     DisposeRgn(iBeamRegions);
315
316     SetPort(oldPort);
317     end;
318       {of procedure DoAdjustCursor}
319
320   { ############################################################################ DoEditMenu }
321
322   procedure DoEditMenu(menuItem : integer);
323
324     begin
325     case (menuItem) of
326
327       iCut:
328         begin
329         TECut(gCurrentEditRecHdl);
330         end;
331
332       iCopy:
333         begin
334         TECopy(gCurrentEditRecHdl);
335         end;
336
337       iPaste:
338         begin
339         TEPaste(gCurrentEditRecHdl);
340         end;
341
342       iClear:
343         begin
344         TEDelete(gCurrentEditRecHdl);
345         end;
346
347       iSelectAll:
348         begin
349         TESetSelect(0, gCurrentEditRecHdl^^.teLength, gCurrentEditRecHdl);
350         end;
351       end;
352         {of case statement}
353     end;
354       {of procedure DoEditMenu}
355
356   { ######################################################################## DoMenuChoice }
357
358   procedure DoMenuChoice(menuChoice : longint);
359
360     var
361     menuID, menuItem : integer;
362     itemName : string;
```

```
363     daDriverRefNum : integer;
364
365     begin
366     menuID   := HiWord(menuChoice);
367     menuItem := LoWord(menuChoice);
368
369     if (menuID = 0) then
370       Exit(DoMenuChoice);
371
372     case (menuID) of
373
374       mApple:
375         begin
376         if (menuItem = iAbout) then
377           SysBeep(10)
378         else
379           begin
380           GetMenuItemText(GetMenuHandle(mApple), menuItem, itemName);
381           daDriverRefNum := OpenDeskAcc(itemName);
382           end;
383         end;
384
385       mFile:
386         begin
387         if (menuItem = iQuit) then
388           gDone := true;
389         end;
390
391       mEdit:
392         begin
393         DoEditMenu(menuItem);
394         end;
395       end;
396         {of case statement}
397
398     HiliteMenu(0);
399     end;
400       {of procedure DoMenuChoice}
401
402   { ############################################################################ DoAdjustMenus }
403
404   procedure DoAdjustMenus;
405
406     var
407     fileMenuHdl, editMenuHdl : MenuHandle;
408
409     begin
410     fileMenuHdl := GetMenuHandle(mFile);
411     editMenuHdl := GetMenuHandle(mEdit);
412
413     if (gCurrentEditRecHdl^^.selStart < gCurrentEditRecHdl^^.selEnd) then
414       begin
415       EnableItem(editMenuHdl, iCut);
416       EnableItem(editMenuHdl, iCopy);
417       EnableItem(editMenuHdl, iClear);
418       end
419     else begin
420       DisableItem(editMenuHdl, iCut);
421       DisableItem(editMenuHdl, iCopy);
422       DisableItem(editMenuHdl, iClear);
423       end;
424
425     if (TEGetScrapLen > 0) then
426       EnableItem(editMenuHdl, iPaste)
427     else
428       DisableItem(editMenuHdl, iPaste);
429
430     if (gCurrentEditRecHdl^^.teLength > 0) then
431       EnableItem(editMenuHdl, iSelectAll)
432     else
433       DisableItem(editMenuHdl, iSelectAll);
434
435     DrawMenuBar;
436     end;
437       {of procedure DoAdjustMenus}
438
439   { ########################################################################## DoAcceptDateField }
```

```
440
441 procedure DoAcceptDateField(dateEditHdl : TEHandle);
442
443   var
444   dateCacheRec : DateCacheRecord;
445   textPtr : Ptr;
446   textLength, lengthUsed : longint;
447   longDateTimeRec : LongDateRec;
448   longDateTimeValue : LongDateTime;
449   dateString : string;
450   ignored : OSErr;
451
452   begin
453   ignored := InitDateCache(@dateCacheRec);
454   textPtr := dateEditHdl^^.hText^;
455   textLength := dateEditHdl^^.teLength;
456
457   ignored := StringToDate(textPtr, textLength, @dateCacheRec, lengthUsed,
458                               longDateTimeRec);
459
460   LongDateToSeconds(longDateTimeRec, longDateTimeValue);
461   longDateTimeValue.lo := longDateTimeValue.lo + 15724800;
462   LongDateString(longDateTimeValue, longDate, dateString, nil);
463
464   MoveTo(166, 268);
465   DrawString(dateString);
466   end;
467     {of procedure DoAcceptDateField}
468
469 { ################################################################## DoAcceptValueField }
470
471 procedure DoAcceptValueField(valueEditHdl : TEHandle; quantEditHdl : TEHandle);
472
473   var
474   itl4ResourceHdl : Handle;
475   numpartsOffset : longint;
476   numpartsLength : longint;
477   numpartsTable : NumberParts;
478   formatString : string;
479   formatStringRec : NumFormatString;
480   inputNumString : string;
481   formattedNumString : string;
482   value80Bit : extended80;
483   quantityString : string;
484   quantity : longint;
485   result : FormatResultType;
486   ignored : OSErr;
487
488   begin
489   formatString := '''$''###,###,###.00;''Valueless'';''Valueless''';
490
491
492   GetIntlResourceTable(smSystemScript, iuNumberPartsTable, itl4ResourceHdl,
493                           numpartsOffset, numpartsLength);
494   numpartsTable := NumberPartsPtr(longint(itl4ResourceHdl^) + numpartsOffset)^;
495
496   ignored := StringToFormatRec(formatString, numpartsTable, formatStringRec);
497
498   GetDialogItemText(valueEditHdl^^.hText, inputNumString);
499
500   ignored := StringToExtended(inputNumString, formatStringRec, numpartsTable,
501                                 value80Bit);
502   ignored := ExtendedToString(value80Bit, formatStringRec, numpartsTable,
503                               formattedNumString);
504
505   MoveTo(166, 228);
506   DrawString(formattedNumString);
507
508   GetDialogItemText(quantEditHdl^^.hText, quantityString);
509   StringToNum(quantityString, quantity);
510   value80Bit := value80Bit * quantity;
511
512   result := ExtendedToString(value80Bit, formatStringRec, numpartsTable,
513                                 formattedNumString);
514   MoveTo(166, 248);
515   if (result = fFormatOverflow) then
516     DrawString('Too large to display')
```

```
517     else
518       DrawString(formattedNumString);
519     end;
520       {of procedure DoAcceptValueField}
521
522   { ############################################################################ DoAcceptQuantField }
523
524   procedure DoAcceptQuantField(quantEditHdl : TEHandle);
525
526     var
527     quantString : string;
528
529     begin
530     GetDialogItemText(quantEditHdl^^.hText, quantString);
531     MoveTo(166, 208);
532     DrawString(quantString);
533     end;
534       {of procedure DoAcceptQuantField}
535
536   { ############################################################################ DoAcceptItemField }
537
538   procedure DoAcceptItemField(itemEditHdl : TEHandle);
539
540     var
541     itemString : string;
542
543     begin
544     GetDialogItemText(itemEditHdl^^.hText, itemString);
545     MoveTo(166, 188);
546     DrawString(itemString);
547     end;
548       {of procedure DoAcceptItemField}
549
550   { ############################################################################ DoAcceptNewRecord }
551
552   procedure DoAcceptNewRecord;
553
554     var
555     myWindowPtr : WindowPtr;
556     docRecHdl : DocRecHandle;
557
558     begin
559     myWindowPtr := FrontWindow;
560     docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
561
562     if ((docRecHdl^^.itemEditHdl^^.teLength = 0) or
563       (docRecHdl^^.quantEditHdl^^.teLength = 0) or
564       (docRecHdl^^.valueEditHdl^^.teLength = 0) or
565       (docRecHdl^^.dateEditHdl^^.teLength = 0)) then
566       begin
567       SysBeep(10);
568       Exit(DoAcceptNewRecord);
569       end;
570
571     DoEraseRecordDisplay;
572
573     DoAcceptItemField(docRecHdl^^.itemEditHdl);
574     DoAcceptQuantField(docRecHdl^^.quantEditHdl);
575     DoAcceptValueField(docRecHdl^^.valueEditHdl, docRecHdl^^.quantEditHdl);
576     DoAcceptDateField(docRecHdl^^.dateEditHdl);
577
578     TESetSelect(0, 32767, docRecHdl^^.itemEditHdl);
579     TEDelete(docRecHdl^^.itemEditHdl);
580
581     TESetSelect(0, 32767, docRecHdl^^.quantEditHdl);
582     TEDelete(docRecHdl^^.quantEditHdl);
583
584     TESetSelect(0, 32767, docRecHdl^^.valueEditHdl);
585     TEDelete(docRecHdl^^.valueEditHdl);
586
587     TESetSelect(0, 32767, docRecHdl^^.dateEditHdl);
588     TEDelete(docRecHdl^^.dateEditHdl);
589
590     TEDeactivate(gCurrentEditRecHdl);
591     gCurrentEditRecHdl := docRecHdl^^.itemEditHdl;
592     TEActivate(docRecHdl^^.itemEditHdl);
593
```

```
  gMaxCharasThisField := kMaxCharasItem;
  end;
    {of procedure DoAcceptNewRecord}

{ ############################################################## DoChangeCurrentEditRec }

procedure DoChangeCurrentEditRec(newEditRecHdl : TEHandle; teClickFlag : Boolean;
                  mouseXY : Point; maxCharas : integer);

  begin
  TEDeactivate(gCurrentEditRecHdl);
  TEActivate(newEditRecHdl);
  gCurrentEditRecHdl := newEditRecHdl;

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

  TESetSelect(0, 32767, gCurrentEditRecHdl);

  gMaxCharasThisField := maxCharas;
  end;
    {of procedure DoChangeCurrentEditRec}

{ ################################################################################ DoActivate }

procedure DoActivate(theEvent : EventRecord);

  var
  myWindowPtr : WindowPtr;
  becomingActive : Boolean;

  begin
  myWindowPtr := WindowPtr(theEvent.message);

  becomingActive := boolean(BAnd(theEvent.modifiers, activeFlag) = activeFlag);

  if (becomingActive) then
    TEActivate(gCurrentEditRecHdl)
  else
    TEDeactivate(gCurrentEditRecHdl);
  end;
    {of procedure DoActivate}

{ ################################################################################ DoUpdate }

procedure DoUpdate(theEvent : EventRecord);

  var
  myWindowPtr : WindowPtr;
  docRecHdl : DocRecHandle;
  oldPort : GrafPtr;

  begin
  myWindowPtr := WindowPtr(theEvent.message);
  docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));

  GetPort(oldPort);
  SetPort(myWindowPtr);

  BeginUpdate(WindowPtr(theEvent.message));

  EraseRect(myWindowPtr^.portRect);
  DoDrawWindow;
  TEUpdate(myWindowPtr^.portRect, docRecHdl^^.itemEditHdl);
  TEUpdate(myWindowPtr^.portRect, docRecHdl^^.quantEditHdl);
  TEUpdate(myWindowPtr^.portRect, docRecHdl^^.valueEditHdl);
  TEUpdate(myWindowPtr^.portRect, docRecHdl^^.dateEditHdl);
  EndUpdate(WindowPtr(theEvent.message));

  SetPort(oldPort);
  end;
    {of procedure DoUpdate}

{ ############################################################################ DoInContent }

procedure DoInContent(theEvent : EventRecord);

```

```
671     var
672     myWindowPtr : WindowPtr;
673     docRecHdl : DocRecHandle;
674     mouseXY : Point;
675     shiftKeyPosition : Boolean;
676
677     begin
678     myWindowPtr := FrontWindow;
679     docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
680
681     mouseXY := theEvent.where;
682     GlobalToLocal(mouseXY);
683
684     shiftKeyPosition := BAnd(theEvent.modifiers, shiftKey) <> 0;
685
686     if (PtInRect(mouseXY, gItemRect)) then
687       begin
688       if (gCurrentEditRecHdl = docRecHdl^^.itemEditHdl) then
689         TEClick(mouseXY, shiftKeyPosition, docRecHdl^^.itemEditHdl)
690       else
691         DoChangeCurrentEditRec(docRecHdl^^.itemEditHdl,true,mouseXY,kMaxCharasItem);
692       end
693     else if (PtInRect(mouseXY, gQuantRect)) then
694       begin
695       if (gCurrentEditRecHdl = docRecHdl^^.quantEditHdl) then
696         TEClick(mouseXY, shiftKeyPosition, docRecHdl^^.quantEditHdl)
697       else
698         DoChangeCurrentEditRec(docRecHdl^^.quantEditHdl,true,mouseXY,kMaxCharasQuant);
699       end
700     else if (PtInRect(mouseXY, gValueRect)) then
701       begin
702       if (gCurrentEditRecHdl = docRecHdl^^.valueEditHdl) then
703         TEClick(mouseXY, shiftKeyPosition, docRecHdl^^.valueEditHdl)
704       else
705         DoChangeCurrentEditRec(docRecHdl^^.valueEditHdl,true,mouseXY,kMaxCharasValue);
706       end
707     else if (PtInRect(mouseXY, gDateRect)) then
708       begin
709       if (gCurrentEditRecHdl = docRecHdl^^.dateEditHdl) then
710         TEClick(mouseXY, shiftKeyPosition, docRecHdl^^.dateEditHdl)
711       else
712         DoChangeCurrentEditRec(docRecHdl^^.dateEditHdl, true, mouseXY, kMaxCharasDate);
713       end;
714     end;
715       {of procedure DoInContent}
716
717   { ############################################################################ DoMouseDown }
718
719   procedure DoMouseDown(theEvent : EventRecord);
720
721     var
722     myWindowPtr : WindowPtr;
723     partCode : integer;
724
725     begin
726     partCode := FindWindow(theEvent.where, myWindowPtr);
727
728     case (partCode) of
729
730       inMenuBar:
731         begin
732         DoAdjustMenus;
733         DoMenuChoice(MenuSelect(theEvent.where));
734         end;
735
736       inSysWindow:
737         begin
738         SystemClick(theEvent, myWindowPtr);
739         end;
740
741       inContent:
742         begin
743         if (myWindowPtr <> FrontWindow) then
744           SelectWindow(myWindowPtr)
745         else
746           DoInContent(theEvent);
747         end;
```

```
748
749       inDrag:
750         begin
751         DragWindow(myWindowPtr, theEvent.where, qd.screenBits.bounds);
752         end;
753
754       inGoAway:
755         begin
756         if (TrackGoAway(myWindowPtr, theEvent.where)) then
757           gDone := true;
758         end;
759       end;
760         {of case statement}
761     end;
762       {of procedure DoMouseDown}
763   { ############################################################################ DoHandleDelKey }
764
765   procedure DoHandleDelKey;
766
767     var
768     selectionLength : integer;
769
770     begin
771     selectionLength := gCurrentEditRecHdl^^.selEnd - gCurrentEditRecHdl^^.selStart;
772     if (selectionLength = 0) then
773       gCurrentEditRecHdl^^.selEnd := gCurrentEditRecHdl^^.selEnd + 1;
774     TEDelete(gCurrentEditRecHdl);
775     end;
776       {of procedure DoHandleDelKey}
777
778   { ############################################################################ DoHandleTabKey }
779
780   procedure DoHandleTabKey;
781
782     var
783     myWindowPtr : WindowPtr;
784     docRecHdl : DocRecHandle;
785     dummy : Point;
786
787     begin
788     myWindowPtr := FrontWindow;
789     docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
790
791     if (gCurrentEditRecHdl = docRecHdl^^.itemEditHdl) then
792       DoChangeCurrentEditRec(docRecHdl^^.quantEditHdl, false, dummy, kMaxCharasQuant)
793     else if (gCurrentEditRecHdl = docRecHdl^^.quantEditHdl) then
794       DoChangeCurrentEditRec(docRecHdl^^.valueEditHdl, false, dummy, kMaxCharasValue)
795     else if (gCurrentEditRecHdl = docRecHdl^^.valueEditHdl) then
796       DoChangeCurrentEditRec(docRecHdl^^.dateEditHdl, false, dummy, kMaxCharasDate)
797     else if (gCurrentEditRecHdl = docRecHdl^^.dateEditHdl) then
798       DoChangeCurrentEditRec(docRecHdl^^.itemEditHdl, false, dummy, kMaxCharasItem);
799     end;
800       {of procedure DoHandleTabKey}
801
802   { ############################################################################## DoKeyEvent }
803
804   procedure DoKeyEvent(charCode : char);
805
806     begin
807     if (charCode = char(kTab)) then
808       DoHandleTabKey
809     else if (charCode = char(kDel)) then
810       DoHandleDelKey
811     else if (charCode = char(kReturn)) then
812       DoAcceptNewRecord
813     else begin
814       if ((gCurrentEditRecHdl^^.teLength < gMaxCharasThisField) or
815                       (integer(charCode) < $20))
816         then TEKey(charCode, gCurrentEditRecHdl)
817         else SysBeep(10);
818       end;
819     end;
820       {of procedure DoKeyEvent}
821
822   { ############################################################################### DoEvents }
823
824   procedure DoEvents (theEvent : EventRecord);
```

```
825
826     var
827     charCode : char;
828
829     begin
830     case (theEvent.what) of
831
832       mouseDown:
833         begin
834         DoMouseDown(theEvent);
835         end;
836
837       keyDown, autoKey:
838         begin
839         charCode := chr(BAnd(theEvent.message, charCodeMask));
840         if (BAnd(theEvent.modifiers, cmdKey) <> 0) then
841           begin
842           DoAdjustMenus;
843           DoMenuChoice(MenuKey(charCode));
844           end
845         else
846           DoKeyEvent(charCode);
847         end;
848
849       updateEvt:
850         begin
851         DoUpdate(theEvent);
852         end;
853
854       activateEvt:
855         begin
856         DoActivate(theEvent);
857         end;
858
859       osEvt:
860         begin
861         if (BAnd(BSR(theEvent.message, 24), $000000FF) = suspendResumeMessage) then
862           if (BAnd(theEvent.message, resumeFlag) = 0) then
863             gInBackground := true
864           else gInBackground := false;
865         end;
866       end;
867         {of case statement}
868     end;
869       {of procedure DoEvents}
870
871   { ############################################################################## DoIdle }
872
873   procedure DoIdle;
874
875     var
876     rawSeconds : UInt32;
877     theTimeString : string;
878     theEraseRect : Rect;
879
880     begin
881     if (gCurrentEditRecHdl <> nil) then
882       TEIdle(gCurrentEditRecHdl);
883
884     GetDateTime(rawSeconds);
885     if (rawSeconds > gOldRawSeconds) then
886       begin
887       TimeString(rawSeconds, true, theTimeString, nil);
888       MoveTo(285, 22);
889       SetRect(theEraseRect, 285, 12, 335, 22);
890       EraseRect(theEraseRect);
891       DrawString(theTimeString);
892       gOldRawSeconds := rawSeconds;
893       end;
894     end;
895       {of procedure DoIdle}
896
897
898   { ############################################################################ EventLoop }
899
900   procedure EventLoop;
901
```

```
902     var
903     eventRec : EventRecord;
904     gotEvent : Boolean;
905     sleepTime : longint;
906
907     begin
908     gDone := false;
909     gCursorRegion := NewRgn;
910     sleepTime := LMGetCaretTime;
911
912     while not (gDone) do
913       begin
914       gotEvent := WaitNextEvent(everyEvent, eventRec, sleepTime, gCursorRegion);
915
916       if not (gInBackground) then
917         DoAdjustCursor(FrontWindow, gCursorRegion);
918
919       if (gotEvent) then
920         DoEvents(eventRec)
921       else
922         DoIdle;
923       end;
924
925     end;
926       {of procedure EventLoop}
927
928   { ################################################################## start of main program }
929
930   begin
931
932   gItemRect.top := 45;
933   gItemRect.left := 92;
934   gItemRect.bottom := 62;
935   gItemRect.right := 242;
936   gQuantRect.top := 68;
937   gQuantRect.left := 92;
938   gQuantRect.bottom := 85;
939   gQuantRect.right := 242;
940   gValueRect.top := 91;
941   gValueRect.left := 92;
942   gValueRect.bottom := 108;
943   gValueRect.right := 242;
944   gDateRect.top := 114;
945   gDateRect.left := 92;
946   gDateRect.bottom := 131;
947   gDateRect.right := 242;
948   GetDateTime(gOldRawSeconds);
949
950     { ……… initialize managers }
951
952     DoInitManagers;
953
954     { ……… set up menu bar and menus }
955
956     menubarHdl := GetNewMBar(rMenubar);
957     if (menubarHdl = nil) then
958       ExitToShell;
959     SetMenuBar(menubarHdl);
960     DrawMenuBar;
961
962     menuHdl := GetMenuHandle(mApple);
963     if (menuHdl = nil) then
964       ExitToShell
965     else
966       AppendResMenu(menuHdl, 'DRVR');
967
968     { ……… open window and attach document record }
969
970     myWindowPtr := GetNewWindow(rWindow, nil, WindowPtr(-1));
971     if (myWindowPtr = nil) then
972       ExitToShell;
973
974     docRecHdl := DocRecHandle(NewHandle(sizeof(DocRec)));
975     if (docRecHdl = nil) then
976       ExitToShell;
977
978     SetWRefCon(myWindowPtr, longint(docRecHdl));
```

```
979     SetPort(myWindowPtr);
980
981     TextSize(10);
982     TextFont(geneva);
983
984     { ……… set up edit records }
985
986     DoSetUpEditRecords(myWindowPtr);
987
988     { ……… enter eventLoop }
989
990     EventLoop;
991
992   end.
993
994   { ########################################################################################## }
```

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.

The constant declaration block

Lines 49-58 establish constants relating to menu IDs and menu item numbers. Lines 60-62 establish constants for the character codes generated by the tab, del, and return keys. Lines 63-66 establish constants which will be used to limit the number of characters which can be entered in a particular field. Lines 68-69 establish constants relating to resources.

The type declaration block

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.

The variable declaration block

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 93-96 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.

The procedure DoSetUpEditRecords

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

Line 133 gets a handle to the window's document record.

Line 135-136 establish a view and destination rectangle size two pixels smaller all round than the rectangle which will be drawn in the window. Line 137 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 139-149).

Line 151 assigns the handle of the first-created edit record to the global variable which will keep track of the currently activated edit record. Line 152 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.

The procedure DoTodaysDate

DoTodaysDate draws the date at the top of the window.

Line 165 gets the raw seconds value, as known to the system. Line 166 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 nil 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 168.

The procedure DoEraseRecordDisplay

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

The procedure DoDrawWindow

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

The procedure 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 270 sets the arrow cursor region to the bounds of the coordinate plane. Lines 277-288 then create four regions coinciding with the rectangles used to draw the data field outlines in the window. Lines 290-292 then combine these four regions into one region (iBeamRegions). Line 294, in effect, cuts this four-part region from the arrow region.

The procedure DoEditMenu

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

The procedure DoMenuChoice

DoMenuChoice handles Apple and File menu choices to completion and passes Edit menu choices to DoEditmenu.

The procedure DoAdjustMenus

DoAdjustMenus adjusts the menus.

If there is a selection range in the current edit record (Line 413), the Cut, Copy, and Clear items are enabled, otherwise they are disabled. If there is any text in the TextEdit private scrap (Line 425), the Paste item is enabled, otherwise it is disabled. If there is any text in the currently activated edit record (Line 430), Select All is enabled, otherwise it is disabled.

The procedure 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 initialized date cache record as a parameter. Accordingly, the first step (Line 453) is to initialize the specified date cache record.

Lines 454-455 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 457, which parses the input string and fills in the relevant fields of the long date-time record.

Line 460 converts the long date-time record to a long date-time value. Line 461 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 462. LongDateString converts the long date-time value to a long format date string formatted according to the specified international resource. (In this case, nil 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 465).

The procedure 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 which convert between floating point numbers and strings. Accordingly, Lines 492-494 get the required pointer.

Line 496 converts the number format specification string at Line 489 into the internal numeric representation required by the functions which convert between floating point numbers and strings.

With that preparation attended to, Line 498 gets the "Value" edit record text in a string. Line 500 converts that string into a floating point number of type extended (80 bits). Line 503 converts that number back to a string, formatted according to the internal numeric representation of the number format specification string. Line 506 draws that string against "Value:" in the record panel.

Lines 508-509 get the string from the "Quantity" edit record and convert this string to an long. The extended80 is then multiplied by the long. The 80-bit number is converted to a formatted string at Line 512 and, provided ExtendedToString does not return fFormatOverflow, this string is drawn against "Total Value:" in the record panel (Lines 515-518).

The procedure DoAcceptQuantField

DoAcceptQuantField is called by DoAcceptNewRecord to draw the text in the "Quantity" edit record in the record panel.

Line 530 gets the text in the "Quantity" edit record into a string, which is then drawn in the record panel against "Quantity:" (Lines 531-532).

The procedure DoAcceptItemField

DoAcceptItemField is called by DoAcceptNewRecord to draw the text in the "Item" edit record in the record panel.

Line 544 gets the text in the "Item" edit record into a string, which is then drawn in the record panel against "Item:" (Lines 545-546). (GetDialogItem is usually used to get the text from an editable text item in a dialog box. It works here because the first parameter, obtained by the GetDialogItem call in the case of dialog boxes, is actually a handle to the hText field of an edit record.)

The procedure 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 559-560 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 562-569).

Line 571 erases the right hand side of the record display panel. Lines 573-576 call application-defined functions for formatting and displaying the fields of the record. Lines 578-588 delete the text from all edit records. Lines 590-594 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.

The procedure DoChangeCurrentEditRec

DoChangeCurrentEditRec is called by both DoHandleTabKey and DoInContent to change the currently activated edit record.

Line 604 deactivates the currently activated edit record. Line 605 activates the edit record specified by the calling function and Line 606 copies its handle to the global variable which keeps track of the currently activated edit record.

If this function was called by DoInContent (Line 608), TEClick is called (Line 609) 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.

The procedure 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 630-633).

The procedure 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 655-660).

The procedure DoInContent

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

Lines 678-679 get a handle to the window's document record. Lines 681-682 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 684 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 686-713 respond to the mouse-down provided that it occurred within one of the four data entry rectangles. Lines 686-692 are typical. If the mouse-down occurred within the "Item" rectangle (Line 686), and if the associated edit record is the currently activated edit record (Line 688), TEClick is called (Line 689) 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 690), an application defined procedure is called (Line 691) 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.

The procedure DoMouseDown

DoMouseDown further processes mouse-down events. Note that, at Lines 743-746, a click in the content region of the front window results in a call to the application-defined procedure DoInContent.

The procedure DoHandleDelKey

DoHandleDelKey handles the del key, which is not supported by TextEdit. Line 771 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 772-773). Line 774 deletes the current selection range from the edit record.

The procedure 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 788-789 retrieves a handle to the document record for the window.

At Lines 791-792, if the currently activated edit record is that associated with the "Item" data entry field, an application-defined procedure 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 which may be entered in the new edit record.

Lines 793-798 do the same for the other three edit records.

The procedure 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 procedures (Lines 807-810). 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 811-812).

If the character code makes it to Line 816, 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 813-815). 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 (Line 817). 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.

The procedure 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 838-847).

The procedure 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 881-882).

Line 884 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 885), Line 887 converts the raw seconds value to a string containing the time formatted according to flags in the numeric format ('itl0') resource. (Since nil 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 888-891) and the retrieved raw seconds value is assigned to the static variable oldRawSeconds for use next time DoIdle is called.

The procedure EventLoop

EventLoop is the main event loop. Before the loop is entered, the global variable gDone is set to false (Line 908), a new empty region is created for the variable whose value which will be assigned to the cursorRgn parameter of the WaitNextEvent call (Line 909), and the current caret blink interval is retrieved for use in the sleep parameter of the WaitNextEvent call (Line 910).

The loop is entered at Line 913 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 916-917), and an idle function is called if the event is a null event (Line 922).

The main program block

The main function initializes the system software managers, sets up the menus, opens a window (Line 970), creates a document record and attaches it to the window (Lines 974-978), 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 986) and enters the main event loop.