Chapter 8

Text

It’s a rare program that doesn’t have to deal with text charactes in some form or another. Windows supports this need with a predefined window class called *edit controls,* which support basic text operations like input, display, selection, and cut-and-paste editing—much like the original (unstyled) TextEdit routines of the Macintosh Toolbox. In this chapter, we’ll learn how edit controls work, along with some other text-related topics like character sets, insertion carets, and general utility functions for working with text characters and strings.

One of the conveniences that Macintosh programmers have always taken for granted is a stable, uniform character set. No matter what model of Macintosh your program is running on, from a 128K “Skinny Mac,” vintage 1984, to a laptop Powerbook to a PowerMac 8100 with 16 MB of RAM and a 2-gigabyte hard disk, you can always confidently assume that the character code **`0x51`** stands for a capital Q, **`0xAD`** is a not-equal sign (≠), and **`0x8D`** is a lowercase c with a cedilla (ç). As in so many other cases, life is not so simple in the DOS/Windows world. Historically, programs running on PC-compatible machines have had to cope with the presence of two different character sets.

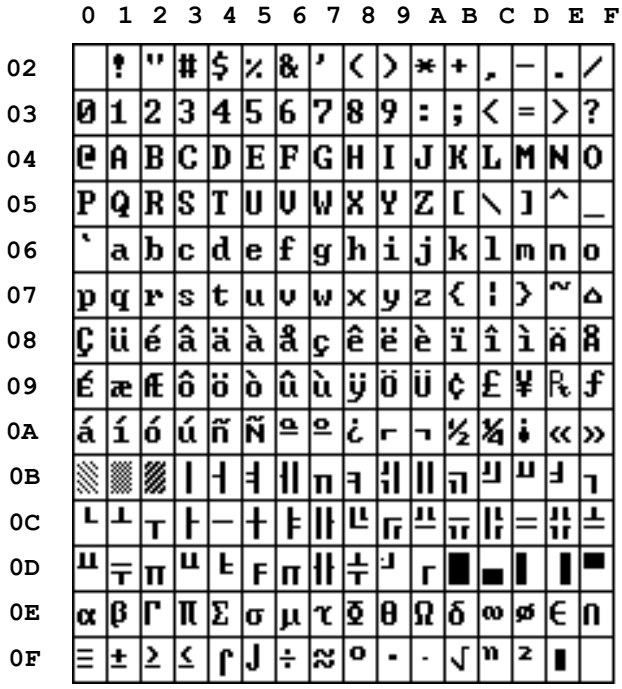
The OEM Character Set

When IBM’s engineers were designing the original PC back in the early ’80s, they knew they had to give it a character set based on the widely accepted industry standard, ASCII (American Standard Code for Information Interchange). But ASCII uses only 7 bits per character, 128 possible character codes in all; and even of those 128, 33 are assigned to nonprinting control functions and only 95 denote actual, printable characters. On a machine with an 8-bit byte, allowing 256 possible values, that left room for another 161 characters. So they scratched their heads a bit and came up with the character set shown in Figure 8–1.

Those three big blue initials guaranteed that this character set would be adopted by all of the rival PC-compatibles seeking to conform to the “IBM standard.” A few variants did appear, mainly for use in foreign countries whose languages demanded additional accents and other special characters not included in the official IBM set.

But in one form or another, virtually every PC-compatible computer sold included a native character set substantially similar to this one. Since this is the character set built into the computer by the manufacturer, it is known in Windows lingo as the *OEM character set* (for “original equipment manufacturer”).

Figure 8–1. The IBM extended character set



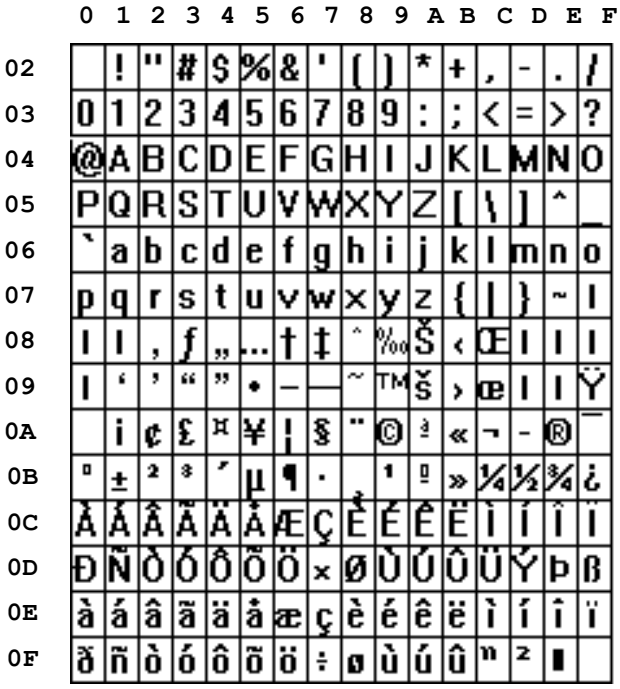
The ANSI Character Set

Perhaps the most striking feature of the IBM extended character set is its collection of line- and block-oriented graphical elements, included to allow graphical effects to be drawn on a fixed-pitch character screen of 80 columns by 24 rows. As time went on, the advent of true high-resolution bit-mapped screens made all those funny graphical characters unnecessary. So instead of using the OEM character set, early versions of Windows opted for the ANSI (American National Standards Institute) character set shown in Figure 8–2. Like the IBM version, ANSI is an 8-bit extended character set based on 7-bit ASCII, but with a more useful selection of accented, foreign, and special characters in place of the unneeded graphical elements. Windows has historically used this character set for all of its internal text handling, but still couldn’t ignore the OEM set because—guess what?—the DOS file system used OEM for its directory and file names. So Windows programs had to be able to speak both dialects, converting between them if necessary with the Windows utility functions **`CharToOem`** and **`OemToChar`**.

With Windows 95, Windows has finally broken free of its dependence on the DOS file system. In general, programs running under Windows 95 can safely afford to ignore

the OEM character set entirely. It's still supported for backward compatibility, though, and if your program is to run under older (pre–Windows 95) versions of the system, you may still need to be prepared to deal with OEM characters in places like file and path names.

Figure 8–2. The ANSI extended character set



Character and String Operations

The Win32 interface includes a set of enhanced versions of standard C library functions for manipulating character strings. The enhanced versions all have names beginning with **l** (the lowercase letter "ell," not the numeral "one"). For example, the Windows function **lstrlen** returns the length of a string in characters, like the standard C library function **strlen**. The Windows functions are designed to recognize and properly handle accented letters: for example, unlike the standard C library functions, the Windows versions will recognize the characters **Ü** and **ü** as upper- and lowercase versions of the same letter (u-umlaut). For the greatest flexibility, you should use enhanced Windows calls like **lstrlen** in your programs instead of standard C library calls like **strlen**. Table 8–1 lists the most common Windows utility functions for manipulating characters and character strings.

Function	Purpose
IsCharAlpha	Is character a letter?
IsCharAlphaNumeric	Is character a letter or digit?
IsCharUpper	Is character uppercase?
IsCharLower	Is character lowercase?
CharUpper	Convert to uppercase
CharLower	Convert to lowercase
CharToOem	Convert from ANSI to OEM
OemToChar	Convert from OEM to ANSI
lstrlen	String length
lstrcmp	Compare strings (case-sensitive)
lstrcmpi	Compare strings (case-insensitive)
lstrcpy	Copy string
lstrcat	Concatenate strings

Edit Controls

The basic Windows mechanism for working with text is the *edit control.* The real purpose for which edit controls were created was to serve as an editable text box in a dialog window, where the user can type in some sort of text for the dialog to operate on, such as a file name or a search string. In this sense, an edit record corresponds to a Macintosh dialog item of type **EditText**. Remember, though, that every Macintosh **EditText** item actually has a TextEdit record (**TERec**) associated with it for processing the user's input. The Windows edit control is actually analogous to this underlying TextEdit record, rather than just to the editable text item itself.

Like any control, an edit control actually a window—in this case, belonging to the predefined window class **EDIT**. Because it's only a control, it can't stand alone, but must be a child of some other window. Often the parent is a dialog window, but this is not a binding restriction: you can use an edit control in any kind of window. Our WiniEdit program, for instance, handles all of its text-editing operations through an edit control whose parent is an ordinary overlapped window.

On the other hand, because an edit control is a window in its own right, it can process its own mouse clicks, keystrokes, and other messages directly from the user or the system, without any intervention by the parent window. There is thus no need for Windows equivalents to Macintosh Toolbox routines like **TEKey**, for relaying keystrokes to a TextEdit record, **TEClick**, for relaying mouse clicks, and **TEUpdate**, for redrawing text on the screen in response to an update event. The built-in window procedure for class **EDIT** handles all of these operations on its own initiative by processing messages such as **WM_CHAR**, **WM_LBUTTONDOWN**, and **WM_PAINT**. An edit

control can also have its own scroll bars, in which case it handles all of its own scrolling as well, again without any explicit action by the parent window. All of this makes working with edit controls quite a bit simpler than working with Macintosh TextEdit records.

Creating an Edit Control

You create an edit control in the usual way, by calling the Windows function **CreateWindow** (or **CreateWindowEx**), giving the string **"EDIT"** as the name of the window class. As always, one of the parameters you supply is a flag word containing style options to specify the new window's appearance and properties. Table 8–2 lists the available style options pertaining specifically to edit controls, all of which carry the prefix **ES_**, for "edit style." You can also include many of the standard window (**WS_**) styles that we learned about in Chapter 4: for instance, although an edit control, by default, has no visible border on the screen, you can give it one with the window style **WS_BORDER**. In particular, since an edit control is always a child of some other window, it should always have style **WS_CHILD**.

Table 8–2. Edit control style options

Style name	Meaning
ES_MULTILINE	Multiple lines of text
ES_WANTRETURN	Enter key starts new line instead of invoking default button
ES_AUTOVSCROLL	Automatic vertical scroll
ES_AUTOHSCROLL	Automatic horizontal scroll
ES_LEFT	Left-aligned text
ES_CENTER	Centered text
ES_RIGHT	Right-aligned text
ES_UPPERCASE	Force text to uppercase
ES_LOWERCASE	Force text to lowercase
ES_OEMCONVERT	Convert text from Windows to OEM character set
ES_READONLY	Read-only text: no input or editing allowed
ES_NOHIDESEL	Don't hide selection on losing input focus
ES_PASSWORD	Mask input with asterisks (*****) or other specified character

The standard form of edit control, intended specifically for use in dialog boxes, is a *single-line* edit control. As the name implies, it accepts just a single line of text from the user, suitable for typing in a file name or other item of information to the dialog. If the user presses the Enter key while typing text, it will be interpreted as a click of the dialog's default button (typically **OK**), rather than as starting a new line within the edit control. When the user's typing reaches the end of the edit box, the default behavior is to stop accepting additional characters and to send an error notification (**EN_MAXTEXT**) back to the parent window. You can override this, however, with the style option **ES_AUTOHSCROLL**, which causes the edit control to keep on accepting

input from the keyboard and automatically scroll to the left to make room for it.

Style **ES_MULTILINE** creates a *multiline* edit control that can accommodate more than one line of text; this is the type of edit control that WiniEdit uses. If the control is part of a dialog box, however, the Enter key will still be taken to invoke the default button instead of starting a new line. The style option **ES_WANTRETURN** changes this behavior so that the Enter key simply begins a new line of text within the control. When the bottom edge is reached, the control will refuse any additional input and notify its parent with an **EN_MAXTEXT** notification—unless it carries the style **ES_AUTOVSCROLL**, in which case it will simply scroll up one line and continue to accept text from the keyboard. In addition to the automatic scrolling options, you can give the edit control explicit, visible scroll bars of its own with the standard window styles **WS_VSCROLL** and **WS_HSCROLL**.

Other style options allow you to specify the alignment of text within the edit control (**ES_LEFT**, **ES_CENTER**, **ES_RIGHT**), convert it automatically to all upper- or all lowercase (**ES_UPPERCASE**, **ES_LOWERCASE**), or prevent the user from typing text into the control or editing its contents (**ES_READONLY**). By default, the control will normally highlight and unhighlight its selection automatically when it gains or loses the input focus; the **ES_NOHIDESEL** option causes the selection to stay highlighted even when the control doesn't have the focus. For working with file or directory pathnames under the old DOS file system (in older versions of Windows), style **ES_OEMCONVERT** automatically converts all characters from the ANSI to the OEM character set. Finally, for edit boxes that accept a security password of some kind from the user, there's the special-purpose style **ES_PASSWORD**. Instead of echoing the characters of the password back to the screen, where they can be seen by unauthorized eyes, this type of edit control masks them with a substitute character. The mask character is an asterisk (*****) by default, but you can change it by sending the edit control the message **EM_SETPASSWORDCHARACTER** or find out the current mask character with **EM_GETPASSWORDCHARACTER**.

Listing 8–1 shows how WiniEdit creates its edit control. This is a slightly simplified version of the program's **DoCreate** routine, called from the window procedure when it receives the message **WM_CREATE**. Recall that this is the first message Windows sends to a window after creating it. For WiniEdit's main document window, the main order of business at creation time is to create its child, the edit control. The style options specify that the edit control is a child, that it is initially visible when created, accommodates multiple lines of left-aligned text, has a vertical scroll bar, and scrolls automatically when typing reaches the bottom edge of the control. Other parameters to **CreateWindow** give the class name (**"EDIT"**), the parent window, and the owning program instance. The control's child identifier, for use in sending notification messages to its parent, is **Edit_Control**, defined as a constant in the program's header file, **WiniEdit.h**. Its title and coordinates are left unspecified, since it has no visible title bar and its location within the parent window will be set later. After creating the edit control, WiniEdit saves its handle for future use in a global variable named **TheEditor**.

After creating an edit control, you can just leave it empty waiting for the user to type text into it, or you can explicitly give it some initial text to display. The control holds its text in a buffer in memory. The general window message **WM_SETTEXT**, which ordinarily sets a window's title, copies text into this content buffer when sent to an edit control; similarly, the message **WM_GETTEXT** returns a copy of the buffer's current contents and **WM_GETTEXTLENGTH** returns the length of the contents in characters. The message **EM_SETHANDLE**, specific to edit controls, operates directly on the buffer handle, replacing it outright with a new buffer instead of copying text into the existing one; **EM_GETHANDLE** retrieves the current buffer handle.

Listing 8-1. Handle WM_CREATE message

```
VOID DoCreate (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_CREATE message.

   {
      DWORD     editStyle;                          // Style options for edit control


      editStyle = WS_CHILD |                       // Child window
                  WS_VISIBLE |                     //    visible on screen
                  WS_VSCROLL |                     //    vertical scroll bar
                  ES_AUTOVSCROLL |                 //    vertical autoscroll
                  ES_MULTILINE |                     //    multiple lines of text
                  ES_LEFT;                         //    flush-left alignment

      TheEditor = CreateWindow ("EDIT",            // Standard edit control
                                NULL,              // No title
                                editStyle,         // Style options as above
                                0, 0, 0, 0,        // Position and size will be set later
                                thisWindow,        // Main window is the parent
                                HMENU(Edit_Control),// Child identifier for notification messages
                                ThisInstance,      // Current program instance is the owner
                                NULL);             // No special creation parameters

   }  /* end DoCreate */
```

Text Layout

Like a Macintosh TextEdit record with its destination and view rectangles, a multiline edit control wraps text against a *formatting rectangle* and clips it to a *bounding rectangle.* Since the edit control is always a child of some other window, both rectangles are expressed in client coordinates, relative to the parent window's client area. The bounding rectangle is simply the boundary of the edit control's own client area, specified at creation time or set with the standard window functions **MoveWindow** or **SetWindowPos**. All text the edit control displays on the screen is clipped to the edges of this rectangle. The formatting rectangle, analogous to the destination rectangle on the Macintosh, defines the boundary against which text is wrapped into lines. The formatting rectangle is initially the same as the bounding

rectangle, but you can change it with the message **EM_SETRECT**. This redefines the formatting rectangle, automatically rewraps the control's text to the new boundary, and redisplays the text on the screen. You can suppress the automatic redisplay by

using the message **EM_SETRECTNP** instead; **EM_GETRECT** retrieves the current formatting rectangle.

Listing 8–2. Handle WM_SIZE message

```
VOID DoSize (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_SIZE message.

   {
      INT     newWidth;                          // New width of client area
      INT     newHeight;                       // New height of client area
      RECT    textRect;                          // Formatting rectangle for wrapping text
      LPARAM  rectParam;                       // Pointer to rectangle as long-word parameter


      newWidth  = LOWORD(lParam);              // Extract new dimensions
      newHeight = HIWORD(lParam);              //    from message parameter

      MoveWindow (TheEditor,                   // Resize edit control to fit
                  0, 0,
                  newWidth, newHeight,
                  TRUE);

      rectParam = LPARAM(&textRect);           // Convert to long integer
      SendMessage (TheEditor, EM_GETRECT, 0, rectParam);   // Get formatting rectangle
      InflateRect (&textRect, -TextMargin, -TextMargin);   // Inset by text margin
      SendMessage (TheEditor, EM_SETRECT, 0, rectParam);   // Set new rectangle

   }  /* end DoSize */
```

WiniEdit's **DoSize** function (Listing 8–2) uses these facilities to adjust the size of its edit control to match that of the main document window. This function is called from WiniEdit's window procedure when it receives the message **WM_SIZE**, telling it that the main document window has been resized. The function extracts the new dimensions of the document window from the message parameters and calls the Windows function **MoveWindow** to set the edit control to these same dimensions. This explicitly sets the control's bounding rectangle and implicitly adjusts the formatting rectangle to match. Our **DoSize** function then retrieves the new formatting rectangle with the message **EM_GETRECT**, insets it by a small margin to provide some white space around the edges (using the Windows utility function **InflateRect**), and adjusts it to the smaller dimensions with the **EM_SETRECT** message. Text will now be wrapped to the smaller, inset formatting rectangle and displayed within the larger bounding rectangle, which coincides with the parent window's client area.

Once the edit control has wrapped its text to the formatting rectangle, you can find out the number of text lines by sending the message EM_GETLINECOUNT. Individual lines or characters are indexed from 0 for the first to *n* – 1 for the last, where *n* is the total number of lines or characters the control contains. The message **EM_LINEINDEX** returns the index of the first character in a given line, **EM_LINELENGTH**

gives the number of characters in the line, and **EM_GETLINE** copies

the entire contents of the line into a designated string buffer. Mapping in the other direction, **EM_LINEFROMCHAR** finds the number of the line containing a given character.

By default, Windows uses spaces and end-of-line characters (carriage returns and line feeds) to delimit words when wrapping text against the formatting rectangle. As on the Macintosh, you can modify this definition by providing your own word-break function to determine where a line may be broken. The messages for this purpose are **EM_SETWORDBREAKPROC** and **EM_GETWORDBREAKPROC**; the *Win32 Programmer's Reference* gives details on how to define the word-break procedure itself. Ordinarily, line breaks that are introduced as part of the word-wrapping process but not typed explicitly by the user are "soft," meaning that they appear on the screen but are not reflected in the text itself; with the message **EM_FMTLINES**, you can specify that such line breaks be embedded directly in the text as sequences of explicit carriage-return and line-feed characters.

Text Selection

An edit control's *selection range* is expressed as a pair of integers representing the character positions at either end of the range. As on the Macintosh, you can think of these numbers as referring to the positions *between* the characters, rather than the characters themselves. The beginning of the text is position **0** and the end is equal to the length of the text in characters. A selection range from **10** to **20** includes the tenth through nineteenth characters in the text, but not the twentieth (since position **20** falls between the nineteenth and twentieth characters). If the beginning and ending positions are equal, the selection collapses to an empty insertion point, marked by a *caret* between characters of text.

The built-in window procedure for class **EDIT** processes the user's mouse clicks in an edit control's client area and uses them to drag out a new selection range. In the process, it automatically takes care of all of the needed details, such as highlighting the selection, extending or shortening it in response to a Shift-click, and selecting at the word instead of the character level on a double click. If the edit control has the **ES_AUTOVSCROLL** or **ES_AUTOHSCROLL** style (or both), it will scroll automatically in the indicated direction when the user drags the mouse outside the formatting rectangle while selecting.

The message **EM_GETSEL** returns an edit control's current selection range, while **EM_SETSEL** sets it. Specifying a range from **0** to **-1** selects the entire contents of the edit control. WiniEdit uses this feature to implement the **Select All** command on its **Edit** menu, as shown in Listing 8–3 (**DoSelectAll**).

As we mentioned earlier, an edit control will ordinarily highlight its selection or display its insertion caret on gaining the input focus and hide them on losing it. (The style option **ES_NOHIDESEL** forces the selection or caret to remain visible even when the control doesn't have the focus.) The control marks these transitions by sending the notification messages **EN_SETFOCUS** and **EN_KILLFOCUS** to its parent window.

Listing 8–3. Handle `Select All` command

```
VOID DoSelectAll (VOID)

   // Handle Select All command.

   {
      SendMessage (TheEditor, EM_SETSEL, 0, -1);        // Select entire document

   }  /* end DoSelectAll */
```

Scrolling

In addition to scrolling automatically in the course of text input or selection, an edit control with scroll bars (specified by the style options **WS_VSCROLL** and **WS_HSCROLL**) handles all of the user's mouse actions affecting them and responds by scrolling its contents on the screen. The edit control alerts its parent window with an **EN_VSCROLL** or **EN_HSCROLL** notification before responding to the scroll request, allowing the parent to intervene if appropriate. The parent can also explicitly scroll the edit control's contents by sending it the message **EM_SCROLL** to scroll vertically by a single line or page, **EM_VSCROLL** to scroll by multiple units horizontally or vertically, or **EM_SCROLLCARET** to scroll the selection or insertion caret into view. The **EM_GETFIRSTVISIBLELINE** message returns the index of the first line of text currently visible in the edit control.

Editing Text

Edit controls support the standard cut-and-paste editing operations via the messages **WM_CUT**, **WM_COPY**, **WM_PASTE**, and **WM_CLEAR**. (They're classified as window messages, with the prefix **WM_** instead of **EM_**, because they can be sent to another form of control, combo boxes, as well as to edit controls.) **WM_CUT** removes the currently selected text from the edit control to the global clipboard, **WM_COPY** copies it to the clipboard without removing it from the edit control, **WM_CLEAR** removes it from the edit control without copying it to the clipboard, and **WM_PASTE** replaces it with the current contents of the clipboard. Another message, **EM_REPLACESEL**, replaces the current selection with a specified string of characters, rather than with the contents of the clipboard. WiniEdit uses these messages to implement the corresponding editing commands on its **Edit** menu by simply relaying them to the edit control for action. Listing 8–4 (**DoCut**) shows an example; the handling of the other editing commands is equally straightforward.

Listing 8–4. Handle `Cut` command

```
VOID DoCut (VOID)

   // Handle Cut command.

   {
      SendMessage (TheEditor, WM_CUT, 0, 0);       // Relay operation to edit control

   }  /* end DoCut */
```

Table 8-3. Edit control messages

Message type	Mac counterpart	Meaning
WM_GETTEXTLENGTH	TERec.teLength	Get length of text in characters
WM_GETTEXT	TEGetText	Get text
WM_SETTEXT	TESetText	Set text
EM_GETHANDLE	TERec.hText	Get text handle
EM_SETHANDLE	TERec.hText	Set text handle
EM_GETLINECOUNT	TERec.nLines	Get number of text lines
EM_LINELENGTH	——————	Get number of characters in given line
EM_GETLINE	——————	Copy contents of specified text line
EM_LINEINDEX	TERec.lineStarts	Get index of first character in given line
EM_LINEFROMCHAR	——————	Get index of line containing given character
EM_SCROLL	TEScroll	Scroll text vertically
EM_LINESCROLL	TEScroll	Scroll text vertically or horizontally or both
EM_SCROLLCARET	TESelView	Scroll insertion caret into view
EM_GETFIRSTVISIBLELINE	——————	Get index of first visible line
EM_GETSEL	TERec.selStart, TERec.selEnd	Get selection range
EM_SETSEL	TESetSelect	Set selection range
EM_REPLACESEL	TEInsert	Replace current selection with specified text
WM_CUT	TECut	Cut current selection to paste buffer
WM_COPY	TECopy	Copy current selection to paste buffer
WM_PASTE	TEPaste	Replace current selection with paste buffer
WM_CLEAR	TEDelete	Clear current selection
EM_CANUNDO	——————	Can last operation be undone?
EM_UNDO	——————	Undo last operation
EM_EMPTYUNDOBUFFER	——————	Disable undo of last operation
EM_GETMODIFY	——————	Has text been modified?
EM_SETMODIFY	——————	Mark or unmark text as modified
EM_SETREADONLY	——————	Set or clear read-only style
EM_SETTABSTOPS	——————	Set tab positions
EM_GETRECT	TERec.destRect	Get formatting rectangle
EM_SETRECT	TERec.destRect	Set formatting rectangle
EM_SETRECTNP	TERec.destRect	Set formatting rectangle without redrawing
EM_FMTLINES	——————	Insert hard line breaks when word-wrapping?
EM_GETWORDBREAKPROC	TERec.wordBreak	Get word-break function

EM_SETWORDBREAKPROC	**SetWordBreak**	Set word-break function
EM_GETPASSWORDCHAR	————	Get mask character for hiding passwords
EM_SETPASSWORDCHAR	————	Set mask character for hiding passwords

The **EDIT** window procedure also implements a built-in undo facility. The undo is only one level deep, meaning that only the immediately preceding editing operation can be undone: a second consecutive undo "undoes the undo," reinstating the results of the original operation. The message **EM_CANUNDO** reports whether the last editing operation can be undone; if so, **EM_UNDO** undoes it. Another related message, **EM_EMPTYUNDOBUFFER**, clears the edit control's internal undo buffer, rendering the undo operation unavailable.

An edit control maintains a flag telling whether its contents are currently "dirty"—that is, have been modified and not yet saved. This *modify flag* is initially cleared to **FALSE** when the edit control is created, and automatically set to **TRUE** whenever any editing or input operation alters the contents of the control. You can also set the state of the flag explicitly, either **TRUE** or **FALSE**, with the message **EM_SETMODIFY**. WiniEdit uses this message, for example, to clear the flag (marking the text as clean) whenever it saves the contents of its window to a file, reads in a new file, or reverts to a previously saved version of a file. It then uses the companion message **EM_GETMODIFY** to check the state of the flag before displaying its **File** menu and decide whether to enable or gray out the **Save** and **Revert to Saved...** commands.

In addition to setting the modify flag, an edit control also sends the notification **EN_CHANGE** to its parent window whenever the contents of its text change in any way; if you need to, you can use this notification to track the changes on the fly instead of polling for them after the fact with **EM_GETMODIFY**. Another notification, **EN_UPDATE**, is sent just before redisplaying the text on the screen, giving you a chance to do any preprocessing or adjustment you may need at that point. Tables 8–3 and 8–4 summarize the control messages an edit control accepts and the notifications it sends.

Table 8–4. Edit control notification messages

Notification code	Meaning
EN_CHANGE	Text contents changed
EN_UPDATE	About to redisplay text
EN_VSCROLL	About to scroll vertically
EN_HSCROLL	About to scroll horizontally
EN_SETFOCUS	Gaining input focus
EN_KILLFOCUS	Relinquishing input focus
EN_MAXTEXT	Text capacity exceeded
EN_ERRSPACE	Out of space

When a window has the input focus, it can display a *caret* to mark the insertion point for text or graphics. In particular, an edit control automatically displays an insertion caret whenever its selection range is empty (provided, of course, that the insertion point is not scrolled out of view in the window's client area). The caret is not limited to edit controls, however: any window can display one to show that it is prepared to accept input from the user. Table 8–5 lists the available Windows functions for working with the caret.

Table 8–5. Caret functions

Function	Purpose
CreateCaret	Set caret shape
DestroyCaret	Remove caret shape
ShowCaret	Display caret
HideCaret	Hide caret
GetCaretPos	Get caret position
SetCaretPos	Set caret position
GetCaretBlinkTime	Get caret blink interval
SetCaretBlinkTime	Set caret blink interval

An important thing to remember about the caret is that it is a shared resource. There isn't a separate caret for each window: there is one and only one for the entire system. Only one window at a time, the window with the current input focus, can own the caret. A window should not attempt to manipulate the caret, then, unless it has the focus. Typically, a window sets the caret's appearance on gaining the input focus and relinquishes it on losing the focus, by calling the Windows functions **CreateCaret** and **DestroyCaret** while processing the messages **WM_SETFOCUS** and **WM_KILLFOCUS**, respectively.

CreateCaret can set the caret to either a solid or gray rectangle of specified dimensions, or to some other graphical appearance defined by a bitmap supplied as a parameter. The typical caret for text insertion is a solid rectangle as high as a text character and the same width as the standard window border. **CreateCaret** will use this border width automatically if you default the width parameter to **0**, allowing the caret to scale with the resolution of the screen so that it will appear reasonable on any display device. There's normally no need for you to specify this value explicitly, but you can find it out if you need to by calling the **GetSystemMetrics** function with a selector of **SM_CXBORDER**. The caret blinks on the screen at a rate you can learn with the Windows function **GetCaretBlinkTime** or set with **SetCaretBlinkTime**.

The **GetCaretPos** and **SetCaretPos** functions retrieve and set the caret's position within a window, expressed in coordinates relative to the window's client area. **HideCaret** makes the caret invisible and **ShowCaret** displays it again. These two functions work by decrementing and incrementing a visibility count similar to the

one that controls the cursor, so they must be balanced for the caret to be visible: that is, if you hide the caret three times in a row, you must show it three times before it

will reappear on the screen. To keep the caret from being inadvertently trashed when a window's contents are repainted, the **BeginPaint** and **EndPaint** functions hide it before the repaint and show it again afterward. This is one reason for always using these functions when responding to a **WM_PAINT** message.

• The Macintosh uses TextEdit records for onscreen text entry and manipulation.	• Windows uses edit controls for onscreen text entry and manipulation.
• A Macintosh TextEdit record wraps text to a destination rectangle and clips it to a view rectangle.	• A Windows edit control wraps text to a formatting rectangle and clips it to a bounding rectangle.
• A Macintosh TextEdit record has a word-break function that can customize the way it wraps text to its destination rectangle.	• A Windows edit control has a word-break function that can customize the way it wraps text to its formatting rectangle.
• A Macintosh TextEdit record has a selection range expressed as a starting and ending character position.	• A Windows edit control has a selection range expressed as a starting and ending character position.
• A Macintosh TextEdit record supports the standard editing operations Cut, Copy, Paste, and Clear.	• A Windows edit control supports the standard editing operations Cut, Copy, Paste, and Clear.

...Only Different

• The Macintosh uses a single, standard 8-bit character set.	• Windows has historically used two different 8-bit character sets: OEM (original equipment manufacturer) and ANSI (American National Standards Institute).
• A Macintosh TextEdit record must have its keystrokes, mouse clicks, and update events relayed to it from the main event loop via the Toolbox routines **TEKey**, **TEClick**, and **TEUpdate**.	• A Windows edit control has its own window procedure and can process its own keystrokes, mouse clicks, and repaint messages.
• A Macintosh TextEdit record is not directly connected to a scroll bar; mouse clicks in the scroll bar must be explicitly relayed to the TextEdit record via the Toolbox routine	• A Windows edit control has its own scroll bars and handles its own scrolling.

TEScroll.

• A Macintosh TextEdit record must be told explicitly to display and hide its selection with the Toolbox routines **TEActivate** and **TEDeactivate**.	• A Windows edit control displays and hides its selection automatically on gaining and losing the input focus.
• A Macintosh TextEdit record doesn’t support the Undo operation.	• A Windows edit control supports the Undo operation.
• A Macintosh TextEdit record must be told to blink its insertion caret via the Toolbox routine **TEIdle**.	• A Windows edit control blinks its insertion caret automatically.