WASTE DOCUMENTATION

Written by: Marco Piovanelli (<URL:mailto:piovanel@dsi.unimi.it>)
Version: 1.1b1, June 1995

This document describes **WASTE**, a **WorldScript™-Aware Styled Text Engine** for the Macintosh which can be used as the basis for simple to moderately complex applications dealing with styled text.

WASTE has been designed from the very beginning to be compatible with **TextEdit** and TextEdit-based applications, although not everything you can do with TextEdit can be done with WASTE and vice versa.

The main features of WASTE are:

• Memory-based editor, with no limit imposed on text size.
• Requires **System 7.0** or newer.
• **WorldScript™**-aware, with one major exception: bidirectional scripts (Arabic and Hebrew) are not currently supported.
• Allows full justification of text.
• Uses offscreen graphics worlds to achieve smooth text redrawing.
• Built-in support for inline input.

Version 1.1 of WASTE adds:

• Several bugs fixes ;-)
• A mechanism for embedding **pictures** and other “objects” in the text.
• Support for **Macintosh Drag and Drop**.
• Built-in **undo** routines.
• **Hooks** for customizing text drawing and measuring.

This document assumes that you are familiar with the TextEdit model and with text handling on the Macintosh in general, including the Script Manager and the Text Services Manager.

A Brief Overview of WASTE

This section gives you a general overview of WASTE and of what it can do for your application. Since WASTE is so similar to TextEdit, a special emphasis is given to those areas in which the two models diverge.

WASTE data structures

WASTE header files contain very few type declarations, since all internal data structures are private and cannot be accessed directly. There is no type declaration for the internal format of a **WE instance**, the WASTE conterpart to a TextEdit edit record. Instead you refer to a WE instance via an opaque handle. This allows future versions of WASTE to add new functionality and new data structures painlessly, without breaking existing applications.

You should make no assumptions as to how style and line-layout information is represented internally, but you can count on the text being stored as a single relocatable block, to which you can obtain a handle. This maximizes compatibility with existing TextEdit-based applications which rely heavily on this assumption.

Long Coordinates

To allow for text taller than 32,767 pixels (a serious limitation of the TextEdit model), WASTE uses long (32-bit) coordinates to identify positions within the destination rectangle. This should not constitute a problem for your application, but be careful if you use a vertical scroll bar!

WASTE comes with an extensive set of utility functions to deal with long coordinates.

How WASTE supports inline input

Support for inline input is built in WASTE so that your application can be friendly to users of double-byte script systems with a minimal contribution of code.

Starting from version 7.1 of the system software, applications interface to inline input methods through the **Text Services Manager** (TSM). TSM routines for use by applications can be roughly divided into two sets: those which refer to **TSM documents** and those who don’t. WASTE is designed to handle internally all routines from the first set, as a TSM document record is automatically associated with each WE instance. Furthermore WASTE implements all required Apple event handlers and is responsible for properly highlighting ranges in the active input area. Your application retains responsibility for a set of just four calls, namely `InitTSMAwareApplication`, `CloseTSMAwareApplication`, `TSMEvent` and `SetTSMCursor`.

Your application may optionally install callback routines to monitor calls to the main TSM Apple event handler (`kUpdateActiveInputArea`).

Embedded Objects

WASTE 1.1 implements a mechanism to embed “objects” in the text stream as if they were ordinary glyphs. In its present, rudimentary form, this mechanism is essentially meant for inline pictures, but you can embed other data types as well, like sounds, and in a future version maybe even QuickTime™ movies.

Embedded objects are referenced by opaque handles of type `WEObjectReference`. The properties of an object are its type tag (e.g. “PICT”), its size (height and width, in pixels), a handle to the actual object data (e.g., a picture handle for PICT objects) and an optional “reference constant” for use by your application. For each object type you want to support, you install handlers to create new objects, to destroy them and to draw them (the first handler is called when a new object is to be created from a raw data handle coming from the Clipboard, from a drag or from a direct call to `WEInsertObject`). You can install an optional handler to intercept mouse clicks on a selected object.

Embedded objects can be involved in Clipboard operations and in drags, either by themselves or as part of a text stream. A special data type, called **`SOUP`**, is used by WASTE to complement the standard `TEXT`/`styl` data types. A soup handle describes zero or more objects embedded in the text stream it accompanies, their types, their sizes and the offsets where they are to be inserted.

How WASTE supports Macintosh Drag and Drop

When the Drag Manager is available, WASTE modifies the behavior of some of its routines so that clicking in the selection and dragging automatically starts a drag. It is up to your application, however, to install handlers to track and receive drags. Your handlers, in turn, can call special WASTE routines to provide standard feedback while tracking and to insert the contents of a drag into a WE instance. Both styled text and embedded graphics can be dragged to and from a WE instance, and even a mixture of the two.

NOTE: WASTE exploits the delayed data delivery feature of the Drag Manager to boost performance and reduce storage needs, but version 1.0 of the Clipping Extension doesn’t seem to work correctly with “lazy drags”, so please use version 1.1 or newer of the Macintosh Drag and Drop package.

Built-in Undo

WASTE can undo the changes made to the text (including changes affecting text styles and embedded objects) by some WASTE calls like `WEKey`, `WECut` and `WEClick` (the latter can cause text to be moved, copied or deleted by a drag-and-drop operation). This feature can be enabled or disabled at any time. Undoable operations include typing, cutting, pasting, dragging and more: see the reference section to find out which calls are undoable and, as such, modify the contents of the internal **undo buffer** associated with each WE instance. Carrying out an undoable operation when undo is enabled destroys the previous contents of the undo buffer, i.e. there is only one “level” of undo.

As a further help for your application, WE instances keep track of an internal **modification count** that lets your application find whether a given WE instance is “clean” or “dirty”.

Where WASTE differs from the TextEdit model

Some subtle and not-so-subtle differences between WASTE and TextEdit are listed below. Most of them are deliberate design choices.

• WASTE keeps track internally of whether the anchor point of the selection range is at the beginning or at the end; when extending a selection (either by shift-clicking or by using shift + arrow keys), what moves is the free endpoint of the selection, but never the anchor point. Your application can control which boundary of the selection range is treated as the anchor point using `WESetSelection` as described in the reference section.

• To select a range of words, you can double click the first word, then shift-click the last word. The first word clicked becomes the anchor word of the selection range. In the same way, you can select a range of lines by triple clicking the first line and shift-clicking the last one, and the first line clicked becomes the anchor line of the selection range.

• WASTE never draws the highlighting outside the destination rectangle, while TextEdit may highlight portions of the view rectangle outside the destination rectangle.

• TextEdit's autoscrolling works on a minimum effort basis (scroll as much as necessary to bring the selection into view, but no more). On the other hand, WASTE tries to keep the selection centered in the middle of the view rectangle.

• TextEdit honors the clip region of the graphics port it draws into, whereas WASTE resets the clip region to the view rectangle.

WASTE Reference

This sections describes all WASTE routines and their parameters in depth.

WEInstallTSMHandlers

Installs Apple event handlers for supporting inline input in the current application’s Apple event dispatch table.

```
pascal OSErr WEInstallTSMHandlers(void);
```

DESCRIPTION

`WEInstallTSMHandlers` installs the Apple event handlers required for supporting inline input in the current application’s Apple event dispatch table. You should call this function if your application is TSM aware.

After the Apple event handlers have been installed, input methods can communicate with a WE instance without the intervention of your application.

RESULT CODES

`noErr`	0	No error
`memFullErr`	-108	Out of memory

WERemoveTSMHandlers

Removes the Apple event handlers previously installed by `WEInstallTSMHandlers`.

```
pascal OSErr WERemoveTSMHandlers(void);
```

DESCRIPTION

`WERemoveTSMHandlers` removes the Apple event handlers required to support inline input which were previously installed by `WEInstallTSMHandlers`.

RESULT CODES

`noErr`	0	No error
`errAEHandlerNotFound`	-1717	TSM handlers were not installed

WENew

Creates a new WE instance and returns a handle to it.

```
pascal OSErr WENew(const LongRect *destRect, const LongRect *viewRect,
                        short flags, WEHandle *hWE);
```

Field descriptions

`destRect`	The initial destination rectangle.
`viewRect`	The initial view rectangle.

`flags`	Miscellaneous flags.
`hWE`	Pointer to a variable of type `WEHandle`.

DESCRIPTION

`WENew` creates a complete text editing environment associated with the current graphics port. You specify the initial destination and view rectangles in the local coordinates of the current graphics port, expressed in long coordinates. The value of `destRect.bottom` is immaterial, since it is dynamically updated whenever line breaks are recalculated so that `(destRect.bottom – destRect.top)` is always equal to the total pixel height of the text, including any blank lines at its end.

The initial style attributes (font, size, QuickDraw styles and color) are copied from the current graphics port. The initial alignment style is `weFlushDefault`. The initial activation state is inactive.

The `flags` parameter allows you to enable certain features on creation instead of calling `WEFeatureFlag`. One of the flags, `weDoUseTempMem`, instructs `WENew` to allocate the main data structures preferably from temporary memory and is only meaningful when passed to `WENew` (it does nothing when passed to `WEFeatureFlag`).

If the Text Services Manager is available and the client application is TSM-aware (i.e., `InitTSMAwareApplication` has been called successfully), `WENew` automatically associates the new instance with a TSM document record.

RESULT CODES

`noErr`	0	No error
`memFullErr`	-108	Out of memory

WEDispose

Disposes of a WE instance and of all associated data structures.

```
pascal void WEDispose(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WEDispose` releases all memory associated with a given WE instance, including the text handle. If you want to retain the text, you can either clone the text handle using `HandToHand` or call `WESetInfo` with `selector` set to `weText` and `*info` set to `0` **immediately** before calling `WEDispose`.

WEGetDestRect / WESetDestRect / WEGetViewRect / WESetViewRect

Get and set the values of the destination rectangle and the view rectangle.

```
pascal void WEGetDestRect(LongRect *destRect, WEHandle hWE);
pascal void WESetDestRect(const LongRect *destRect, WEHandle hWE);
pascal void WEGetViewRect(LongRect *viewRect, WEHandle hWE);
pascal void WESetViewRect(const LongRect *viewRect, WEHandle hWE);
```

Field descriptions

`destRect`	Pointer to the destination rectangle.
`viewRect`	Pointer to the view rectangle.

`hWE`	The WE instance.

DESCRIPTION

These functions allow you to get and set the destination rectangle and the view rectangle associated with the specified WE instance. The rectangles are in local coordinates. As in the TextEdit model, the destination rectangle is the area in which the text is drawn (the width of this rectangle specifies the line width used to wrap the text), while the view rectangle is the portion in which the text is actually displayed. All drawing is clipped to the intersection of these two rectangles.

When the text is scrolled, the destination rectangle is automatically offset by the scrolling amount; the view rectangle is never changed save by a call to `WESetViewRect`. The only reason for using long coordinates is to allow for text taller than 32,767 pixels when scrolling, but both the view rectangle and the horizontal coordinates of the destination rectangle must always be limited to the QuickDraw range (-32767 to 32767).

A call to `WESetDestRect` which alters the line width does not automatically trigger the recalculation of line breaks: you must call `WECalText`.

WEGetAlignment / WESetAlignment

Get and set the alignment style associated with a given WE instance.

```
enum {
      weFlushLeft = -2,
      weFlushRight,
      weFlushDefault,
      weCenter,
      weJustify
}

pascal char WEGetAlignment(WEHandle hWE);
pascal void WESetAlignment(char alignment, WEHandle hWE);
```

Field descriptions

`alignment`	The alignment style.
`hWE`	The WE instance.

DESCRIPTION

Use `WEGetAlignment` and `WESetAlignment` to get and set the alignment style associated with the specified WE instance. The alignment style applies to the whole text and can be one of the five values listed above. The `WESetAlignment` call does not affect the internal undo buffer in any way.

`WeFlushDefault` (the default value) aligns the text according to the current setting of the system global variable `SysDirection` (previously known as `TESysJust`): if you change the value of `SysDirection`, you should force a redraw of all WE instances set to this alignment style.

`WeJustify` aligns the text in the destination rectangle to both left and right margins. The specific way this effect is achieved is script-dependent.

SPECIAL CONSIDERATIONS

If your application will never need anything but left-aligned text, setting the alignment to `weFlushLeft` soon after creating a new WE instance is a good idea. Doing so disables some of the line layout calculations routinely performed by WASTE, effectively speeding up several calls, notably `WECalText`.

WEGetText

Returns a handle to the text associated with a given WE instance.

```
pascal Handle WEGetText(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WEGetText` returns a handle to the text associated with the specified WE instance; this handle contains the raw character codes without any formatting information (this information is stored elsewhere).

This handle belongs to the WE instance; you should not destroy it or modify it in any way.

WEGetTextLength

Returns the length of the text, in bytes.

```
pascal long WEGetTextLength(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WEGetTextLength` returns the length of the text, in bytes, initially zero.

WEGetChar

Returns the character code at a given byte offset.

```
pascal short WEGetChar(long offset, WEHandle hWE);
```

Field descriptions

`offset`	The byte offset to the desired character code.
`hWE`	The WE instance.

DESCRIPTION

`WEGetChar` returns the character code at a given offset inside the text handle associated with the specified WE instance. This routine always returns byte values, so when dealing with double-byte characters, it returns only one half of the character. Use `WECharByte` to determine the byte type of the character code at a given offset. If an invalid offset is specified, `WEGetChar` returns zero.

WECharByte

Returns the byte type (`smSingleByte`, `smFirstByte` or `smLastByte`) of the character code at the specified offset.

```
pascal short WECharByte(long offset, WEHandle hWE);
```

Field descriptions

`offset`	The byte offset to the desired character code.
`hWE`	The WE instance.

DESCRIPTION

`WECharByte` returns the byte type of the character code at a given offset inside the text handle associated with the specified WE instance. If an invalid offset is specified, `WECharByte` returns `smSingleByte`.

WECharType

Returns the character type of the character code at the specified offset.

```
pascal short WECharType(long offset, WEHandle hWE);
```

Field descriptions

`offset`	The byte offset to the desired character code.
`hWE`	The WE instance.

DESCRIPTION

`WECharType` returns the character type of character code at a given offset inside the text handle associated with the specified WE instance. If an invalid offset is specified, `WECharType` returns zero.

WEGetRunInfo

Returns style information associated with the text run containing the specified offset.

```
typedef struct WERunInfo {
   long               runStart;
   long               runEnd;
   short              runHeight;
   short              runAscent;
   TextStyle          runStyle;
   WEObjectReference  runObject;
} WERunInfo;

pascal void WEGetRunInfo(long offset, WERunInfo *runInfo, WEHandle hWE);
```

Field descriptions

`offset`	The byte offset to the desired character code.
`runInfo`	Pointer to a record where the requested information is returned.
`hWE`	The WE instance.

DESCRIPTION

`WEGetRunInfo` returns a `WERunInfo` record which describes the style attributes associated with the style run the specified offset belongs to. This record specifies the boundaries of the style run, font metrics information and style attributes proper. The `runObject` field can be either `NULL` or a reference to an embedded object (each embedded object is treated by WASTE like a one-character wide style run). When called for the last style run in the text, `WEGetRunInfo` returns `textLength + 1` in runEnd, instead of `textLength`.

WEContinuousStyle

Determines which text attributes are continuous over the current selection range.

```
pascal Boolean WEContinuousStyle(short *mode, TextStyle *ts, WEHandle hWE);
```

Field descriptions

`mode`	Pointer to a selector. On input, the selector specifies the attributes to test. On output, the selector specifies the attributes continuous over the selection range.
`ts`	Pointer to a TextStyle record set to the continuous attributes.
`hWE`	The WE instance.

DESCRIPTION

Call `WEContinuousStyle` to determine whether a given set of text attributes is continuous over the selection range. On input, you specify in `mode` which attributes are to be tested for continuousness. On output, `mode` specifies which ones were found to be continuous over the current selection range and the corresponding fields of `ts` are set to the continuous attributes. The function result is `TRUE` if all tested attributes are continuous, `FALSE` otherwise.

On output, the `weDoFace` bit is set in `mode` if at least one QuickDraw style is continuous over the selection range: in this case `ts.tsFace` specifies only the continuous styles. If `weDoFace` is set and `ts.tsFace` is zero (i.e., the empty set), then the whole selection range is plain text.

If the selection range is empty, the returned attributes are copied from an internal **null style record** which holds the styles to be applied to the next character typed.

If `WEContinuousStyle` detects that the keyboard script has changed since the null style record was last updated, it changes the font in the null style record to match the new keyboard script. The new font is searched among the fonts preceding the insertion point; if none is found, the default application font for the keyboard script is used.

EXAMPLES

```
short mode;
TextStyle ts;

mode = weDoAll;                              // check all attributes
WEContinuousStyle(&mode, &ts, hWE);          // ignore function result

if (mode & weDoFont)
      MyCheckFontMenu(ts.tsFont);

if (mode & weDoSize)
      MyCheckSizeMenu(ts.tsSize);

if (mode & weDoFace)
      MyCheckStyleMenu(ts.tsFace);
```

```
    if (mode & weDoColor)
```

```
        MyCheckColorMenu(&ts.tsColor);
```

WECopyRange

Makes a copy of the text, the styles and/or the embedded object data in the specified range.

```
pascal OSErr WECopyRange(long rangeStart, long rangeEnd, Handle hText,
                  StScrpHandle hStyles, WESoupHandle hSoup, WEHandle hWE);
```

Field descriptions

`rangeStart`	Offset to the beginning of the range.
`rangeEnd`	Offset to the end of the range.
`hText`	Handle to a relocatable block where a copy of the text is returned.
`hStyles`	Handle to a relocatable block where a copy of the styles is returned.
hSoup	Handle to a relocatable block where a copy of the embedded object data is returned.
`hWE`	The WE instance.

DESCRIPTION

`WECopyRange` makes a copy of the text, the style information and/or the embedded object data in the specified range. You pass valid handles in `hText`, `hStyles` and `hSoup` and these handles are resized appropriately; you can also pass `NULL` in any parameter if you don't want the corresponding information returned. The style information is returned in the standard TextEdit style scrap format (the same format used for the `styl` Clipboard data type). Be aware that while this format is very simple to use, it is also very inefficient space-wise and it can take up a lot of memory. Furthermore, if there are more than 32,767 style runs in the specified range, the `scrpNStyles` field of the style scrap will contain invalid information. The `hSoup` parameter, if supplied, is filled with information describing the objects embedded within the specified range (if any). This information can be saved and later passed to `WEInsert` to restore the embedded objects in their old places within the text stream. If there are no objects in the specified range, the `hSoup` handle is set to a zero-size block.

RESULT CODES

`noErr`	`0`	No error
`memFullErr`	`-108`	Out of memory

WECountLines

Returns the number of lines.

```
pascal long WECountLines(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WECountLines` returns the number of lines of text associated with the specified WE instance, initially one. If the last character in the text is a carriage return (ASCII 13), the last line is not taken into account by `WECountLines`.

WEGetHeight

Returns the cumulative pixel height of a given line range.

```
pascal long WEGetHeight(long startLine, long endLine, WEHandle hWE);
```

Field descriptions

`startLine`	Index to the first line in the range.
`endLine`	Index to the last line in the range.
`hWE`	The WE instance.

DESCRIPTION

`WEGetHeight` returns the cumulative pixel height of the specified line range. The `startLine` and `endLine` parameters specify positions between lines (just as byte offsets specify positions between characters): 0 specifies the top of the destination rectangle, 1 specifies the position between the first and the second line, etc. Alternatively, you can think of `startLine` and `endLine` as line indices (the first line being line zero), but in this case keep in mind that `WEGetHeight` returns the pixel height from `startLine` inclusive to `endLine` **exclusive**, while the TextEdit routine `TEGetHeight` includes both lines in the computation. `StartLine` and `endLine` are pinned to the range `0..nLines` and reordered if necessary. If the last character in the text is a carriage return (ASCII 13), the height of the last line is not taken into account by `WEGetHeight`. The data structures used by WASTE make `WEGetHeight` a cheap call (much faster then `TEGetHeight` when endLine - startLine is large).

WEGetPoint

Returns the screen position corresponding to a given text offset.

```
pascal void WEGetPoint(long offset, LongPt *thePoint, short *lineHeight,
                       WEHandle hWE);
```

Field descriptions

`offset`	A byte offset into the text.
`thePoint`	Pointer to a `LongPt` record where the corresponding screen position is returned.
`lineHeight`	Pointer to a short integer where the corresponding line height is returned.
`hWE`	The WE instance.

DESCRIPTION

`WEGetPoint` converts a text offset into a screen position, expressed in local coordinates. The screen position corresponds to the top left corner of the rectangle enclosing the character glyph at the specified offset; the height of this rectangle is returned in `lineHeight`.

WEGetOffset

Returns the offset/edge pair corresponding to a given screen position.

```
pascal long WEGetPoint(const LongPt *thePoint, char *edge, WEHandle hWE);
```

Field descriptions

`thePoint`	A screen position, in local coordinates.

`edge`	Pointer to a char where the edge value is returned.
`hWE`	The WE instance.

DESCRIPTION

`WEGetOffset` converts a screen position into a byte offset into the text. The function result is the offset to the nearest character glyph; the value returned in the `edge` parameter specifies whether the given point falls on the leading (`kLeadingEdge`) or trailing (`kTrailingEdge`) edge of this glyph.

WASTE can also return the value `kObjectEdge` in this parameter, indicating that the given position lies in the middle half of an embedded object, but this feature can be disabled by setting the compiler switch `WASTE_OBJECTS_ARE_GLYPHS` to `TRUE`.

WECalText

Recalculates line breaks and other data structures used internally to keep track of line layout for the whole text.

```
pascal OSErr WECalText(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WECalText` recalculates line breaks and other data structures related to line layout for the whole text. You normally don’t need to call this function during normal editing operations since WASTE performs all the necessary recalculations automatically. You do need to call `WECalText`, however, if you called `WEUseText` to completely replace the text or if you called some editing routines with automatic recalculation turned off (see `WEFeatureFlag` for details on how to disable automatic recalculation). `WECalText` is an expensive call which can easily take several seconds to complete, so use it sparingly.

RESULT CODES

`noErr`	0	No error
`memFullErr`	-108	Out of memory

WEUpdate

Call `WEUpdate` in response to an update event in the view rectangle.

```
pascal void WEUpdate(RgnHandle updateRgn, WEHandle hWE);
```

Field descriptions

`updateRgn`	Handle to the region to redraw, in local coordinates.
`hWE`	The WE instance.

DESCRIPTION

`WEUpdate` draws the portion of text specified by `updateRgn`. You tipically call this function after getting an update event in the view rectangle. Be sure to erase the update area to the background color before calling `WEUpdate`, otherwise the text may not be redrawn correctly.

If you pass `NULL` in `updateRgn`, the whole view rectangle is erased and redrawn.

SPECIAL CONSIDERATIONS

If you use `WEUpdate` within a standard printing loop for imaging the text to a printer, be sure to turn off offscreen drawing, otherwise the QuickDraw bottlenecks set up for printing will only intercept `_StdBits` calls instead of `_StdText` calls, with possible ill effects on print quality.

WEActivate

Call `WEActivate` when the window that owns the WE instance receives an activate event.

```
pascal void WEActivate(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WEActivate` marks the specified WE instance as active and redraws the current selection range accordingly. If a TSM document record is associated with the WE instance, `WEActivate` notifies the Text Services Manager of the change. You should call `WEActivate` before calling `WEClick` or `WEKey`; otherwise the selection range may not be drawn correctly.

WEDeactivate

Call `WEDeactivate` when the window that owns the WE instance receives a deactivate event.

```
pascal void WEDeactivate(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WEDeactivate` marks the specified WE instance as inactive and redraws the current selection range accordingly. If a TSM document record is associated with the WE instance, `WEDeactivate` notifies the Text Services Manager of the change.

WEIsActive

Call `WEDeactivate` to determine whether the specified WE instance is active or inactive.

```
pascal Boolean WEIsActive(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WEIsActive` returns `true` if the specified WE instance is active.

WEScroll

Call `WEScroll` to scroll the text within the view rectangle by a given amount of pixels.

```
pascal void WEScroll(long hOffset, long vOffset, WEHandle hWE);
```

Field descriptions

`hOffset`	Amount to scroll horizontally, in pixels.
`vOffset`	Amount to scroll vertically, in pixels.
`hWE`	The WE instance.

DESCRIPTION

`WEScroll` offsets the destination rectangle by the specified amount of pixels, horizontally and/or vertically, and it updates the text in the view rectangle to reflect the change. Positive values of `hOffset` move the text to the right. Positive values of `vOffset` move the text down.

`WEScroll` may be called internally by other WASTE routines if you enabled the auto scrolling feature: when this happens, the scroll callback routine (see the description of the `WESetInfo` routine), if present, is invoked. The scroll callback is *not* invoked, however, when you call `WEScroll` directly.

WESelView

Call `WESelView` to ensure that the current selection range is visible.

```
pascal void WESelView(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WESelView` checks to see if the current selection range (specifically, the free endpoint of the selection range) is within the view rectangle. If it isn't, `WESelView` scrolls the text to show the selection range, trying to center it in the middle of the view rectangle.

If automatic scrolling is disabled (see the description of the `WEFeatureFlag` routine), `WESelView` has no effect.

WEStopInlineSession

`WEStopInlineSession` stops the ongoing inline input session (if any) and causes all unconfirmed text in the active input area to be confirmed.

```
pascal void WEStopInlineSession(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WEStopInlineSession` terminates the ongoing input session (if any) by calling the TSM function `FixTSMDocument`, which in turn calls a component function in the current input method, which finally results in an Apple event being sent to the specified WE instance to close the active input area. If the Text Services Manager isn't available, nothing happens.

WEKey

Call `WEKey` when you receive a `keyDown` or `autoKey` event directed to the window that owns a given WE instance.

```
pascal void WEKey(short key, short modifiers, WEHandle hWE);
```

Field descriptions

`key`	The character code.
`modifiers`	The `modifiers` field of the event record.
`hWE`	The WE instance.

DESCRIPTION

`WEKey` inserts the specified character at the insertion point. If the current selection is not empty, it is replaced by the new character.

If `key` is the backspace character, `WEKey` deletes the character preceding the insertion point or, if the selection range is not empty, it deletes the current selection. Similarly, if `key` is the forward delete character (ASCII `0x7F`), `WEKey` deletes the character following the insertion point or, if the selection range is not empty, it deletes the current selection. Don't forget that these keys delete characters, not bytes.

If `key` is an arrow key, `WEKey` moves the insertion point accordingly or, if the selection range is not empty, it collapses the current selection to one of its endpoints. The `modifiers` parameter is taken into account when handling arrow keys: option+left/right arrow moves the insertion point to the nearest word boundary in the respective direction, command+left/right arrow moves the insertion point to the beginning/end of the line, option+up/down arrow moves the insertion point to the beginning/end of the text and the shift key can be used in combination with any of the above modifiers to extend or shrink the selection.

If a double-byte script system is installed and `key` is the first half of a double-byte character, `key` is not immediately inserted into the text, but rather it is cached internally. When the second byte arrives, the whole character is inserted.

If undo support is enabled, changes made by a series of `WEKey` calls (called a **typing sequence**) are recorded internally so that they can be later undone. A typing sequence can include any number of backspace and forward delete characters and is interrupted only when the insertion point is moved to a different location or when another undoable WASTE routine is called. You can call `WEIsTyping` to find out whether a `WEKey` call would be part of an ongoing typing sequence or would cause a new one to be started.

WEClick

Call `WEClick` in response to a mouse-down event in the view rectangle.

```
pascal void WEClick(Point hitPt, short modifiers, long clickTime,
                    WEHandle hWE);
```

Field descriptions

`hitPt`	The hit point in local coordinates.
`modifiers`	The `modifiers` field of the event record.
`clickTime`	The `when` field of the event record.

`hWE`	The WE instance to activate.

DESCRIPTION

`WEClick` handles key-down events directed to the view rectangle of the WE instance, retaining control until the mouse button is released. The current selection range is continuously modified as the mouse moves and the highlighting is redrawn accordingly.

If the shift key wasn't held down, the hit point becomes the new anchor point of the selection range while the position where the mouse button is released becomes the new free endpoint. If the shift key was held down, the anchor point is not changed, but the free endpoint can be moved.

A double-click selects a word, and dragging the mouse or shift-clicking afterwards extends or shrinks the selection word by word. Triple-clicks do the same, but this time line by line.

If the Drag Manager is available, clicking in the selection range and dragging starts a new drag, consisting of a single drag item. Ordinarily, this drag item has three flavors, namely `TEXT`, `styl` and `SOUP` (the latter is empty most of the times). If the selection range consists of a single embedded object, however, WASTE uses its type tag as the flavor type for the drag item, so that, for example, dragging a single picture to the desktop creates a picture clipping. If `WEClick` detects that the drop location for the drag is the trash, it deletes the original selection range (this operation is undoable, however).

You can install a callback routine which is called repeatedly while the mouse is being tracked by `WEClick`: call `WESetInfo` with `selector` set to `weClickLoop` and `*info` set to the address of your callback routine (use a UPP if you're using a PowerPC native version of WASTE). This routine is meant to be used to implement text auto-scrolling when the mouse is outside the view rectangle. This callback may be invoked by `WETrackDrag` as well (see below).

Your callback should be a function of type `WEClickLoopProcPtr`, declared as follows:

```
pascal Boolean MyClickLoop(WEHandle hWE);
```

The `hWE` parameter contains the WE instance where mouse tracking is taking place. Your callback routines should normally return `true`. Returning `false` causes mouse tracking to be immediately stopped and `WEClick` to return to its caller.

You should never call `WEClick` when the WE instance is inactive.

WETrackDrag

Call `WETrackDrag` from your application drag tracking handler to provide drag feedback for the specified WE instance.

```
pascal OSErr WETrackDrag(DragTrackingMessage message, DragReference drag,
                   WEHandle hWE);
```

Field descriptions

`message`	Selector used to distinguish the phases of a drag: should be `dragTrackingEnterWindow`, `dragTrackingInWindow` or `dragTrackingLeaveWindow`.
`drag`	The drag reference.
`hWE`	The WE instance.

DESCRIPTION

`WETrackDrag` determines whether the specified drag can be accepted and provides the necessary drag feedback, blinking the caret at the offset where the drag would be inserted, highlighting the view rectangle appropriately and removing the feedback when the drag leaves the view rectangle. When `WETrackDrag` detects that the drag has remained outside the view rectangle for more than 10 ticks, it calls the click loop routine (see `WEClick`) so that auto-scrolling can be implemented.

RESULT CODES

`noErr`	0	No error
`badDragRefErr`	-1850	Invalid drag reference

WEReceiveDrag

Call `WETrackDrag` from your application drag receive handler to insert the contents of a drag into the specified WE instance.

```
pascal OSErr WEReceiveDrag(DragReference drag, WEHandle hWE);
```

Field descriptions

`drag`	The drag reference.
`hWE`	The WE instance.

DESCRIPTION

`WEReceiveDrag` calculates the text offset corresponding to the drop location, extracts the relevant data from the drag and inserts it into the WE instance. If the drag originates from the same WE instance, the selection range is moved, rather than copied, to the new destination. A copy can be forced by holding down the option key either at the beginning or at the end of the drag. Intelligent cut-and-paste rules are applied if the corresponding feature has been enabled. The effects of `WEReceiveDrag` can be undone, if undo support is enabled.

For each item in the drag, `WEReceiveDrag` first tries to extract a `TEXT` flavor; if `TEXT` is available, it looks for the (optional) accompanying `styl` and `SOUP` information. If no `TEXT` is available, `WEReceiveDrag` tries to extract flavor types matching the registered object types, like `WEPaste` does for scrap types. For example, if you have installed a ‘`new `’ handler for ‘`snd `’ objects, `WEReceiveDrag` tries to extract a sound from the drag item; if one is found, your ‘`new `’ handler is called to initialize a new sound object which is then inserted in the text.

RESULT CODES

`noErr`	0	No error
`memFullErr`	-108	Out of memory
`badDragRefErr`	-1850	Invalid drag reference
`badDragFlavorErr`	-1852	At least one drag item contains no acceptable flavor
`dragNotAccepterErr`	-1857	Invalid drop location
`weReadOnlyErr`	-9476	The specified WE instance is read-only

WEIdle

Call `WEIdle` when your application receives a null event to ensure a regular blinking of the caret.

```
pascal void WEIdle(long *maxSleep, WEHandle hWE);
```

Field descriptions

`maxSleep`	Pointer to a long variable set to the maximum time (in ticks) that should be allowed to elapse before the next call to `WEIdle`.

`hWE`	The WE instance.

DESCRIPTION

`WEIdle` inverts the caret if the WE instance is active, the selection range is empty and if at least `CaretTime` (a system global variable) ticks have elapsed since the last time the caret was inverted. `MaxSleep` is set to the amount of time remaining before the caret must be inverted again to ensure a regular blinking. Pass `NULL` in this parameter if you don't want this value returned.

WEAdjustCursor

Call `WEAdjustCursor` periodically to give WASTE a chance to set the cursor when the mouse is within the view rectangle.

```
pascal Boolean WEAdjustCursor(Point mouseLoc, RgnHandle mouseRgn,
                        WEHandle hWE);
```

Field descriptions

`mouseLoc`	The mouse location, in global coordinates.
`mouseRgn`	Handle to a region within which the cursor is to retain its shape.
`hWE`	The WE instance.

DESCRIPTION

`WEAdjustCursor` checks to see if the given mouse location is within the view rectangle of the specified WE instance. If yes, it sets the cursor to an I-beam (or to an arrow, if the Drag Manager is available and the mouse location is within the selection range) and returns `TRUE`; otherwise `WEAdjustCursor` does not set the cursor and it returns `FALSE`. The `mouseRgn` parameter can be either `NULL` or a valid region handle. In the latter case, `rgnHandle` is intersected with a region within which the cursor is to retain its current shape.

WEGetSelection

Returns the endpoint offsets of the current selection.

```
pascal void WEGetSelection(long *selStart, long *selEnd, WEHandle hWE);
```

Field descriptions

`selStart`	Pointer to a long variable set to the start of the selection range.
`selEnd`	Pointer to a long variable set to the end of the selection range.
`hWE`	The WE instance.

DESCRIPTION

`WEGetSelection` returns the offsets to the start and the end of the current selection range. `SelStart` is always set to a value less than or equal to `selEnd`, regardless of which one is the anchor point.

WESetSelection

Use `WESetSelection` to set the selection range.

```
pascal void WESetSelection(long selStart, long selEnd, WEHandle hWE);
```

Field descriptions

`selStart`	The byte offset to the anchor point.
`selEnd`	The byte offset to the free endpoint.
`hWE`	The WE instance.

DESCRIPTION

`WESetSelection` sets the selection range and redraws the highlighting appropriately. `SelStart` and `selEnd` are pinned to the range `0..textLength` and reordered if necessary, but `selStart` always becomes the new anchor point. If auto scrolling is enabled, the text may be scrolled to make the free endpoint visible. `WESetSelection` works correctly even if the WE instance is inactive and outline highlighting is enabled, but when the WE instance is active, `WESetSelection` is optimized to highlight only the difference between the old and the new selection range.

EXAMPLES

```
      /* selects the whole text */
WESetSelection(0, 0x7FFFFFFF, hWE);

/* displays the caret at the beginning of the text */
WESetSelection(0, 0, hWE);

/* selects the range 5 to 10; 10 becomes the new anchor point */
WESetSelection(10, 5, hWE);
```

WEInsert

Inserts the specified text at the insertion point.

```
pascal OSErr WEInsert(Ptr textPtr, long textLength, StScrpHandle hStyles,
                  WESoupHandle hSoup, WEHandle hWE);
```

Field descriptions

`textPtr`	Pointer to a text buffer.
`textLength`	Size of the text buffer.
`hStyles`	Handle to a style scrap (optional).
`hSoup`	Handle to a soup (optional).
`hWE`	The WE instance.

DESCRIPTION

`WEInsert` inserts the specified text at the insertion point (if the current selection range is not empty, it is replaced by the inserted text). You can optionally specify style information and embedded object information (“soup”) accompanying the text by passing a standard TextEdit style scrap in `hStyles` and/or a `WESoupHandle` in `hSoup`. `WEInsert` calls are undoable and are affected by intelligent cut-and-paste rules if the corresponding features are enabled.

RESULT CODES

`noErr`	0	No error

`memFullErr`	-108	Out of memory
`weReadOnlyErr`	-9476	The specified WE instance is read-only

WEDelete

Deletes the selection range.

```
pascal OSErr WEDelete(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WEDelete` removes the text in the current selection range. `WEDelete` calls are undoable and are affected by intelligent cut-and-paste rules if the corresponding features are enabled.

RESULT CODES

`noErr`	0	No error
`memFullErr`	-108	Out of memory
`weReadOnlyErr`	-9476	The specified WE instance is read-only

WESetStyle

Use `WESetStyle` to modify the style attributes associated with the current selection range.

```
pascal OSErr WESetStyle(short mode, TextStyle *ts, WEHandle hWE);
```

Field descriptions

`mode`	Set of bits determining which attributes are to be changed and how.
`ts`	Pointer to a TextStyle record.
`hWE`	The WE instance.

DESCRIPTION

`WESetStyle` applies the specified style attributes to the current selection range. The `mode` parameter is a interpreted as a set of bits specifying which attributes are to be changed and how.

If `weDoAddSize` is specified, the `tsSize` field of the `ts` record is added to the font sizes in the selection range, rather than replacing them; the sum is pinned to the positive integer range.

The rules for applying QuickDraw styles (the `tsFace` field of the `ts` record) are rather complex: `tsFace` replaces the target styles outright if it is zero (i.e., the empty set) or if `weDoReplaceFace` is specified in `mode`. Otherwise `tsFace` is interpreted as a selector indicating which styles are to be altered — all other styles are left intact. What exactly happens to the styles indicated in `tsFace` depends on whether `weDoToggleFace` is specified in `mode` or not. If `weDoToggleFace` is specified, a style is turned off if it's continuous over the selection range, else it is turned on. If `weDoToggleFace` is not specified, the indicated styles are always turned on. `WESetStyle` calls are undoable.

RESULT CODES

`noErr`	0	No error
`memFullErr`	-108	Out of memory

`weReadOnlyErr`	-9476	The specified WE instance is read-only

WEUseText

Replaces the text in the specified WE instance with a given text handle.

```
pascal OSErr WEUseText(Handle hText, WEHandle hWE);
```

Field descriptions

`hText`	Handle to the text.
`hWE`	The WE instance.

DESCRIPTION

`WEUseText` replaces the text handle in the specified WE instance with the given handle. The original handle is released. You should call `WEUseText` soon after creating a WE instance with `WENew`, possibly to restore text from a previously saved file. `WEUseText` does not automatically recalculate line breaks or redraw the text: you must call `WECalText` explicitly. This call is not undoable.

RESULT CODES

`noErr`	0	No error

WEUseStyleScrap

Applies the specified style information to the current selection range.

```
pascal OSErr WEUseStyleScrap(StScrpHandle hStyles, WEHandle hWE);
```

Field descriptions

`hStyles`	Handle to a style scrap.
`hWE`	The WE instance.

DESCRIPTION

`WEUseStyleScrap` applies the specified style scrap to the selection range. This call is not undoable.

RESULT CODES

`noErr`	0	No error
`memFullErr`	-108	Out of memory

WECopy

Copies the selection range to the Clipboard.

```
pascal OSErr WECopy(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WECopy` copies the selection range to the desk scrap. Ordinarily, three scrap types are put into the scrap, i.e. the standard `TEXT`/`styl` pair plus a possibly empty `SOUP` used to save embedded object information. If the selection range consists of a single embedded object, however, its type tag is used as scrap type and its associated data is put into the scrap. For a variety of reasons, you should exercise care when calling this function for more than 32K of text. This call is not undoable.

RESULT CODES

`noErr`	0	No error
`memFullErr`	-108	Out of memory
`weEmptySelectionErr`	-10013	The selection range is empty

WECut

Copies the selection range to the Clipboard and removes it from the text.

```
pascal OSErr WECut(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WECut` combines the functions of `WECopy` and `WEDelete`. It is undoable, but the previous contents of the desk scrap are not saved. If the given WE instance is read-only, `WECut` copies the selection range to the Clipboard but does not delete it and returns an error code.

RESULT CODES

`noErr`	0	No error
`memFullErr`	-108	Out of memory
`weReadOnlyErr`	-9476	The specified WE instance is read-only
`weEmptySelectionErr`	-10013	The selection range is empty

WEPaste

Pastes the contents of the Clipboard at the insertion point.

```
pascal OSErr WEPaste(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WEPaste` first looks in the desk scrap for a pasteable item: if one is found, it is inserted into the text at the insertion point (if the selection range is not empty, it is replaced by the pasted item). `WEPaste` first looks for a `TEXT` item; if one is found, `WEPaste` looks for the (optional) accompanying `styl` and `SOUP` information. If no `TEXT` is found, `WEPaste` tries to get a scrap type matching one of the registered object types, like `WEReceiveDrag` does for flavor types.

RESULT CODES

`noErr`	`0`	No error
`noTypeErr`	`-102`	No pasteable items in the desk scrap
`memFullErr`	`-108`	Out of memory
`weReadOnlyErr`	`-9476`	The specified WE instance is read-only

WECanPaste

Determine whether the current contents of the Clipboard can be pasted into the specified WE instance.

```
pascal Boolean WECanPaste(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WECanPaste` checks the contents of the desk scrap looking for pasteable items: if one is found, it returns `TRUE`.

WEUndo

Undoes the most recent undoable operation.

```
pascal OSErr WEUndo(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WEUndo` reverses the effects of the most recent undoable operation, if any. Use `WEGetUndoInfo` to find out what kind of action can be undone by calling `WEUndo`. `WEUndo` is itself an undoable operation and the only one which decrements, rather than increment, the modification count (the modification count *is* incremented, however, when you undo an undo).

RESULT CODES

`noErr`	`0`	No error
`memFullErr`	`-108`	Out of memory
`weReadOnlyErr`	`-9476`	The specified WE instance is read-only
`weCantUndoErr`	`-10015`	The undo buffer is empty

WEClearUndo

Clears the undo buffer associated with the specified WE instance.

```
pascal void WEClearUndo(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WEClearUndo` destroys the contents of the undo buffer associated with the specified WE instance.

WEGetUndoInfo

Returns a description of the most recent undoable operation.

```
enum {
      weAKNone          =     0,    /* null action */
      weAKUnspecified   =     1,    /* action of unspecified nature */
      weAKTyping        =     2,    /* some text has been typed in */
      weAKCut           =     3,    /* the selection range has been cut */
      weAKPaste         =     4,    /* something has been pasted */
      weAKClear         =     5,    /* the selection range has been deleted */
      weAKDrag          =     6,    /* drag and drop operation */
      weAKSetStyle      =     7     /* some style has been applied */
};

pascal WEActionKind WEGetUndoInfo(Boolean *redoFlag, WEHandle hWE);
```

Field descriptions

`redoFlag`	Pointer to a Boolean variable set to `TRUE` if calling `WEUndo` would cause a “redo”.
`hWE`	The WE instance.

DESCRIPTION

`WEGetUndoInfo` returns a code describing the kind of operation that would be undone by calling `WEUndo`. For example, after calling `WECut`, `WEGetUndoInfo` would return `weAKCut`. If the undo buffer is empty, `WEGetUndoInfo` returns `weAKNone`. Unlike the other undoable operations, `WEUndo` does not change the current action kind, but rather negates the current setting of the `redoFlag`.

WEIsTyping

Determines whether a typing sequence is in progress.

```
pascal Boolean WEIsTyping(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WEIsTyping` returns `TRUE` if the specified WE instance is currently tracking a typing sequence, i.e. if the next character will add to, rather than replace, the contents of the undo buffer. `WEIsTyping` returns `FALSE` if Undo has not been enabled.

WEGetModCount

Returns the modification count for the specified WE instance.

```
pascal long WEGetModCount(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WEGetModCount` returns the modification count for the specified WE instance. This is an internal count initialized to zero by `WENew` and incremented by one by undoable WASTE calls. This call can be handy to determine whether a given WE instance is “dirty” (i.e. whether is has been modified since the last time it was saved). The modification count is actually decremented by one by `WEUndo` (unless it’s undoing an undo) so that undoing a single action that has dirtied an otherwise clean instance makes the instance clean again.

WEResetModCount

Resets the modification count for the specified WE instance and clears the undo buffer.

```
pascal void WEResetModCount(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance.

DESCRIPTION

`WEResetModCount` sets the modification count for the specified WE instance to zero and clears the undo buffer. If you use a WE instance’s built-in modification count to keep track of a document’s “dirty” state, you may want to call `WEResetModCount` every time the on-disk version of the document gets synchronized with the in-memory version, i.e. after a save or a revert command.

WEInstallObjectHandler

Installs a routine to handle the specified operation (e.g. drawing) for a given type of objects.

```
/* values for the selector parameter */

enum {
      weNewHandler      =     'new ',           /* new handler */
      weDisposeHandler  =     'free',           /* dispose handler */
      weDrawHandler     =     'draw',           /* draw handler */
      weClickHandler    =     'clik'            /* click handler */
};

pascal OSErr WEInstallObjectHandler(OSType objectType, OSType selector,
                  UniversalProcPtr handler, WEHandle hWE);
```

Field descriptions

`objectType`	Specifies the object type to which the handler applies.
`selector`	Specifies the handler type.
`handler`	The address of the handler routine.
`hWE`	The WE instance for which the handler is registered, or `NULL` for global handlers.

DESCRIPTION

Use `WEInstallObjectHandler` to register "object handlers" with WASTE. Currently, WASTE defines handlers for four different purposes:

`weNewHandler`	Creating a new object from raw data coming from the desk scrap or from a drag.
`weDisposeHandler`	Disposing of objects.
`weDrawHandler`	Drawing objects.
`weClickHandler`	Responding to mouse-down events in active objects.

A description of these handlers is given in the section about application-supplied routines.

If you pass `NULL` in `hWE`, the handler will be registered in a **global handler table** that can be accessed by all WE instances throughout your application, whereas if you pass a valid WE instance in `hWE`, the handler will be registered in an **instance-specific handler table**.

RESULT CODES

`noErr`	0	No error
`weUndefinedSelectorErr`	-50	Invalid selector
`memFullErr`	-108	Out of memory

WEInsertObject

Embeds an object at the insertion point.

```
pascal OSErr WEInsertObject(OSType objectType, Handle objectDataHandle,
                   Point objectSize, WEHandle hWE);
```

Field descriptions

`objectType`	Qualifies the data type passed in `objectDataHandle`.
`objectDataHandle`	The actual data for the object.
`objectSize`	Desired height and width for the rectangle enclosing the object (optional).
`hWE`	The WE instance.

DESCRIPTION

Use `WEInsertObject` to embed an "object" at the current insertion point (if the selection is not empty, it will be replaced by the inserted object). By the time you call `WEInsertObject`, you should have registered handlers to initialize, dispose and draw objects of the specified type. `WEInsertObject` will call the appropriate handlers to set up any additional data structures, to figure out the height and width of the rectangle enclosing the object and to draw the object.

`WEInsertObject` looks for handlers for the specified object type, first in the instance-specific handler table, then in the global handler table (see the description of `WEInstallObjectHandler`).

`WEInsertObject` will return successfully even if no handlers are found for the specified object type.

You normally pass `{0,0}` in the `objectSize` parameter, indicating that you want to use the default size for the object as calculated by the new handler. A nonzero value overrides the default size.

Here's a brief explanation of the internals of embedded object implementation. The inserted object is represented within the text stream by a single control character (ASCII `0x01`, to be precise, but this is an implementation detail you should not depend on). This control character lives in a style run of its own, marked internally by WASTE as an embedded object. An internal data structure is created to track the attributes of the object: its type, size, data handle, owner instance and "reference constant". Your application can access these attributes by using accessor functions.

RESULT CODES

`noErr`	`0`	No error
`memFullErr`	`-108`	Out of memory

WEGetSelectedObject

Use this call to find out whether an object is currently selected and to get a reference to it.

```
pascal OSErr WEGetSelectedObject(WEObjectReference *objectRef, WEHandle hWE);
```

Field descriptions

`objectRef`	Set to a reference to the selected object, or `NULL` if none is found.
`hWE`	The WE instance.

DESCRIPTION

If the selection range consists of exactly one embedded object, `WEGetSelectedObject` returns a reference to it in `objectRef`, otherwise a `weObjectNotFoundErr` result code is returned and `objectRef` is set to `NULL`.

RESULT CODES

`noErr`	`0`	No error
`weObjectNotFoundErr`	`-9477`	Object not found

WEFindNextObject

Call this routine repeatedly to find all embedded objects in a given WE instance .

```
pascal long WEFindNextObject(long offset, WEObjectReference *objectRef,
                  WEHandle hWE);
```

Field descriptions

`offset`	The search starts at `(offset + 1)`.
`objectRef`	Set to a reference to the next object, or `NULL` if none is found.
`hWE`	The WE instance.

DESCRIPTION

`WEFindNextObject` looks for the first embedded object *following* `offset`: if one is found, a reference to it is returned in `objectRef` and the byte offset to the object in the text stream is returned as function result. If there is no object following the specified offset, `WEFindNextObject` returns `-1` as function result and sets `objectRef` to `NULL`.

EXAMPLE

```
WEObjectReference objectRef = NULL;
long offset = -1;

do {
      offset = WEFindNextObject(offset, &objectRef, hWE);
```

```
        if (objectRef != NULL)
        {
              // do something with the object
        }
  } while (offset >= 0);
```

WEGetObjectType / WEGetObjectDataHandle / WEGetObjectSize / WEGetObjectOwner / WEGetObjectRefCon / WESetObjectRefCon

These calls let you access attributes of embedded objects.

```
pascal OSType WEGetObjectType(WEObjectReference objectRef);
pascal Handle WEGetObjectDataHandle(WEObjectReference objectRef);
pascal Point WEGetObjectSize(WEObjectReference objectRef);
pascal WEHandle WEGetObjectOwner(WEObjectReference objectRef);
pascal long WEGetObjectRefCon(WEObjectReference objectRef);
pascal void WESetObjectRefCon(WEObjectReference objectRef, long refCon);
```

Field descriptions

`objectRef`	The object reference.
`refCon`	A “reference constant” for use by your application.

DESCRIPTION

You typically use these accessor functions from within your object handlers.

WEFeatureFlag

Use `WEFeatureFlag` to enable, disable and test miscellaneous features of a WE instance.

```
/* values for the action parameter */

enum {
      weBitToggle =     -2,   /* toggles the specified feature */
      weBitTest   =     -1,   /* returns the current setting */
      weBitClear  =     0,    /* disables the specified feature */
      weBitSet    =     1     /* enables the specified feature */
};

/* values for feature parameter */

enum {
      weFAutoScroll = 0,            /* automatic scrolling */
      weFOutlineHilite = 2,         /* outline highlighting */
      weFReadOnly = 5,              /* disallows changes */
      weFUndo = 6,                  /* undo support */
      weFIntCutAndPaste = 7,        /* intelligent cut and paste */
      weFDragAndDrop = 8,           /* drag and drop support */
      weFInhibitRecal = 9,          /* inhibits line break recalculation */
      weFDrawOffscreen = 11         /* offscreen drawing */
};

pascal short WEFeatureFlag(short feature, short action, WEHandle hWE);
```

Field descriptions

`feature`	Identifies the feature being set or tested.
`action`	Identifies the action being performed.
`hWE`	The WE instance.

DESCRIPTION

Specify `weBitToggle`, `weBitSet`, `weBitClear` or `weBitTest` to toggle, set, clear or just test the setting of the specified feature. In all four cases, the old setting is returned. A number of features can be controlled, including automatic scrolling, outline highlighting, drag and drop editing, undo support, intelligent cut and paste and offscreen drawing. `WEFeatureFlag` can also be used to temporarily disable automatic recalculation of line breaks during editing operations and to disallow changes to the text. All features are initially set according to the `flags` parameter passed to `WENew`.

AUTOMATIC SCROLLING

When automatic scrolling is enabled, the destination rectangle is automatically scrolled to keep a particular text position centered in the middle of the view rectangle. This position is normally the insertion point or, if the selection range is not empty, the free endpoint of the range, but an input method may instruct WASTE to scroll a different range into view using an appropriate Apple event. You can set up a callback routine if you want to be notified of implicit calls to `WEScroll` (see the description of the `WESetInfo` routine). If this feature is disabled, only an explicit call to `WEScroll` can scroll the text.

OUTLINE HIGHLIGHTING

When outline highlighting is enabled, the selection range is framed with the highlight color while the WE instance is inactive. When outline highlighting is disabled, no highlighting is applied to the text while the WE instance is inactive. Contrary to the behavior of TextEdit, the caret is never drawn while the WE instance is inactive.

READ-ONLY

This feature flag disallows modifications to the text: when it's set, a number of WASTE routines do nothing and return the error code `weReadOnlyErr`. Although it may seem that simply avoiding text-modifying calls is enough to prevent changes to the text, there are a number of subtle cases difficult to catch (e.g. when WASTE receives an appropriate Apple event from an inline input component or when the user drags the selection range to the trash), hence the need for this flag.

UNDO SUPPORT

Set the `weFUndo` feature flag if you want to be able to call the `WEUndo` routine: when this feature is enabled, WASTE maintains an internal **undo buffer** holding the information needed to reverse the effect of the following calls: `WEKey`, `WEInsert`, `WEInsertObject`, `WEDelete`, `WESetStyle`, `WECut`, `WEPaste`, `WEReceiveDrag` and `WEUndo` itself.

Clearing this flag does not automatically dispose of the undo buffer (you must call `WEClearUndo` for this), so you can call any of the above routines without affecting it.

INTELLIGENT CUT AND PASTE

When this flag is set, WASTE uses "intelligent cut and paste" rules to remove and/or add extra blank characters according to the context when you call `WEInsert`, `WEDelete`, `WECut`, `WEPaste` and `WEReceiveDrag`. For example, when this flag is set, you can change the following text:

```
Returns are only accepted if the merchandise is damaged.
```

to this:

```
        Returns are accepted only if the merchandise is damaged.
```

by double-clicking the word “only” (thus selecting four characters and no spaces) and dragging it after the word “accepted” or before the word “if”. Without intelligent cut and paste, the result would look like this:

```
        Returns are  acceptedonly if the merchandise is damaged.
```

which would make drag and drop editing less useful.

NOTE: currently, the only character which may be added or removed by WASTE is the Roman space character (ASCII 32): WASTE will do nothing with blank characters found in non-Roman script systems, like the “zenkaku” (double-byte) space found in the Japanese script.

DRAG AND DROP SUPPORT

`WEClick` and `WEAdjustCursor` change their behavior to support drag and drop if the Drag Manager is available and if your application sets the `weFDragAndDrop` feature flag: `WEClick` will let clicks in the selection start a drag and `WEAdjustCursor` will change the cursor shape to an arrow when the mouse is over the selection. Please don’t forget that in order for drag and drop editing to work correctly, your application has to install drag tracking and receive handlers: WASTE won’t do this for you.

OFFSCREEN DRAWING

When offscreen drawing is enabled, text is first drawn to an offscreen buffer and then copied to the screen when an editing operation requires one or more lines to be redrawn. Since the text is always drawn in `srcOr` mode (to allow for character glyphs superimposing one another), portions of the view rectangle would need to be erased before redrawing, possibly resulting in a flicker effect. Offscreen drawing avoids this need and ensures smooth visual results. Offscreen drawing is not used when `WEUpdate` is called with a non-`NULL` `updateRgn` parameter (since the area to redraw is assumed to have already been erased anyway) or when not enough memory is available for the offscreen buffer. The offscreen buffer is allocated dynamically (possibly from temporary memory) and is always made purgeable or altogether disposed of before control is returned to the application.

INHIBITING LINE BREAK RECALCULATION

When the `weFInhibitRecal` bit is set, line breaks are not recalculated and text is not redrawn during editing operations. In certain situations, for example when you have to apply a long sequence of editing operations to a WE instance, you can achieve a significant performance boost by inhibiting line break recalculation before starting the sequence and doing a complete recalculation (with `WECalText`) when you are finished.

WEGetInfo / WESetInfo

Retrieve and set miscellaneous information associated with a specified WE instance.

```
pascal OSErr WEGetInfo(OSType selector, void *info, WEHandle hWE);
pascal OSErr WESetInfo(OSType selector, const void *info, WEHandle hWE);
```

Field descriptions

`selector`	Four-letter tag identifying the information being requested.
`info`	Pointer to storage where the requested information is to be copied to or from.

`hWE`	The WE instance.

DESCRIPTION

`WEGetInfo` and `WESetInfo` provide an extensible mechanism to retrieve and set internal fields of a WE instance without knowledge of where these fields are actually stored. The currently defined fields are all 32-bit wide, but nothing prevents the addition of fields of different sizes in a future release.

Here is a list of the selectors currently defined.

`weClickLoop`	`'clik'`	Address of the click loop callback routine.
`weCurrentDrag`	`'drag'`	Drag currently being tracked by `WEClick`, or zero if none.
`wePort`	`'port'`	Pointer to the associated graphics port.
`weRefCon`	`'refc'`	Reference constant for use by the client application.
`weScrollProc`	`'scrl'`	Address of scroll callback routine.
`weText`	`'text'`	Handle to the text.
`weTSMDocument`	`'tsmd'`	Associated TSM document ID.
`weTSMPreUpdate`	`'pre '`	Address of the TSM pre-update callback routine.
`weTSMPostUpdate`	`'post'`	Address of the TSM post-update callback routine.

The fields specified by the callback selectors (`weClickLoop`, `weScrollProc`, `weTSMPreUpdate`, etc.) can be set to `NULL` (as they are initially) to indicate that no callback should be used. If you're using a PowerPC native version of the WASTE library, don't forget to use UPPs to reference your callback routines.

The following sections provide a short description of the available callbacks.

EXAMPLES

```
/* install a click loop callback routine */

static WEClickLoopUPP sClickLoopUPP = NULL;
OSErr err;

if (!sClickLoopUPP)
      sClickLoopUPP = NewWEClickLoopProc(MyWEClickLoop);

err = WESetInfo(weClickLoop, (const void *) &sClickLoopUPP, hWE);
```

RESULT CODES

`noErr`	0	No error
`weUndefinedSelectorErr`	-50	Invalid selector

MyWEClickLoop

Your application supplies this routine to perform additional actions during a call to `WEClick` or `WETrackDrag`.

```
pascal Boolean MyWEClickLoop(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance where mouse tracking is taking place.

DESCRIPTION

The click loop callback is very similar to its TextEdit counterpart and is typically used to provide auto-scrolling behavior during calls to `WEClick` or to `WETrackDrag`.

From within your click loop routine, you typically want to sample the cursor position (using `GetMouse`) and take additional actions if it is outside the view rectangle. Notice that while `WEClick` keeps calling your click loop routine, `WETrackDrag` only calls it when the mouse has been outside the view rectangle for at least 10 ticks. Here's how you could determine whether your click loop is being called by WETrackDrag:

```
DragReference currentDrag = 0L;
Boolean fromTrackDrag =
(WEGetInfo(weCurrentDrag, (void *) &currentDrag, hWE) == noErr)
&& (currentDrag != 0L);
```

MyWEScroll

WASTE calls this application-supplied routine when the destination rectangle is changed.

```
pascal void MyWEScroll(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance being auto-scrolled.

DESCRIPTION

When the auto-scrolling feature is enabled (see `WEFeatureFlag`), `WEScroll` may be called internally in order to keep the selection range visible. If you want your application to be notified when this happens (e.g. in order to keep the scroll bars in sync with the text), you can install a scroll callback. Notice that if you call `WEScroll` directly, your callback will *not* be invoked.

The scroll callback will also be called when an editing action changes the text height, and therefore the destination rectangle: `destRect.bottom` will be updated so that `(destRect.bottom - destRect.top)` equals the pixel height of the whole text, including any blank lines at the bottom.

MyWETSMPreUpdate

WASTE calls this application-supplied routine immediately before handling an Update Active Input Area Apple event sent by a text service component.

```
pascal void MyWETSMPreUpdate(WEHandle hWE);
```

Field descriptions

`hWE`	The WE instance where inline input is taking place.

DESCRIPTION

This callback was provided in WASTE 1.0 mainly for compatibility with existing TextEdit-based applications relying on the TSMTE extension to provide inline input support. Please refer to the technical note TE 27, *Inline Input for TextEdit with TSMTE*, for information about the TSMTE extension. A typical use of this callback in WASTE 1.0 (and in TextEdit) is to save information needed to implement the Undo functionality. This use is no longer necessary in WASTE 1.1 if you use the built-in Undo routines, designed to work seamlessly with inline input.

MyWETSMPostUpdate

WASTE calls this application-supplied routine immediately after handling an Update Active Input Area Apple event sent by a text service component.

```
pascal void MyWETSMPostUpdate(WEHandle hWE,
            long fixLength, long inputAreaStart, long inputAreaEnd,
            long pinRangeStart, long pinRangeEnd);
```

Field descriptions

hWE	The WE instance where inline input is taking place.
fixLength	The length of the confirmed text in the active input area.
inputAreaStart	Offset to the beginning of the active input area.
inputAreaEnd	Offset to the end of the active input area.
pinRangeStart	Offset to the beginning of the range to scroll into view.
pinRangeEnd	Offset to the end of the range to scroll into view.

DESCRIPTION

Like the TSM pre-update routine (see above), this callback is provided mainly for compatibility with existing TextEdit-based applications relying on the TSMTE extension to provide inline input support. Typical uses of this callback in TextEdit-based applications include updating the scroll bars (in case the text was scrolled or the total text height changed), keeping track of Undo information and marking a document as "dirty". In WASTE 1.1 you can use a scroll callback to update scroll bars and, if you use the built-in Undo support, you can use `WEGetModCount` to determine if a document has been "dirtied" by an Update Active Input Area event.

MyWENewObject

WASTE calls this application-supplied routine when a new embedded object must be created from raw data coming from the Clipboard, from a drag or from a direct call to `WEInsertObject`.

```
pascal OSErr MyWENewObject(Point *objectSize,
                  WEObjectReference objectRef);
```

Field descriptions

`objectSize`	The default height and width of the object.
`objectRef`	Reference to the embedded object being created.

DESCRIPTION

`WEInsertObject` calls this handler when creating a new embedded object from raw data (`WEInsertObject`, in turn, may be called internally by other WASTE routines, like `WEPaste` and `WEReceiveDrag`). Your handler can examine the raw data handle (using `WEGetObjectDataHandle`), manipulate it if necessary and possibly associate auxiliary data structures with the object (each object has a “reference constant” that you can use for this purpose). Finally, your handler should set `*objectSize` the size (height and width, in pixels) of the rectangle which is to enclose the object when it’s drawn.

EXAMPLES

```
/* new handler for PICT objects */

pascal OSErr MyHandleNewPicture(Point *objectSize,
                  WEObjectReference objectRef)
{
      PicHandle thePicture;
      Rect theFrame;

/* get handle to object data (in this case, a picture handle) */
      thePicture = (PicHandle) WEGetObjectDataHandle(objectRef);

/* figure out object size by looking at the picFrame record */
      theFrame = (*thePicture)->picFrame;
      OffsetRect(&theFrame, -theFrame.left, -theFrame.top);
      *objectSize = botRight(theFrame);

      return noErr;
}
```

MyWEDisposeObject

WASTE calls this application-supplied routine when it needs to delete an embedded object.

```
pascal OSErr MyWEDisposeObject(WEObjectReference objectRef);
```

Field descriptions

`objectRef`	Reference to the embedded object being deleted.

DESCRIPTION

Your handler should do whatever action is necessary to destroy the specified embedded object, including disposing of the data handle associated with the object and of any other additional data structures set up by the new handler. If you don’t supply a dispose handler, WASTE will just call `DisposeHandle` on the object data handle.

MyWEDrawObject

WASTE calls this application-supplied routine to draw embedded objects.

```
pascal OSErr MyWEDrawObject(const Rect *destRect,
                  WEObjectReference objectRef);
```

Field descriptions

`destRect`	The rectangle in which the object must be drawn, in local coordinates.
`objectRef`	Reference to the embedded object to draw.

DESCRIPTION

WASTE calls this handler to draw an embedded object. The QuickDraw graphics port will have been set up correctly.

EXAMPLES

```
/* draw handler for PICT objects */

pascal OSErr MyHandleDrawPicture(const Rect *destRect,
                  WEObjectReference objectRef)
{
      PicHandle thePicture;

/* get handle to object data (in this case, a picture handle) */
      thePicture = (PicHandle) WEGetObjectDataHandle(objectRef);

/* draw the picture */
      DrawPicture(thePicture, destRect);

      return noErr;
}
```

MyWEClickObject

WASTE calls this application-supplied routine to give you a chance to intercept mouse clicks in a selected object.

```
pascal Boolean MyWEClickObject(Point hitPt, short modifiers,
                  long clickTime, WEObjectReference objectRef);
```

Field descriptions

`hitPt`	The hit point, in local coordinates.
`modifiers`	The modifiers field of the mouse-down event record.
`clickTime`	The when field of the mouse-down event record.
`objectRef`	Reference to the embedded object to draw.

DESCRIPTION

WASTE calls this handler when a *selected* object is clicked. Your handler should determine whether it wants to intercept the click, in which case it should return `TRUE`, or whether it wants WASTE to handle the click normally, in which case it should return `FALSE`. Typically, your handler will want to intercept double clicks and leave single clicks to WASTE (intercepting *all* clicks indiscriminately is not a good idea, because it stops the user from starting a drag by clicking in the selected object). To make life easier for your handlers, WASTE sets the low bit of modifiers on double clicks.

EXAMPLES

```
/* click handler for sound objects */

pascal Boolean HandleClickSound(Point hitPt, short modifiers,
            long clickTime, WEObjectReference objectRef)
{
      SndListHandle     theSound;

      if (modifiers & 0x0001)       // look for double-clicks
      {
            theSound = (SndListHandle) WEGetObjectDataHandle(objectRef);
```

```
        SndPlay(NULL, theSound, false);
        return true;
}
```

```
        else
                return false;
    }
```

Distribution & Licensing

You can use the WASTE library in any way you like in freeware, shareware and commercial programs, subject to the following conditions:

• I, **Marco Piovanelli**, retain all rights on the library and on the original source code.

• You expressly acknowledge and agree that use of this software is at your sole risk. This software and the related documentation are provided "AS IS" and without warranty of any kind, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Under no circumstances including negligence, shall I be liable for any incidental, special or consequential damages that result from the use or inability to use the software or related documentation, even if advised of the possibility of such damages.

• You give me credit in your program's about box and/or in some other suitable place (something like "WASTE text engine © 1993-1995 Marco Piovanelli" would be OK).

• You notify me that you're using my code. This allows me to send updates to "registered" users and to compile a list of users of WASTE.

• You send me a complimentary copy of the finished product, either electronically or by postal mail. For shareware and commercial programs, this means that I get a registered copy. This item does not apply to in-house applications.

The WASTE 1.1 package can be freely distributed in electronic form on computer networks. It cannot, however, be distributed by other means (such as in printed form, on magnetic media or on CD-ROM) without permission from the author. Special permission is granted to the following companies for including the WASTE 1.1 package in their CD collections:

Pacific HiTech, Inc.	Info-Mac CD-ROM
Celestin Company	Apprentice

The latest version of the WASTE package can be downloaded from:

`<URL:ftp://ghost.dsi.unimi.it/pub2/papers/piovanel/>`

There's even an informal **mailing list** dedicated to all developers working with WASTE. To join the list, send a request to `<URL:mailto:waste-request@umich.edu>`.

Acknowledgements

I'd like to thank the following people for their help and inspiration:

• **Dan Crevier**, who maintains the C version of WASTE, without which WASTE could never become popular. Dan also wrote a set of wrapper classes for the THINK Class Library, gave me many suggestions and pointed out a number of bugs.
• **Mark Alldritt**, who used WASTE in his cool Script Debugger application and showed me a way to implement tabs.
• **Greg Galanos** and all the good people at Metrowerks for the free copy of CodeWarrior. They're a great company, aren't they?
• **Matthew Xavier Mora**, for nominating WASTE for the Mac Programming Award in the freeware category.
• **Michael F. Kamprath**, who made a lot of questions and pointed several weaknesses.
• **Mark Lanett**, for a lengthy e-mail exchange about embedded objects which inspired the current implementation.
• **Alan Steremberg**, for more chat about embedded objects and for maintaining the WASTE Mailing List.
• **Matsubayashi Kohji**, for his careful explanation of the Japanese way of using a Macintosh and for testing WASTE with KanjiTalk.
• **Leonard Rosenthol**, who suggested several improvements.
• **Rick Giles**, early adopter of WASTE.
• **Ari Halberstadt**, for his insightful comments and suggestions.
• **René G.A. Ros**, for all his generous aid during the past few years.
• **Paul Celestin**, for the complimentary copy of the Apprentice CD.
• **Adrian Le Hanne**, for indirectly suggesting the signature of the WASTE Demo (this is cryptic, I know).
• **Steven Stapleton** and **Andrew M. McKenzie**, for their beautiful music.

Hardware and software used to develop WASTE:

• Macintosh IIsi 5/40
• THINK Pascal 4.0.2
• THINK C 4.0 (for the 68K assembly portions only)
• Metrowerks Pascal 1.0
• Metrowerks C/C++ 1.2
• MacsBug 6.5dx
• ResEdit 2.1.x
• Swatch 1.2.2
• Even Better Bus Error
• BBEdit Lite 3.0
• Tex-Edit 2.5
• Style 1.3