Chapter 25
Text Edit

The GS TextEdit package has been designed to provide text editing capabilities for any application. It can be used as a complete text editor, as the core of a word processor, or just to allow entering text in a dialog box. TextEdit has been designed around three main criterion:

- Ease of use by the application programmer. Only a few routines need to be called to use a "plain vanilla" text edit box and no extra support routines need to be written.
- Speed. All operations must happen with no or minimal delays. The code will handle special cases separately in order to do this.
- Expansibility. TextEdit is able to be expanded without rewriting the entire package. It is possible to add your own custom objects or to do text operations your own way with "hook routines"

GS TextEdit has been loosely based on the Macintosh version, but a number of enhancements have been added. A feature list follows.

FEATURES

- TextEdit can edit any amount of text that will fit in memory. There are actually several internal limits on the total number of characters in the text and the total number of lines, but these limits range from about 128 million characters on upwards so they should never be reached.

- Mouse activity is translated into text selection:
  - Single clicking moves the cursor to that spot and dragging selects a block of text.
  - Double clicking selects a word and subsequent dragging moves by words.
  - Triple clicking selects a line of text and subsequent dragging moves by lines.

- The text is word wrapped, that is when a line becomes too long it is automatically wrapped to the next line. Note that these are "soft" line breaks; no carriage returns are inserted.

- Cutting, Copying and Pasting of text is supported in the standard manner. These operations transfer the text directly to and from the scrap manager and they can operate on as much text as the scrap manager can handle.

- Intelligent cut and paste is optionally supported. This is a way of adjusting cut and paste operations so as to support word selections more fully. The rules for this adjustment are as follows: When a cut operation is done, remove all spaces to the left of the selection. If there are no spaces to the left of the selection, then remove all spaces to the right of the selection. When a paste operation is done, if the character either to the left or the right of the current selection is part of a word, then insert a space first and then do the paste. This allows the user to double-click a word, cut it and then paste it somewhere else without having to remove or add extra spaces.

- Certain "control" keys perform text manipulation:
  - DELETE & CTRL-D remove the current selection if there is one or if there isn't they remove the character before the cursor.
  - CTRL-F removes the current selection or the character in front of the cursor.
  - CTRL-Y removes all the characters until the end of a line even if there is a selection. It does not remove the last CR in the line (if there is one).
  - CLEAR removes the current selection if there is one and does nothing if there isn't.
  - CTRL-X cuts the current selection. If there is no selection this will have no effect on the current clipboard. (Note: TextEdit can tell the difference between CTRL-X and CLEAR; it looks to see if the CONTROL key is down to tell if CTRL-X was pressed instead of CLEAR.
  - CTRL-C copies the current selection. If there is no selection this will have no effect on the current clipboard.
  - CTRL-V replaces the current selection by whatever was last cut or copied. If there is no selection, the current clipboard (whatever was last cut or copied) is inserted at the current selection.

- Cursor movement keys (the arrow keys) are fully supported with the following enhancements:
  - Holding down the COMMAND key will move by words or by pages.
  - Holding down the OPTION key will move to the beginning and end of the current line or the top and bottom of the entire document. OPTION will override COMMAND if they are both held down.
  - Holding down the SHIFT key will extend the current selection. If the COMMAND key is also held down it will extend by words. If the OPTION key is held down the selection will be extended to the beginning/end of the current line or the top/bottom of the entire document.

- The text can contain any number of stylistic variations put anywhere in the text. Stylistic variations are changes in Font, Style, Size, or Color.

- The text can contain a "Ruler" that specifies formatting information such as: left margin, left indent, right margin, justification, and tab stops.

- Four types of justification are supported: left, center, right, and full. When tab stops are being used, the document justification applies only to the last field in the line. All fields before the last one are left justified.

- Tabs are automatically supported in three separate ways:
  - No tabs. This is the fastest and takes up the least amount of space.
  - Tab stops are put evenly every *X* number of pixels. This could be used to implement MPW style tabs.
  - Tab stops move to particular pixel increments (MacWrite style tabs). Currently only left justified tab stops are supported.

- The maximum amount of text that can be entered can be limited to a particular number of lines, or a particular number of characters, or what will fit in a particular rectangle.

- Primitive support for automatic pagination is built in to TextEdit. If the application indicates that it wants page breaks kept track of, TextEdit will automatically keep track of pages and display the whitespace between pages. TextEdit will also allow the user to insert and remove explicit page breaks from the text.

- Text can be vertically scrolled in a window.

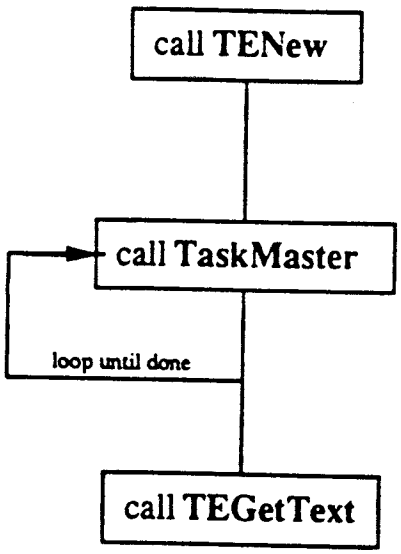
- Text is automatically scrolled to the current selection when the user types and when text is being selected with the mouse, dragging outside the text box will automatically scroll the text in that direction. T' arrow keys will also scroll to the current selection as will cut and paste. Note that copy operations w not scroll to the current selection.

- Text selection, insertion, deletion, and scrolling must be very fast.

GENERAL OVERVIEW

GS TextEdit is implemented as a combination of a super control and a toolset. The reason for this is that as a control TextEdit shouldn't require very many toolbox calls (the control manager will do a lot of the housekeeping) and as a toolset, people will be able to call TextEdit in a familiar manner (they won't have to learn how to call a custom defproc). TextEdit can also be told not to create a control for any given record. This allows a TextEdit record to be used in a port that is not a window, but it makes life more difficult for the application programmer so it should only be used sparingly.

GS TextEdit contains calls to setup and dispose of text boxes (places to edit text), extract text from them, process simple events (mouseDown, keyDown, etc) and to support complex events (menu selections). If the application is using *TaskMaster*, things get even simpler; *TaskMaster* automatically passes most events directly to TextEdit and the application only has to allocate and dispose of the record. The following is a simple flowchart of what an application needs to do to have a TextEdit record using *TaskMaster*:



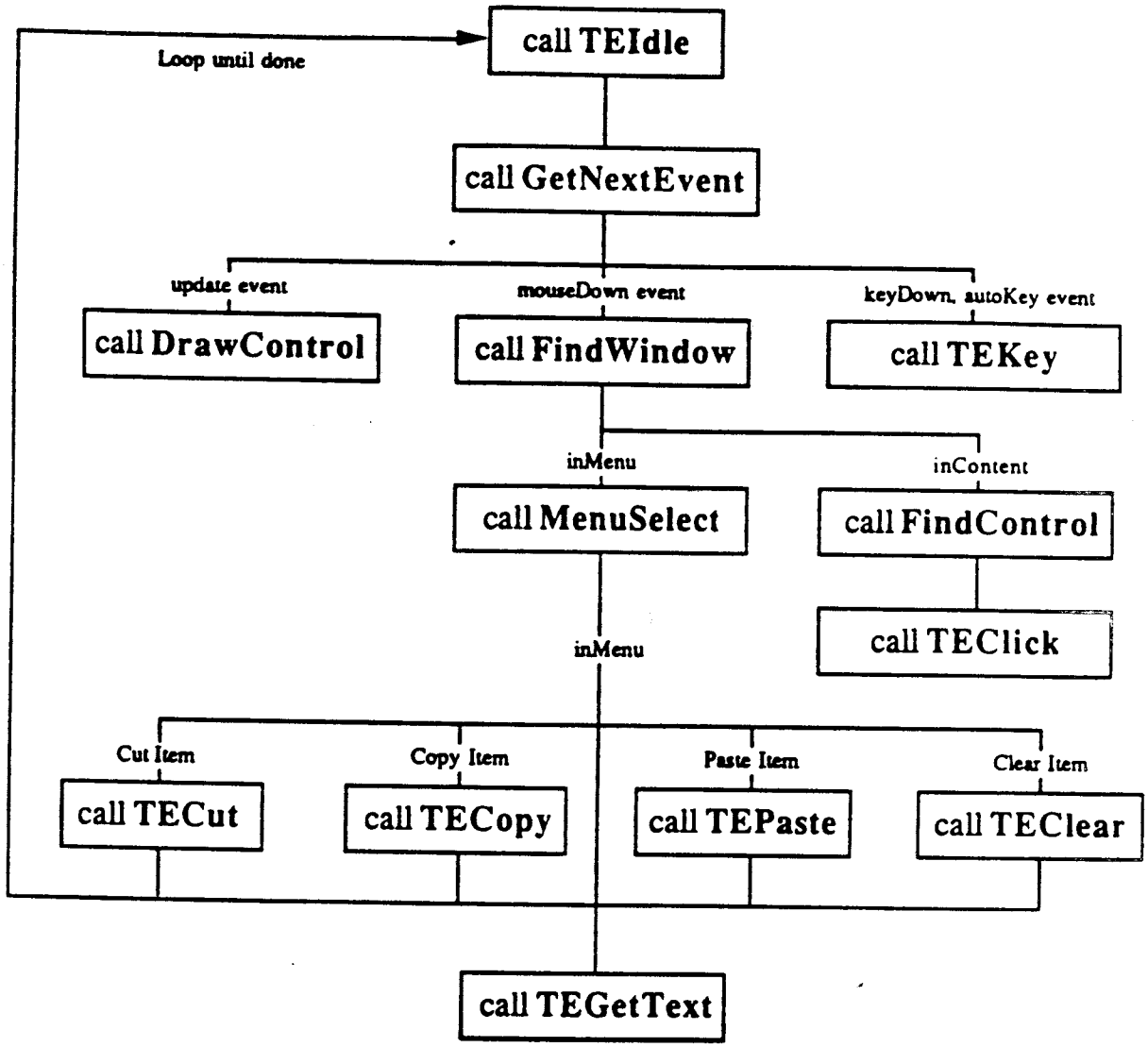
TENew should be called after the window is created to allocate a new TextEdit record.

TaskMaster handles all the events; it inserts all the keys that the user types, handles all the mouse activity and takes care of blinking the cursor. It even calls *TECut*, *TECopy*, *TEPaste* and *TEClear* for the TextEdit record.

TEGetText should be called before the window is disposed of to save the text and style information the user has typed.

Note that *TEKill* need not be called, since closing the window will dispose of all the controls in the window including the TextEdit record. This means that for a simple, single-style TextEdit record only three calls are needed ! If the application wants the user to be able to use multiple styles, it must provide a way for the user to choose the font/style and it must provide some way of displaying what the current style is. To do this it will need to make two more calls (*TEGetSelectionStyle* and *TEStyleChange*).

If the application is not using task master it needs to make the following calls in place of it:



If the application creates a TextEdit record that is not a control, it will need to do even more. First, it must call *TEUpdate* for every record in the port to be redrawn, instead of *DrawControls*. Second, *TEActivate* and *TEDeavtivate* must be called when the application switches between windows (or between TextEdit records). Finally, when the application receives a mouseDown event, it has to determine which TextEdit record was clicked in on it's own; *FindControl* will not work. Note that all the TextEdit routines that take and return control handles work the same way; the handle has exactly the same structure as when a control record has been allocated. The only difference is that these record handles cannot be passed to the control manager for anything.

Internals

Internally TextEdit uses background processing and allocates extra memory to improve response time. TextEdit's basic philosophy is that memory is cheap, but speed is critical. This doesn't mean that TextEdit won't work in limited memory situations; it does mean that TextEdit will be much faster if it has more memory. Of course TextEdit does require a certain amount of free memory to operate. This is currently about 8K. Also TextEdit stores other information in addition to the text itself. This overhead is currently approximately 20% of the text, so for example, if an application wanted to put 20K of text into a TextEdit record there would have to be about 20K + 4K + 8K (32K) of free memory.

The text in a TextEdit record is not stored in a linear fashion and should be treated as a "black box" — not directly accessed. TextEdit routines must be used to extract text from a record and to copy text into a record. Formatting information is stored separately from the text and it also cannot be directly accessed. To access either the text or style of a TextEdit record, the user must call *TEGetText*; this will return the text in one contiguous block and the style information in a *TEStyleRecord* record.

The scroll bars for a TextEdit record are separate controls that are loosely tied to the main control. TextEdit replaces the actionProcPtr field of the scroll bar with a pointer to it's own scrolling routine.

The "extended" control record that TextEdit uses is a super-set of the super control record as defined in Chapter 4 of the Universe Toolbox Update. This means that TextEdit records can be created with *NewControl2*.

HIGH LEVEL DATA STRUCTURES

struct TERuler

```
{
```

word	leftMargin;	the number of pixels to indent from the left edge of the text rectangle for all lines except the start of a paragraph.
word	leftIndent;	the number of pixels to indent from the left edge of the text rectangle for the start of a paragraph.
word	rightMargin;	the right boundary of the text measured in pixels from the left edge of the text rectangle. (used for word wrapping and justification)
word	just;	This parameter has four states: 0 (left), 1 (right), 2 (center), and 3 (full). Left and right justified text starts from the left and right margins respectively. Center justification centers the text between the left and right margins. Full justification adds extra pixels to every space character to make the text flush with both the right and the left margins.
word	extraLS;	the number of pixels to add between the lines of text. This can even be a negative number, but if it is the lines may overlap.
word	flags;	set of bit flags — see the following flag documentation
long	userData;	This will be used for future expansion
word	tabType;	This tells what type of tab data follows. 0=no tabs, 1=every X number of pixels, and 2=absolute pixel positions. The next two fields are only there is **tabType** = 2. In all other cases they are omitted.
TabItem	theTabs[];	this is an array of tabItems, one for each tab stop
word	tabTerminator;	this parameter depends on **tabType**. If **tabType** = 0, this is not even here (the structure ends with tabType). If **tabType** = 1, this is the number of pixels that each tab will move to a multiple of. If **tabType** =2, this is $FFFF to terminate the array.

flags:

bit #	name	if SET
bit 15:	fEqualLineSpacing	Equal line spacing. This uses the maximum line spacing for every line in the block. ***This must be clear in v1.0.***
bit 14:	fShowInvisibles	Show invisible characters. (Space, Tab, Return). ***This must be clear in v1.0.***
bit 13 - 0:	-----	Reserved.

```
}
```

struct TabItem

```
{
```

word	tabKind;	this is the kind of tab stop (left, center, right, and decimal)
word	tabData;	this is the pixel offset (from the left of the text rectangle) to move to

```
}
```

struct TEStyle

```
{
    long    fontID;     the fontID for the text in this style
    word    foreColor;  the foreground color for the text
    word    backColor;  the background color for the text
    long    userData;   Reserved for future expansion
}
```

struct StyleItem

```
{
    long    length;     this is the number of characters in the text that are in the following style.
    long    offset;     this is an offset into the style list that specifies the actual style.
}
```

struct TEStyleRecord

```
{
    word        version;            this is the style record version number. It is currently $0000.
    long        rulerListLength;    this is the length of the following ruler in bytes.
    TERuler     theRulerList[];     this is the initial ruler for the text. Note: this is not a pointer; the data
                                    for the ruler is embedded in the StyleRecord
    long        styleListLength;    this is the length of all of the following styles in bytes.
    TEStyle     theStyleList[];     this is a list of all the unique styles in the text. It is not a pointer,
                                    these are the actual styles embedded in the record. These styles
                                    should all be unique — do NOT duplicate styles.
    long        numberOfStyles;     this is the number of style items that follow. Each style item consists
                                    of a length and a style offset.
    StyleItem   theStyles[];        this is an array of all the style items.
}
```

Explanation of fields in a TEStyleRecord:

This record allows an application to set the format and style of any block of text that it is going to insert into a TERecord. The style record is fairly complicated and TextEdit is not tolerant of mistakes in the record, so be careful. These are the important things to know about style records:

Lets start with the *version* field. It tells TextEdit what version of the style record to expect. Future versions of TextEdit may have a completely different style record format, but they will still be able to accept older style records since they can tell the difference by the *version* field.

The *rulerListLength* field tells TextEdit how many bytes will follow for the ruler list. This length does not include itself; do not add another four bytes for the *rulerListLength* field! This ruler list is copied directly into the TERecord that it pertains to and used as the initial ruler. This version of TextEdit will only look at the first ruler in the ruler list, but more than one ruler may be specified; the rest will just be ignored. Also note that this version of TextEdit ignores the passed ruler information in every call except for *TENew* and *TESetText*. This means that the application can specify an initial ruler for the entire record, but it can't insert or paste a new one into the record.

The *rulerList* field is the actual data for the initial ruler. It is in the format of a TERuler and is not a fixed size. Note that the StyleRecord description allows for more than one ruler to be passed but for this version of TextEdit, only the first one is used. There is one disadvantage of passing more than one ruler: memory is allocated for ALL the passed rulers!

The *styleListLength* field tells TextEdit how many bytes will follow for the style list. The style list con. of all the unique styles in the TERecord. Note that this field is the total number of bytes for all of the following styles and it does not include itself.

The *styleList* field is a list of all the unique styles in the TERecord. Each style is a TEStyle record (12 bytes long) and each style must be different than all the others. If the application wants to use the same style twice, it just makes the two style items point to the same unique style in this list.

The *numberOfStyles* field is the number of style items that follow; it is not the number of bytes that these items take up.

The *theStyles* array consists of style items. Each style item first tells the number of characters in the text that will be in the specified style and then it supplies an offset into the style list that specifies the actual style. Note that this offset is an offset into the styleList field, NOT into the entire StyleRecord. This means that $0000 0000 is the first style in the style list. Also note that this is a byte offset; to get the second style in the style list the offset would be $0000 000C, not two. The length field of the item does not specify position information; it is purely relative. This means that you must be very careful how you specify the length.

struct TEColorTable

```
{
```

word	contentColor;	color for the inside of the entire boundsRect.
word	outlineColor;	color for the box drawn around the record.
word	pageBoundaryColor;	color for the boundaries of pages, that is the space between the bottom of the footer of one page and the top of the header of the next.
word	hilightForeColor;	foreground color for the hilited text to be drawn in.
word	hilightBackColor;	background color for the hilited text to be drawn in. This is the color used for the caret.
word	vertColorDescripter	bits 2 and 3 specify the color table reference descripter, exactly like the *moreFlags* field of the scroll bar template.
long	vertColorRef	reference to the color table for the vertical scroll bar.
word	horzColorDescripter	bits 2 and 3 specify the color table reference descripter, exactly like the *moreFlags* field of the scroll bar template.
long	horzColorRef	reference to the color table for the vertical scroll bar.

```
}
```

All of the bits in all of the color words in the TEColorTable are significant. The way a color pattern is formed is by taking the color word and copying it sixteen times to form a pattern.

struct TEParamBlock
{

word	pCount;	this is the number of parameters that follow. The minimum count is 7 and the current maximum is 25.
long	ID;	this is the control ID that TaskMaster will use.
Rect	boundsRect;	this rectangle will contain the entire TextEdit control including the scroll bars and any outline drawn around the text.
long	procRef;	this must be $85000000.
word	flags;	see below.
word	moreFlags;	see below.
long	refCon;	for application use.
long	textFlags;	see below.
Rect	indentRect	each coordinate of this rectangle specifies the amount of white space to leave between that edge of the *boundsRect* and the text itself. To use the default value for a particular coordinate use $FFFF. The default values are (2,6,2,4) in 640 mode and (2,4,2,2) in 320 mode
Handle	vertBar;	this is a handle to the vertical scroll bar to use. NULL => the record doesn't have a vertical scroll bar. $FFFFFFFF => create a scroll bar just inside the right edge of the *boundsRect*.
word	vertAmount;	this is the number of pixels to scroll whenever the up or down arrow on the vertical scroll bar is pressed. $0000 => default of 9 pixels.
Handle	horzBar;	reserved. This currently must be NULL.
word	horzAmount;	reserved. This currently must be $0000.
long	styleRef;	this is a reference to the initial style information. NULL => use the default style and ruler information. Bits 0 and 1 of the *moreFlags* field are used to specify how this is used.
word	textDescripter;	this is an input text descripter that specifies how the initial text is referenced and what format it is in.
long	textRef;	this is the reference to the initial text. If this is NULL, then there is no initial text.
long	textLength;	this is the initial text length for all descripters that require one.
long	maxChars;	this is the maximum number of characters allowed in the text. NULL => no limit on characters.
long	maxLines;	this is the maximum number of lines allowed in the text. NULL => no limit on lines. ***This must be NULL in v1.0***
word	maxHeight;	this is the maximum height of the document in pixels. NULL => no limit on the total height of the text. Note: all these *max* fields are cumulative; they will all be respected. ***This must be NULL.***
word	pageHeight;	this is the height of a page in pixels. If this parameter is $0000, then page breaks will not be kept track of and the user will not be allowed to insert explicit page breaks.
word	headerHeight;	this is the number of pixels to allocate from the top of the page for the header. This must be less than (*pageHeight - footerHeight*).
word	footerHeight;	this is the number of pixels to allocate from the bottom of the page for the footer. This must be less than (*pageHeight - headerHeight*).
word	pageBoundary;	this is the number of pixels between the footer of one page and th header of the next. It is used to visibly indicate page breaks. $FFI means use the default value of 4 pixels.

long	colorRef;	see the description of the colorTable record. Bits 2 and 3 of the *moreFlags* field are used to specify the way the color table is referenced.
word	drawMode	this is the text mode that quickdraw uses to draw the text in. The default mode is $0000 (modeCopy).

flags:

bit #	name	if SET
bit 15 - 8:	-----	Reserved. Must be CLEAR.
bit 7:	ctlInvis	The control will be invisible.
bit 6 - 0:	-----	Reserved. Must be CLEAR.

moreFlags:

bit #	name	if SET
bit 15:	fCtlActive	The control is active. Must be CLEAR to start with.
bit 14:	fCtlCanBeActive	Must be SET.
bit 13:	fCtlWantsEvents	Must be SET.
bit 12:	fCtlProcRefNotPtr	Must be SET.
bit 11:	fTellAboutGrow	The record will resize itself when the window changes size.
bit 10 - 4:	-----	Reserved. Must be CLEAR.
bit 3 - 2:	colorDescripter	00 = ptr, 01 = handle, 10 = resource, 11 is invalid.
bit 1 - 0:	styleDescripter	00 = ptr, 01 = handle, 10 = resource, 11 is invalid.

textFlags:

bit #	name	if SET
bit 31:	fNotControl	Don't allocate a custom control for the TextEdit record.
bit 30:	fSingleFormat	Allow only one ruler in the text. (Currently this bit is ignored)
bit 29:	fSingleStyle	Allow only one style in the text.
bit 28:	fNoWordWrap	Do not word wrap the text; only break lines on CR's.
bit 27:	fNoScroll	Do not allow any manual or auto scrolling.
bit 26:	fReadOnly	Do not allow editing of the text. Note that copy operations may still be performed.
bit 25:	fSmartCutPaste	Intelligent cut and paste will be supported as described in the initial feature list for GS TextEdit.
bit 24:	fTabSwitch	When the user types a TAB, switch to the next control in the control list that can be activated. (see the initial feature list for more details)
bit 23:	fDrawBounds	Draw a box around the TextEdit control, just inside the *boundsRect*. The pen size for the box is $(2,1)_{(h,v)}$.
bit 22:	fColorHilight	Use the color table to do color hilighting. This is slower than the default of just inverting the hilighted text. (Currently this bit must be CLEAR)
bit 21 - 0:	-----	Reserved. Must be CLEAR.

}

REFERENCE and TEXT DESCRIPTERS

Reference descripters are parameters that describe how other parameters are referenced. For instance, a certain routine may take a *styleDescripter* and a *styleRef* as parameters. The *styleDescripter* would tell the routine whether the *styleRef* was a pointer to the data, a handle to the data, or a resource ID to the data. Descripters can also be used to tell a routine what kind of data to expect; the descripter *dataIsPString* tells a routine to expect the data to be a pascal string (a length byte followed by the string itself).

TextEdit uses two different types of descripters. The first type are the **Reference Descripters**, which are used for passing and returning style information. These descripters indicate whether the data is referenced by a pointer, a handle, or a resource ID.

The second type of descripters, **Text Descripters**, are used for passing and returning text from a routine. Text descripters have two parts: the lower three bits (0-2) specify what kind of data is being passed or returned and the next two bits (3-4) are a reference descripter to the data. Text descripters modify two other parameters, the reference parameter and the length parameter, but in some cases the length parameter and even the reference parameter are ignored.

When text descripters are used for passing text to a routine, the length parameter is significant only for *dataIsTextBox2* and *dataIsTextBlock* descripters; it is ignored in all other cases. Also the *textIsNewHandle* field is invalid when the descripter is describing **input** parameters (passing text to a routine). When the descripter is being used to describe **output** parameters (getting text back from a routine), the length parameter is the size of the buffer that is to be filled with the text. Note that this is the total size of the buffer; it must include space for any length words or terminating zero bytes.

Reference Descripters:

Descripter	Name	What it means
$0000	refIsPointer	The reference parameter points to the block of data. The length must either be fixed or it must somehow be specified by the block of data.
$0001	refIsHandle	The reference parameter is a handle to the block of data.
$0002	refIsResource	The reference parameter is a resource ID which can be used to get the block of data.
$0003	refIsNewHandle	A new handle will be created to store the data; the reference parameter is a pointer to a four byte buffer to store this new handle. This is only valid for outputs.

Text Descripters:

Bits 0-2	Name	What it means
000	dataIsPString	The data starts with a length byte and is followed by the text itself.
001	dataIsCString	The data starts with the text and is followed by a zero byte.
010	dataIsC1Input	The data is in the same format as a GS/OS class one input string. The buffer will start with a length word and will be followed by the text itself.
011	dataIsC1Output	The data is in the same format as a GS/OS class one output string. The buffer will start with a buffer length word, followed by the text length word, and will end with the text itself.
100	dataIsTextBox2	The data is in the same format as what LineEdit's TextBox2 call takes. Note: if this format is used, the embeded style information will override the style information passed in the *styleRef* parameter. The length parameter contains the length of the text.
101	dataIsTextBlock	The data is just raw text; the length of the data is passed in the length parameter.
110	------	**invalid.**
111	------	**invalid.**

Bits 3-4	Name	What it means
00	textIsPointer	The reference parameter points to the text. This is valid for both inputs and outputs.
01	textIsHandle	The reference parameter is a handle to the text. This is valid for both inputs and outputs.
10	textIsResource	The reference parameter is a resource ID to the text. This is valid for both inputs and outputs.
11	textIsNewHandle	A new handle will be created to store the text; the reference parameter is a pointer to a four byte buffer to store this new handle. This is only valid for outputs.

HIGH LEVEL CALLS

The calls in bold do not need to be used if you are using custom controls and TaskMaster.

Call name	Description
TEBootInit	Required for every toolset.
TEStartup	Must be called before any other TextEdit call.
TEShutdown	Must be called when an application quits.
TEVersion	Returns the version number of TextEdit.
TEReset	Required for every toolset.
TEStatus	Returns the status of TextEdit.
TENew	Allocates a new TextEdit record.
TEKill	**Disposes of a TextEdit record.**
TESetText	Sets all the text in a preexisting TextEdit record to the passed text.
TEGetText	Returns the text in the specified TextEdit record.
TEGetTextInfo	Returns information on the specified TextEdit record.
TEIdle	**Called periodically to blink the cursor & do background tasks.**
TEActivate	**Activates the selection.**
TEDeactivate	**Makes the selection inactive**
TEClick	**Activates a TextEdit record, then selects text in it.**
TEUpdate	**Redraws a TextEdit record.**
TEPaintText	Paints the text of a TextEdit record into an offscreen port. Used for printing.
TEKey	**Inserts a character into the active TextEdit record.**
TEInsertPageBreak	Inserts a page break into the text.
TECut	**Cuts the current selection into the desk scrap.**
TECopy	**Copies the current selection into the desk scrap.**
TEPaste	**Pastes the contents of the desk scrap into the text.**
TEClear	**Clears the current selection.**
TEInsert	Inserts the passed text just before the current selection.
TEReplace	Replaces the current selection with the passed text.
TEGetSelection	Returns the starting and ending offset of the current selection.
TESetSelection	Sets the selection to the starting and ending offset passed.
TEGetSelectionStyle	Returns style information for the current selection.
TEStyleChange	Changes the style of the current selection.
TEGetHooks	Returns a record that contains pointers to the low level hook routines.
TESetHooks	Sets the low level hook routines to the passed pointers.
TEGetDefProc	Returns a pointer to TextEdit's custom defproc.

$0122 TEBootInit

This call does nothing.

Parameters: None.

Errors: None.

$0222 TEStartup

This call starts up the toolset and must be called by every application that uses TextEdit. It allocates space for all the TextEdit global variables and sets up TextEdit's direct page.

Parameters:

Stack before call

previous contents	
userID	**Word** - The user ID of the application.
directPage	**Word** - TextEdit needs one page of direct page.
	<— SP

Stack after call

previous contents	
	<— SP

Errors: $2201 - teAlreadyStarted. TextEdit is already started.

$0322 TEShutdown

This call must be called by every application that uses TextEdit when the application is completely through using it. This call deallocates all the memory that TextEdit itself allocated. Note that the application must dispose of each TextEdit record in turn; this call only deallocates TextEdit's global variables.

Parameters: None

Errors: $2202 - teNotStarted. TextEdit was never started.

$0422 TEVersion

This call returns the version number of TextEdit. This works whether or not TextEdit is active, although it always must be loaded.

Parameters:

Stack before call

previous contents	
wordspace	**Word** - Space for result.
	<— **SP**

Stack after call

previous contents	
versionInfo	**Word** - Version number of TextEdit.
	<— **SP**

Errors: None

$0522 TEReset

This call is made by the system when CTRL-RESET is pressed. Currently, it does nothing.

Parameters: None

Errors: None

$0622 TEStatus

This call returns the status of TextEdit.

Parameters:

Stack before call

previous contents	
wordspace	**Word** - Space for result.
	<— **SP**

Stack after call

previous contents	
activeFlag	**Word** - $FFFF if TextEdit is active, $0000 if not active
	<— **SP**

Errors: None

$0922 TENew

This call allocates a new TextEdit record in the current port. The parameter block that it takes is exactly the same one that would be passed to *NewControl2*. The only difference is that this call creates one and only one TextEdit record; *NewControl2* can create a entire list of controls. Unless you are not using controls, you should probably call *NewControl2* to allocate all your controls at once.

Parameters:

Stack before call

previous contents	
longspace	**Long** - Space for result.
parameterBlock	**Long** - Pointer to parameter block. See the Data Structures section.
	<— SP

Stack after call

previous contents	
teHandle	**Long** - Handle to the new TextEdit record.
	<— SP

Errors:

$2202 - teNotStarted. TextEdit was never started.
$2204 - teInvalidDescripter. The descripter in the parameter block was invalid.
$2205 - teInvalidFlag. The flag word in the parameter block was invalid.
$2206 - teInvalidPCount. The parameter count was invalid.
$2207 - teInvalidRect. The view rect must be at least twenty pixels wide.
$02xx - Memory manager errors are propagated.

$0A22 TEKill

This call deallocates the passed TextEdit record and all associated memory but it does not erase or invalidate the screen; the application needs to do that for itself. Make this call only when you are completely through with the record; all the text in it will be lost after this call. If the active record is killed, then no record will be left active; it is the application's responsibility to activate any other record if it wants to.

If the application is using TextEdit controls, it doesn't need to make this call; calling *KillControls* or *DisposeControl* will do the same thing.

Parameters:

Stack before call

previous contents	
teHandle	**Long** - Handle to the TextEdit record to dispose of.
	<— **SP**

Stack after call

previous contents	
	<— **SP**

Errors:

$2202 - teNotStarted. TextEdit was never started.
$2203 - teInvalidHandle. The record handle was not a valid TERecord.

$0B22 TESetText

This call replaces the text in a TextEdit record with the passed text. It updates all the internal information and if the record is a TextEdit control, it invalidates the entire text rectangle (the next update event will redraw it). Otherwise it will redraw the entire text rectangle. This routine accepts text in a large variety of formats; which format you are using depends on the *textDescripter* parameter (see the section on reference and text descripters).

Parameters:

Stack before call

previous contents	
teHandle	**Long** - Handle to the TextEdit record to change.
textDescripter	**Word** - The input text descripter that tells what form the passed text is in.
textRef	**Long** - Reference to the text.
textLength	**Long** - Length of the text (only valid if the text descripter requires a length).
styleDescripter	**Word** - The reference descripter that tells what form the style info is in.
styleRef	**Long** - Reference to the style information to use. if *styleRef* is NULL, then use the first style in the existing record.
	<— SP

Stack after call

previous contents	
	<— SP

Errors:

$2202 - teNotStarted. TextEdit was never started.
$2203 - teInvalidHandle. The record handle was not a valid TERecord.
$2204 - teInvalidDescripter. The descripter was not valid.
$02xx - Memory manager errors are propagated.

$0C22 TEGetText

This call returns the text and the style information from the passed TextEdit record in a wide variety of formats (see the section on reference and text descripters). It returns the total length of the text, even if the buffer is too small to contain it. If there is more text than will fit in the buffer, the buffer is first filled, then a buffer full error is returned. The same thing will happen if the text will not fit in the required format — for example, if there are 300 characters and the programmer asks for a Pascal string.

Parameters:

Stack before call

previous contents	
longspace	**Long** - Space for result.
teHandle	**Long** - Handle to the TextEdit record to extract the text from.
bufferDescripter	**Word** - The output text descripter that tells what form to put the text in.
bufferRef	**Long** - Reference to the buffer.
bufferLength	**Long** - Length of the buffer (or ignored depending on the text descripter).
styleDescripter	**Word** - The reference descripter that tells what form to put the style info into.
styleRef	**Long** - Reference to where the style information is to be stored. if *styleRef* is NULL, then don't return any style information.
	<— SP

Stack after call

previous contents	
textLength	**Long** - Total length of all the text in the record.
	<— SP

Errors:

$2202 - teNotStarted. TextEdit was never started.
$2203 - teInvalidHandle. The record handle was not a valid TERecord.
$2204 - teInvalidDescripter. The descripter was not valid.
$2208 - teBufferOverflow. The buffer was too small.
$220C - teInvalidTextBox2. The TextBox2 format codes were inconsistant.
$02xx - Memory manager errors are propagated.

$0D22 TEGetTextInfo

This call returns information about the passed TextEdit record. It allows the application programmer to control how many parameters (not bytes) the call will return. Currently this call returns five parameters, but future versions of TextEdit may return more.

Parameters:

Stack before call

previous contents	
teHandle	**Long** - Handle to the TextEdit record to get information on.
infoRecPtr	**Long** - Pointer to the buffer for the Information Record.
parameterCount	**Word** - Number of parameters to be returned in the buffer.
	<— SP

Stack after call

previous contents	
	<— SP

Errors:

$2202 - teNotStarted. TextEdit was never started.
$2203 - teInvalidHandle. The handle was not a valid TERecord.
$2206 - teInvalidPCount. The parameter count was too large for this version of TextEdit.

Information Record Description:

These are the parameters that will be returned into your buffer. Future versions of TextEdit may add more parameters to the end of this record, but they will not change the contents or positions of the existing parameters.

Offset	Size	Name	What it means
$0000	long	charCount	The number of characters in the Text Record.
$0004	long	lineCount	The number of lines in the Text Record.
$0008	long	formatMemory	The amount of memory needed to contain the style info.
$000C	long	totalMemory	The total number of bytes used by the current TextEdit record.
$0010	long	styleCount	The number of unique styles in the Text Record.
$0014	long	rulerCount	The number of rulers in the Text Record.

$0E22 TEIdle

This routine should be called as often as possible; usually every time through the main event loop and periodically during time consuming operations. What it does is blink the cursor in the active TextEdit record and run TextEdit background tasks. Note that no matter how many times you call TEIdle, the time between cursor blinks will never be less than the user's control panel setting.

If the active record is a TextEdit control, the application doesn't need to call TEIdle; TaskMaster will take care of it.

Parameters: None

Errors: $2202 - teNotStarted. TextEdit was never started.

$0F22 TEActivate

This call activates the passed TextEdit record; the selection is rehilighted in it's active state and all future editing actions apply to the new record.

If the application is using TextEdit controls, it doesn't need to make this call; TaskMaster will take care of it.

Parameters:

Stack before call

previous contents	
teHandle	**Long** - Handle to the TextEdit record to activate.
	<— SP

Stack after call

previous contents	
	<— SP

Errors:

$2202 - teNotStarted. TextEdit was never started.
$2203 - teInvalidHandle. The record handle was not a valid TERecord.

$1022 TEDeactivate

This call deactivates the passed TextEdit record; the selection is hilighted into its inactive state and future editing operations, such as keys, cut & paste, etc are ignored.

If the application is using TextEdit controls, it doesn't need to make this call; TaskMaster will take care of it.

Parameters:

Stack before call

previous contents	
teHandle	Long - Handle to the TextEdit record to deactivate.
	<— SP

Stack after call

previous contents	
	<— SP

Errors:

$2202 - teNotStarted. TextEdit was never started.
$2203 - teInvalidHandle. The record handle was not a valid TERecord.

$1122 TEClick

This call will first activate the TextEdit record that was passed to it if it's not already active. After this, it will track the mouse selecting all the text that it passes over until the user lets up on the mouse button. If the shift key is held down, the selection will be extended. This call will automatically scroll the text in the appropriate direction if the mouse is dragged outside of the text rectangle. This call will also handle double and triple clicks. In the case of a double click, a word will be selected and dragging will expand (or shorten) the selection by words. In the case of a triple click, the entire line will be selected and the selection will be expanded (or shortened) by lines.

If the application is using TextEdit controls, it doesn't need to make this call; TaskMaster will take care of it.

Parameters:

Stack before call

previous contents	
teHandle	**Long** - Handle to the TextEdit record to that was clicked in.
eventRecordPtr	**Long** - Pointer to the event record for the click.
	<— SP

Stack after call

previous contents	
	<— SP

Errors:

$2202 - teNotStarted. TextEdit was never started.
$2203 - teInvalidHandle. The record handle was not a valid TERecord.

$1222 TEUpdate

This routine redraws the contents of the TextEdit record that was passed to it. Only the part that needs to be redrawn is actually redrawn (this is determined by looking at the visRgn). This call should be made after a BeginUpdate and before a EndUpdate. If the application is using TextEdit controls, it doesn't need to use this call; it should just call DrawControls instead. If not, the application should make this call for every TextEdit record in the window. TextEdit will check to see if it needs to do any drawing at all and if it doesn't it will return very quickly (less than 7 ms).

Parameters:

Stack before call

previous contents	
teHandle	**Long** - Handle to the TextEdit record to be redrawn.
	<— SP

Stack after call

previous contents	
	<— SP

Errors:
$2202 - teNotStarted. TextEdit was never started.
$2203 - teInvalidHandle. The record handle was not a valid TERecord.

$1322 TEPaintText

This routine is for printing a TextEdit record; it draws the contents of the passed record into the specified grafport. If the record supports page breaks (the *pageHeight* field of the TENew parameter block is non zero) the *startingValue* field is the page to print. In this case this routine will draw the page that was specified (clipped to *rectPtr*, of course) and then return the next page number. If the record does not support page breaks, the *startingValue* field is the first line to print. In this case *TEPaintText* will draw all the lines that fit into the rectangle specified by *rectPtr*, and then return the next line to draw. The *flags* field allows the application to skip certain pages or draw lines that get split between pages.

Parameters:

Stack before call

previous contents	
longspace	**Long** - Space for result.
grafPort	**Long** - A pointer to the GrafPort to draw into
teHandle	**Long** - The TEHandle to draw from
startingValue	**Long** - The page number to draw (or the starting line number)
rectPtr	**Long** - pointer to the rectangle to draw into
flags	**Word** - flags. See the following description.
	<— **SP**

Stack after call

previous contents	
nextValue	**Long** - The next page (or line number) to draw
	<— **SP**

flags:

bit #	name	if SET
bit 15:	fPartialLines	Display all the lines that fit in *rectPtr*, even if the last one is clipped.
bit 14:	fDontDraw	Only calculate how many lines there are on this page (don't actually draw anything.

Errors:	$2202 - teNotStarted. TextEdit was never started. $2209 - teInvalidLine. The starting line number exceeded the number of lines in the text. (you are through displaying)

$1422 TEKey

This routine should be called whenever a KeyDown or AutoKey event occurs. The only parameter is a pointer to the event record (this does not have to be a TaskMaster record; a simple event record is all that is needed). This routine will insert the key into the text of the active TextEdit record if it is not a "control key". If it is a "control key", TextEdit will perform the appropriate action as described in the Standard Editing Keys section.

If the application is using TextEdit controls, it doesn't need to make this call; TaskMaster will take care of it.

Parameters:

Stack before call

previous contents	
eventRecordPtr	**Long** - Pointer to the event record for the key.
	<— SP

Stack after call

previous contents	
	<— SP

Errors:

$2202 - teNotStarted. TextEdit was never started.
$02xx - Memory manager errors are propagated.

$2322 TEInsertPageBreak

This routine inserts a page break into the text, just before the start of the current selection. It will not replace the current selection. This call only works on records that have a non zero *pageHeight* field.

Parameters: None

Errors:
$2202 - teNotStarted. TextEdit was never started.
$220A - teInvalidCall. This TextEdit record does not support page breaks.
$02xx - Memory manager errors are propagated.

$1622 TECut

This call copies the current selection to the desk scrap, scrolls to the beginning of it, deletes it and finally redraws (not invalidates!) the screen. In the process, the old desk scrap is destroyed. The style information will also be copied to the desk scrap along with the text. This call acts on the active TextEdit record, but if there is no active record, then it simply does nothing; no error is returned. Similarly, if there is no selection, this call will do nothing; it will NOT destroy the old desk scrap.

If the application is using TextEdit controls, it doesn't need to make this call; TaskMaster will take care of it.

Parameters: None

Errors:
$2202 - teNotStarted. TextEdit was never started.
$02xx - Memory manager errors are propagated.

$1722 TECopy

This call copies the current selection to the desk scrap. In the process, the old desk scrap is destroyed. The style information will also be copied to the desk scrap along with the text. This call acts on the active TextEdit record, but if there is no active record, then it simply does nothing; no error is returned. Similarly, if there is no selection, this call will do nothing; it will NOT destroy the old desk scrap. The text will not be scrolled to the current selection whether or not there was one.

If the application is using TextEdit controls, it doesn't need to make this call; TaskMaster will take care of it.

Parameters: None

Errors:
$2202 - teNotStarted. TextEdit was never started.
$02xx - Memory manager errors are propagated.

$1822 TEPaste

This call replaces the current selection with the contents of the desk scrap, then it redraws (not invalidates!) the screen. If there is style information in the scrap, this will also be pasted in. This call operates on the active TextEdit record. If there is no active record, it will do nothing; it will not return an error. Note: Even if there is nothing in the desk scrap the selection will always be deleted.

If the application is using TextEdit controls, it doesn't need to make this call; TaskMaster will take care of it.

Parameters: None

Errors: $2202 - teNotStarted. TextEdit was never started.
$02xx - Memory manager errors are propagated.

$1922 TEClear

This call removes the current selection in the active TextEdit record, then it redraws (not invalidates!) the screen. If there is no active TextEdit record, this call will do nothing and return no error. If there is no selection, this call will also do nothing and return no error. Note: this call does not affect the desk scrap.

If the application is using TextEdit controls, it doesn't need to make this call; TaskMaster will take care of it.

Parameters: None

Errors: $2202 - teNotStarted. TextEdit was never started.

$1A22 TEInsert

This call inserts the passed text just before the current selection range in the active TextEdit record and then it redraws the text. The current selection is not deleted or even unselected. It accepts the same text and reference descripters that TESetText does — see the section on text and reference descripters. If there is no active TextEdit record, this call will do nothing; it will not return an error. This call does not affect the desk scrap. Note: this call always redraws the text; it never generates an update event.

Parameters:

Stack before call

previous contents		
textDescripter	Word	The input text descripter that tells what form the passed text is in.
textRef	Long	Reference to the text to insert.
textLength	Long	Length of the text to insert (only valid if the text descripter requires a length).
styleDescripter	Word	The reference descripter that tells what form the style info is in.
styleRef	Long	Reference to the style information to use. if *styleRef* is NULL, then use the first style in the selection.
	<— SP	

Stack after call

previous contents	
	<— SP

Errors:

$2202 - teNotStarted. TextEdit was never started.
$220C - teInvalidTextBox2. The TextBox2 format codes were inconsistant.
$02xx - Memory manager errors are propagated.

$1B22 TEReplace

This call replaces the current selection in the active TextEdit record with the passed text and then redraws the text. It accepts the same text and reference descripters that **TESetText** does — see the section on text and reference descripters. If there is no active TextEdit record, this call will do nothing; it will not return an error. This call does not affect the desk scrap. Note: this call always redraws the text; it never generates an update event.

Parameters:

Stack before call

previous contents	
textDescripter	**Word** - The input text descripter that tells what form the passed text is in.
textRef	**Long** - Reference to the new text.
textLength	**Long** - Length of the new text (only valid if the text descripter requires a length).
styleDescripter	**Word** - The reference descripter that tells what form the style info is in.
styleRef	**Long** - Reference to the style information to use. if *styleRef* is NULL, then use the first style in the selection.
	<— SP

Stack after call

previous contents	
	<— SP

Errors:

$2202 - teNotStarted. TextEdit was never started.
$220C - teInvalidTextBox2. The TextBox2 format codes were inconsistant.
$02xx - Memory manager errors are propagated.

$1C22 TEGetSelection

This call returns the starting and ending offsets of the current selection.

Parameters:

Stack before call

previous contents	
teHandle	**Long** - Handle to the TextEdit record to get the selection from
selectionStart	**Long** - Pointer to where to store the starting offset of the selection
selectionEnd	**Long** - Pointer to where to store the ending offset of the selection
	<— **SP**

Stack after call

previous contents	
	<— **SP**

Errors:
$2202 - teNotStarted. TextEdit was never started.
$2203 - teInvalidHandle. The record handle was not a valid TERecord.

$1D22 TESetSelection

This call sets the selection to the passed starting and ending offsets. If the starting offset is greater than the ending offset, they will be swapped. Also if either offset is beyond the end of the text, it will be clipped to the end of the text.

Parameters:

Stack before call

previous contents	
teHandle	**Long** - Handle to the TextEdit record to set the selection.
selectionStart	**Long** - The starting offset for the new selection.
selectionEnd	**Long** - The starting offset for the new selection.
	<— **SP**

Stack after call

previous contents	
	<— **SP**

Errors:

$2202 - teNotStarted. TextEdit was never started.
$2203 - teInvalidHandle. The record handle was not a valid TERecord.

$1E22 TEGetSelectionStyle

This routine returns the all styles contained in the current selection. It also processes them and builds a common style that contains all the style elements that are common to the entire selection. The returned flag word indicates which parts of the common style record are significant. If the bit in the flag word is clear it means that there is no common corresponding style element in the selection.

Parameters:

Stack before call

previous contents		
wordSpace	Word	- Space for result.
commonStylePtr	Long	- Pointer to a TextStyle record. The common style will be returned here.
styleHandle	Long	- Handle to a buffer to fill with style info.
	<— SP	

Stack after call

previous contents		
commonFlag	Word	- This indicates what parts of the common style are significant.
	<— SP	

Errors: $2202 - teNotStarted. TextEdit was never started.

commonFlag: This tells what part of the common style record is significant.

bit #	name	if SET
bit 15 - 6:	-----	Reserved. Must be CLEAR.
bit 5:	fUseFont	The font family part of the fontID in the common style record is significant.
bit 4:	fUseSize	The size part of the fontID in the common style record is significant.
bit 3:	fUseForeColor	The foreground color in the common style record is significant.
bit 2:	fUseBackColor	The background color in the common style record is significant.
bit 1:	fUseUserData	The user data field in the common style record is significant.
bit 0:	fUseAttributes	The attributes part of the fontID of the common style record is significant.

Buffer Description:

```
word        count;      /* this is the number of styles in the current selection */
TextStyle   style1;     /* the first style record */
. . .       . . .
TextStyle   stylen      /* the last style record */
```

$1F22 TEStyleChange

This routine changes the style of the current selection in the active TextEdit record. If there is no active record, this call does nothing and returns no error. If there is an active record, but no selection then this call acts on the "null style record", which is the style that is used for future insertions. Note that the "null style record" gets reset every time the selection changes. This call has a flag word that tells what parts of the passed style record are significant and how to use them.

Parameters:

Stack before call

previous contents	
flags	Word - This indicates what parts of the style record are significant.
stylePtr	Long - Pointer to a TextStyle record. see the Data Structures section.
	<— SP

Stack after call

previous contents	
	<— SP

Errors:

$2202 - teNotStarted. TextEdit was never started.
$2205 - teInvalidFlag. The flag word that was passed is invalid.

Flags:

bit #	name	if SET
bit 15 - 7:	-----	Reserved. must be CLEAR.
bit 6:	fReplaceFont	replace the font of all the styles in the current selection.
bit 5:	fReplaceSize	replace the size of all the styles in the current selection.
bit 4:	fReplaceForeColor	replace the foreground color of all the styles in the current selection.
bit 3:	fReplaceBackColor	replace the background color of all the styles in the current selection.
bit 2:	fReplaceUserField	replace the user field of all the styles in the current selection.
bit 1:	fReplaceAttributes	replace the font attributes of all the styles in the current selection.
bit 0:	fSwitchAttributes	If the entire selection contains the specified attributes, they will be removed from the selection. If only part of the selection contains the specified attributes, they will be added to the entire selection.

Note that bits 0 and 1 are mutually exclusive; if both are set, the teInvalidFlag error will be returned.

$2022 TEGetHooks

This routine returns a record of hook routines that TextEdit uses to do low level operations. See the list of hook routines for a description of the buffer format as well as a description of each hook routine.

Parameters:

Stack before call

previous contents	
teHandle	**Long** - Handle to the TextEdit record to get the hooks from
bufferPtr	**Long** - Pointer to a buffer to fill with hook routines.
count	**Word** - Number of hook routines the application wants pointers for.
	<— SP

Stack after call

previous contents	
	<— SP

Errors:

$2202 - teNotStarted. TextEdit was never started.
$2203 - teInvalidHandle. The record handle was not a valid TERecord.
$2206 - teInvalidPCount. The application asked for too many hook pointers.

$2122 TESetHooks

This routine sets the hook routines that TextEdit uses to do low level operations. See the list of hook routines for a description of the buffer format as well as a description of each hook routine.

Parameters:

Stack before call

previous contents	
teHandle	**Long** - Handle to the TextEdit record to set the hooks for.
bufferPtr	**Long** - Pointer to a buffer that contains pointers to the hook routines
count	**Word** - Number of hook routines the application is setting.
	<— **SP**

Stack after call

previous contents	
	<— **SP**

Errors:

$2202 - teNotStarted. TextEdit was never started.
$2203 - teInvalidHandle. The record handle was not a valid TERecord.
$2206 - teInvalidPCount. The application is setting too many hook pointers.

$2222 TEGetDefProc

This routine returns a pointer to TextEdit's custom defproc. It is normally called only by the Control Manager.

Parameters:

Stack before call

previous contents	
longspace	**Long** - Space for result.
	<— **SP**

Stack after call

previous contents	
defProcPtr	**Long** - Pointer to the defProc
	<— **SP**

Errors: $2202 - teNotStarted. TextEdit was never started.

ERROR CODES:

Number	Name	Description
$2201	teAlreadyStarted	TextEdit has already been started.
$2202	teNotStarted	TextEdit has not been started yet.
$2203	teInvalidHandle	The TERecord handle was not a handle to a valid TextEdit record.
$2204	teInvalidDescripter	The descripter that was passed is not supported by this call.
$2205	teInvalidFlag	The flag that was passed was invalid.
$2206	teInvalidPCount	The parameter count passed to the call was invalid.
$2207	teInvalidRect	The view rectangle was not valid. (it was less than twenty pixels wide)
$2208	teBufferOverflow	The buffer was too small. It is still filled with whatever text will fit.
$2209	teInvalidLine	The line number that was passed was to large.
$220A	teInvalidCall	This call is not allowed on the current TextEdit record.
$220B	teInvalidParameter	Some parameter that was passed was invalid.
$220C	teInvalidTextBox2	The embedded formatting codes in the TextBox2 format are inconsistant.

LIST OF HOOK ROUTINES:

Each TextEdit record has a number of standard hooks that the application programmer can install routines into. This allows applications to customize TextEdit so as to add features easily. The call for finding the address of the current hooks is *TEGetHooks* and the call for installing new hooks is *TESetHooks*. The following is a list of all the hook routines and their offsets into the buffer passed to *TEGetHooks* or *TESetHooks*:

Offset	Name	What it does
$0000	charFilter	This routine takes each character the user types and translates it into a TextEdit editing code or a character to insert.
$0004	wordWrap	This routine takes a character and returns whether or not it is a word break character. It is called only when TextEdit is word wrapping.
$0008	wordBreak	This routine takes a character and returns whether or not it is a word break character. It is called only when TextEdit is checking for a double click.
$000C	drawText	This routine is called when TextEdit redraws the text.
$0010	eraseText	This routine is called when TextEdit erases a rectangle.

Format for the hook routines will follow in a future version of this chapter

CONTROL MANAGER CALLS:

TextEdit records that are custom controls that support the following control manager calls. Note that the term "TextEdit control" is used to mean the actual TextEdit control itself and all it's associated scroll bars.

Call	Description
DisposeControl	This call disposes of the TextEdit control, just as if *TEKill* was called.
KillControls	This call disposes of all the TextEdit controls, just as if *TEKill* was called on each one of them.
HideControl	This will hide the TextEdit control. Note: this will not deactivate the TextEdit control; it will still take keystrokes and accept cut, copy, and paste operations. The only difference is that nothing will be displayed until the control is reshown.
EraseControl	This will erase the TextEdit control. It works just like *HideControl* except the control's bounding rectangle is not invalidated.
ShowControl	This will show the TextEdit control, undoing the effect of a Hide or Erase.
DrawControls	This will draw all the TextEdit controls in the window.
DrawOneCtl	This will draw the TextEdit control.
HilightControl	This will activate or deactivate the TextEdit control. Note: all part codes except for 0 and 255 are invalid.
FindControl	This will return $xxxx if the point was in a TextEdit control.
TestControl	This will return $xxxx if the point is in the TextEdit control.
TrackControl	This will select text just as if TEClick was called. It MUST be called with a negative number for the actionProcPtr to make the control manager call the built-in action procedure.
MoveControl	This will move the TextEdit control to it's new position and update all the internal fields that it needs to.
DragControl	This will let the user drag the TextEdit control around and reposition it.
SetCtlRefCon	This works normally; the RefCon field is reserved for the application.
GetCtlRefCon	This works normally; the RefCon field is reserved for the application.

These calls MUST NOT be made with a TextEdit control: *SetCtlTitle*, *SetCtlValue*, *SetCtlAction*, *SetCtlParams*

These calls should not be made with a TextEdit control: *GetCtlTitle*, *GetCtlValue*, *GetCtlAction*, *GetCtlParams*

STANDARD EDITING CHARACTERS:

Key	Alias	Description
LEFT-ARROW	CTRL-H	Moves the insertion point to the previous character in the text. If the **command** key is held down, the insertion point will be moved to the previous word. If the **option** key is held down the insertion point will be moved to the beginning of the line. If the **shift** key is held down the selection will be extended from the start. **shift** can be used in conjunction with **option** and **command**. Note: **option** overrides **command** .
RIGHT-ARROW	CTRL-U	Moves the insertion point to the next character in the text. If the **command** key is held down, the insertion point will be moved to the next word. If the **option** key is held down the insertion point will be moved to the end of the line. If the **shift** key is held down the selection will be extended from the end. **shift** can be used in conjunction with **option** and **command**. Note: **option** overrides **command** .
UP-ARROW	CTRL-K	Moves the insertion point to the next line up. If the **command** key is held down the insertion point will be moved to the beginning of the current page. If the **option** key is held down the insertion point will be moved to the top of the entire document. If the **shift** key is held down the selection will be extended from the start. **shift** can be used in conjunction with **option** and **command**. Note: **option** overrides **command** .
DOWN-ARROW	CTRL-J	Moves the insertion point to the next line down. If the **command** key is held down the insertion point will be moved to the end of the current page. If the **option** key is held down the insertion point will be moved to the bottom of the entire document. If the **shift** key is held down the selection will be extended from the end. **shift** can be used in conjunction with **option** and **command**. Note: **option** overrides **command** .
DELETE	CTRL-D	Removes the character to the left of the insertion point if there is no selection. If there is a selection, then only the selection is removed.
CLEAR		Clears the current selection if there is one. Does nothing if there is no selection. NOTE: TextEdit distinguishes clear from ctrl-x by checking the controlKey bit of the keyboard modifiers field.
CTRL-F		Removes the character to the right of the insertion point if there is no selection. If there is a selection, then only the selection is removed.
CTRL-Y		Clears all characters up to the end of the line whether or not there is a selection. If the line ends with a CR, it is not removed.
CTRL-X		Cuts the text into the clipboard (same as *TECut*).
CTRL-C		Copies the text into the clipboard (same as *TECopy*).
CTRL-V		Pastes from the clipboard (same as *TEPaste*).

Revision History

Date	Version	Description
12/16/88	0.19	Changed the TEPaintText and TEStyleChange calls. Changed the initial parameter block and made it incompatible with previous versions.
?/?/88	0.20	Converted to Microsoft Word. Also this document became a chapter in the Universe Toolbox Update instead of a seperate ERS.
1/9/88	0.21	Changed the initial parameter block. Once again it is incompatible. Changed the TEInsert and TEReplace calls to allow the style information to be passed as a pointer, handle, or resource ID. Cleaned up the format of the document. Changed how verbs (henceforth, reference descripters) are specified.
1/17/89	0.22	Added a word to the color table to specify the color of the area between pages. Removed the count field of the ruler structure. Changed the names of the ruler, style and color table structures. Changed the flowchart of using TextEdit without TaskMaster. Added a new structure: the StyleRecord.
2/3/89	0.23	Removed the fTransparentText bit from the TextFlags field. Also added a parameter to the TENew parameter block. Note: the parameter was added at the end, so old records are still valid ! Changed the description of TEPaintText call. The only parameter that actually changed is the *fl*_ parameter. Fixed description of TEInsertPageBreak. Documented that the *maxLines* and *maxHeight* fields are illegal in TextEdit v1.0. Added a new error code