Chapter 52 **Window Manager Update**

This chapter documents new features of the Window Manager. The complete reference to the Window Manager is in Volume 2, Chapter 25 of the *Apple IIGS Toolbox Reference.*

Error corrections

This section corrects some errors in the Window Manager documentation in the *Apple IIGS Toolbox Reference.*

- The manual's description of `SetZoomRect` is incorrect. The correct description is as follows:

  Sets the `fZoomed` bit of the window's `wFrame` record to 0. The rectangle passed to `SetZoomRect` then becomes the window's zoom rectangle. The window's size and position when `SetZoomRect` is called becomes the window's unzoomed size and position, regardless of what the unzoomed characteristics were before `SetZoomRect` was called.

- *Apple IIGS Toolbox Reference* page 25-126, third line:

  If `wmTaskMask` bit `tmInfo` (bit 15) = 1

  should read:

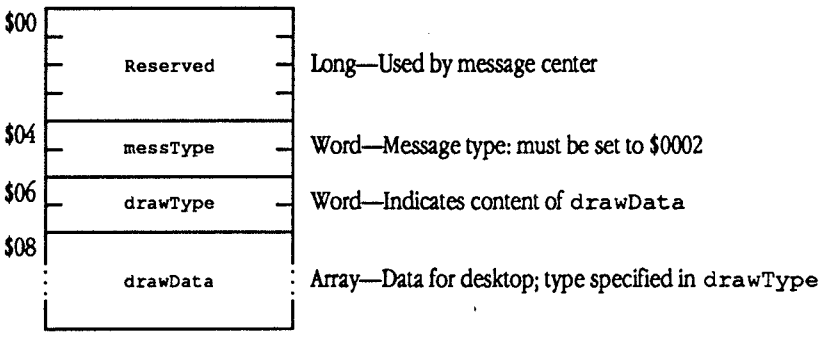
  If `wmTaskMask` bit `tmInfo` (bit 15) = 0

- When used with a window that does not have scroll bars, the call `WindNewRes` calls the window's defProc to recompute window regions. A call to `SizeWindow` is not necessary under these circumstances.

New features in the Window Manager

This section explains new features of the Window Manager, and clarifies points that were not made explicit before.

- TaskMaster now brings application windows to the front after dragging is complete. TaskMaster previously brought windows to the front before dragging.
- Using the `SetOriginMask` call, a programmer can control the horizontal scrolling characteristics of windows that TaskMaster scrolls. A common use of `SetOriginMask` is to ensure that the window origin is aligned on an even pixel, so that colors do not change if the display mode is changed between 320 and 640. When using the call, be sure that the horizontal scroll value is a whole multiple of the mask value. Otherwise, strange behavior can occur. As an extreme example, consider an origin value of 32 and a scroll amount of 1. Using the right scroll arrow will not scroll the window at all, and using the left one will scroll it by a value of 32. The new control value for the scrolling is calculated by adding or subtracting the scroll value and the current value and applying the mask. In this case adding 1 and masking results in the original value. Subtracting 1 and masking results in a new value that is 32 less than the old value.
- Standard windows can now draw their titles in 16 colors regardless of mode.
- The *grid* parameter of the call `DragWindow` has been renamed *dragFlag*. Bits 0 through 7 specify the *grid* value. Bits 8 through 14 are reserved bits; they must be set to 0. Bit 15 is a selection flag; if its value is 1, then the window will be brought to the top after dragging.
- It is no longer possible to specify *grid* values of 256 or 512.
- The Window Manager now uses the same default desktop drawing scheme as the Finder. When the Window Manager starts up, it looks for a `DeskMessage` in the message center. This `DeskMessage` is formatted as follows:



`drawType` Indicates the type of data stored at `drawData`:

0	`drawData` contains pattern information
1	`drawData` contains picture information

`drawData` Contains the pattern or picture data for the desktop image. If `drawType` is set to 0, then `drawData` contains 32 bytes of pattern data. The pattern defines 64 pixels arranged in an 8-by-8 array. In 320 mode 4 bits are needed for each pixel; in 640 mode, the system requires 2 bits per pixel. The system uses this pattern to seed the desktop image.

If `drawType` is set to 1, then `drawData` contains 32,000 bytes of picture data; the system copies this data directly to screen memory. See Chapter 16, "QuickDraw II," in the *Toolbox Reference* for details on pattern or picture images.

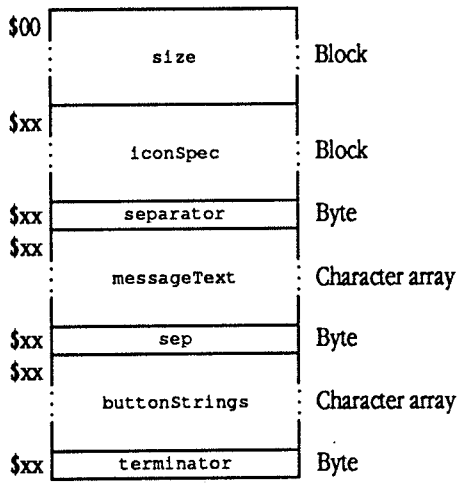
By loading a `DeskMessage` into the message center, your program can set a custom desktop image.

- Window Manager now supports a new entry point, TaskMasterDA, that allows desk accessories to use TaskMaster. Previously, desk accessories could not rely on TaskMaster, because they had to work with applications that do not use TaskMaster. Desk accessories obtain the data for their task record from the Desk Manager. TaskMaster processes task records for desk accessories in the same way that it processes application task records.
- The `SizeWindow` and `ResizeWindow` tool calls now invoke the `NotifyCtls` Control Manager tool call whenever the user changes the window size. This allows applications to show a control in a constant position with respect to the lower or right border of a window. For example, now the `growControl` control definition procedure can automatically move controls in response to a user dragging the size box.

Alert windows

The new `AlertWindow` call (described in "New Window Manager calls," later in this chapter) can be used to create **alert windows** for presenting the user with important messages. An alert window is similar to a modal dialog box. It requires that the user click a button in the window before doing anything else, and so provides a useful way to communicate vital messages such as warnings or error reports. The call does all the work of creating and displaying the window and contents for the alert, and returns the ID of the button that the user chooses.

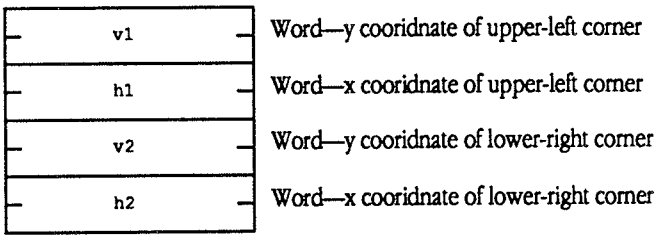
`AlertWindow` accepts a reference to a string that contains its message, and a reference to an array of substitution strings. The substitution strings can be any of seven standard strings (such as "OK," "Continue," and so on) or can be specified by the application and stored in the buffer to which the substitution-string pointer refers. The format of the `AlertWindow` input string is



size A variable-length block that specifies the size of the alert window to be displayed. Valid ASCII values for the first byte lie in the range from 0 through 9 and have the following meanings:

0	Custom size and position, specified by rectangle definition (as shown below)
1	30-character display window
2	60-character display window
3	110-character display window
4	175-character display window
5	110-character display window
6	150-character display window
7	200-character display window
8	250-character display window
9	300-character display window

If the value of the first byte of *size* is not 0, then the block consists only of that byte. If *size* is set to 0, then you must specify the custom rectangle immediately after the *size* field:



Since `AlertWindow` provides a limited number of standard sizes, it is possible to create alerts that display properly whether the Apple IIGS is in 320 or 640 mode. It is necessary, however, to design the text and buttons carefully in order to make this work.

Table 52-1 shows the dimensions of the standard alert windows. This table gives only an approximate idea of the size of each window. Application code should not rely on the exact widths, heights, or position of standard windows.

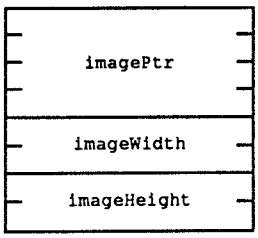
■ **Table 52–1** Standard alert window sizes

size value	Height 320	Width 320	Height 640	Width 640
1	46	152	46	200
2	62	176	54	228
3	62	252	62	300
4	90	252	72	352
5	54	252	46	400
6	62	300	54	452
7	80	300	62	500
8	108	300	72	552
9	134	300	80	600

iconSpec A variable-length block that specifies the type of icon to be displayed in the alert window. Valid ASCII values for the first byte lie in the range from `0` through `9` and have the following meanings:

`0`	No icon
`1`	Custom icon; followed by an icon specification, as shown below
`2`	Stop icon
`3`	Note icon
`4`	Caution icon
`5`	Disk icon
`6`	Disk swap icon
`7–9`	Reserved

If the first byte of *iconSpec* has a value other than `1`, then the field consists only of that byte. If the first byte is set to `1`, then it must be followed by an icon specification:



Long—Pointer to image data

Word—Width, in bytes, of the image data

Word—Height in scan lines, of the image data

separator Specifies a character that will divide sub-strings in the remainder of the `AlertWindow` input string. The *separator* field can contain any character, but the character cannot appear in the message text or button strings. The *separator* character divides the message from the first button string and the button strings from each other. For purposes of standardization, the slash (/) character is recommended, unless you will be substituting pathnames.

Do not include a separator character in any substitution strings. The Window Manager performs substitutions before scanning the alert string for separators. For example, if the separator character is a slash and a pathname containing several slashes is substituted for the string, the resulting alert window will contain several more buttons than you intended.

messageText Specifies the message to be displayed in the alert window. Any characters allowed by `LETextBox2` are allowed in the message text. See "Special Characters" later in this chapter for additional characteristics of `AlertWindow` message text. The total size of message text, after substitution of strings, is limited to 1000 characters.

sep A *separator* character.

buttonStrings Specifies titles for up to three buttons to be displayed in the alert window. If there is more than one title, then the titles must be separated from one another by a *separator* character. These buttons will be evenly spaced and centered at the bottom of the alert window. The width of each button is the same and is set by the widest button title. The maximum length of button text after substitution of strings is 80 characters.

terminator Marks the end of the alert string. Must be set to 0 ($00).

Special characters

The following **special characters** can be embedded in the message text and button strings of an `AlertWindow` input string. If a special character is to appear in the text of a button or message, you must enter it twice in the string. For example, if you want "^" to appear in an alert message, you must enter it in the message string as "^^".

^ A caret (^) designates the default button. The default button is the button selected if the user presses the Return key on the keyboard. This button will also appear outlined in bold on the screen. Only one button can be the default button. After the caret, the button title must follow, as for any other button. Other special characters may also appear after the caret. A single caret in the body of message text has no effect and is deleted from the message.

\# Substitute standard string. The number sign (#) must be followed by a decimal number. Numbers 0 through 6 can be used. Numbers 7 through 9 are reserved and should not be used. The standard substitution strings are

`#0`	OK
`#1`	Cancel
`#2`	Yes
`#3`	No
`#4`	Try Again
`#5`	Quit
`#6`	Continue

* Substitute given string. The asterisk (*) character followed by an ASCII decimal number from '0' through '9' denotes a substitution string to be inserted at that point. The asterisk and the following number will be replaced by the corresponding string in the specified substitution array. A pointer to the substitution array is passed as a parameter to the `AlertWindow` call. The substitution array is defined as an array of pointers. Table 52-2 shows the format of a substitution string array.

- **Table 52-2** Substitution string array

LONG[0]	Pointer to string that will substitute for `*0`
LONG[1]	Pointer to string that will substitute for `*1`
LONG[2]	Pointer to string that will substitute for `*2`
LONG[3]	Pointer to string that will substitute for `*3`
LONG[4]	Pointer to string that will substitute for `*4`
LONG[5]	Pointer to string that will substitute for `*5`
LONG[6]	Pointer to string that will substitute for `*6`
LONG[7]	Pointer to string that will substitute for `*7`
LONG[8]	Pointer to string that will substitute for `*8`
LONG[9]	Pointer to string that will substitute for `*9`

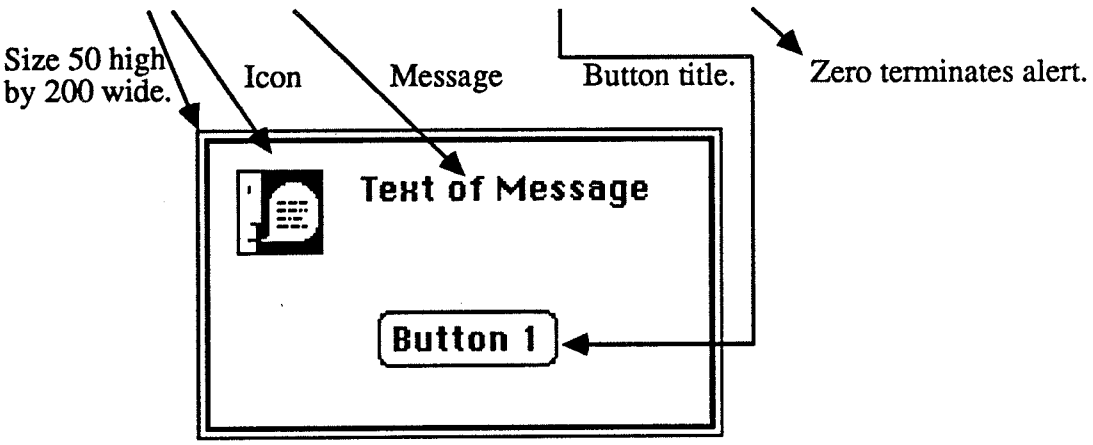
Substitution strings can be C strings or Pascal strings, or may be terminated by a carriage return. A parameter to the `AlertWindow` tool call allows you to specify the type of strings in the substitution array.

Alert window example

Following are some examples of alert strings that can be passed to `AlertWindow` in 65816 Assembly language syntax.

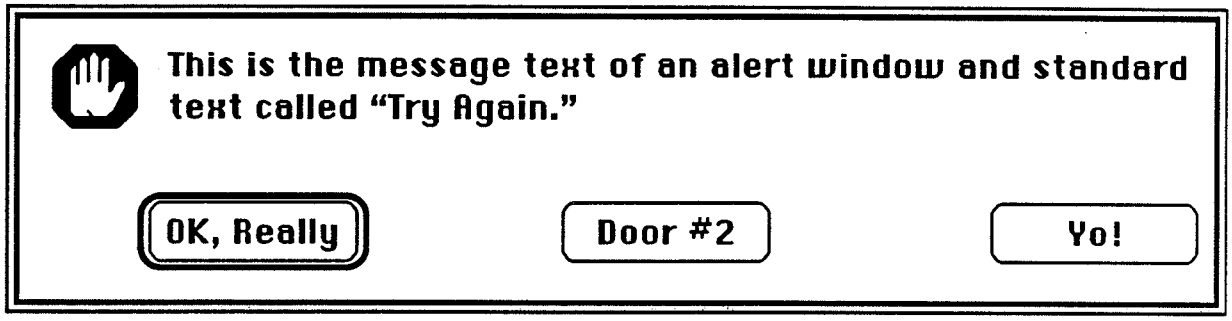
A simple alert string:

```
dc    c'13/Text of Message/Button 1',il'0'
```



A more complex alert string:

```
dc   c'51/This is the *0 of *3 alert *2*1 and standard'
dc   c'text called "#4" /'
dc   c'^#0,Really/*4/Yo!',il'0'
```



Where the substitution array =

```
      dc   i4'sub0,sub1,sub2,sub3,sub4'
sub0  dc   c'message text',il'0'
sub1  dc   c'dow',il'0'
sub2  dc   c'win',il'13'
sub3  dc   c'an',il'0'
sub4  dc   c'Door #2',il'0'
```

TaskMaster result codes

Table 52-3 lists all the possible TaskMaster result codes.

■ **Table 52-3** TaskMaster result codes

Name	Value	Description
Null	$0000	Successful
`mouseDownEvt`	$0001	Event Code -
`mouseUpEvt`	$0002	Event Code -
`keyDownEvt`	$0003	Event Code -
`autoKeyEvt`	$0005	Event Code -
`updateEvt`	$0006	Event Code -
`activateEvt`	$0008	Event Code -
`switchEvt`	$0009	Event Code -
`deskAccEvt`	$000A	Event Code -
`driverEvt`	$000B	Event Code -
`app1Evt`	$000C	Event Code -
`app2Evt`	$000D	Event Code -
`app3Evt`	$000E	Event Code -
`app4Evt`	$000F	Event Code -
`wNoHit`	$0000	Alias for no event
`inNull`	$0000	Alias for no event
`inKey`	$0003	Alias for keystroke
`inButtDwn`	$0001	Alias for button down
`inUpdate`	$0006	Alias for update event
`wInDesk`	$0010	On Desktop
`wInMenuBar`	$0011	On System Menu Bar
`wClickCalled`	$0012	`SystemClick` called (returned only as action)
`wInContent`	$0013	In content region
`wInDrag`	$0014	In drag region
`wInGrow`	$0015	In grow region, active window only
`wInGoAway`	$0016	In go-away region, active window only

`wInZoom`	$0017	In zoom region, active window only
`wInInfo`	$0018	In information bar
`wInSpecial`	$0019	Item ID selected was 250–255
`wInDeskItem`	$001A	Item ID selected was 1–249
`wInFrame`	$001B	In frame, but not on anything else
`wInactMenu`	$001C	Inactive menu item selected
`wClosedNDA`	$001D	Desk accessory closed (returned only as action)
`wCalledSysEdit`	$001E	SystemEdit called (returned only as action)
`wTrackZoom`	$001F	Zoom box clicked, but not selected (action only)
`wHitFrame`	$0020	Button down on frame, made active (action only)
`wInControl`	$0021	Button or keystroke in control (can be returned as event code and as action)
`wInControlMenu`	$0022	Control handled menu item
`wInSysWindow`	$8000	High bit set for system windows

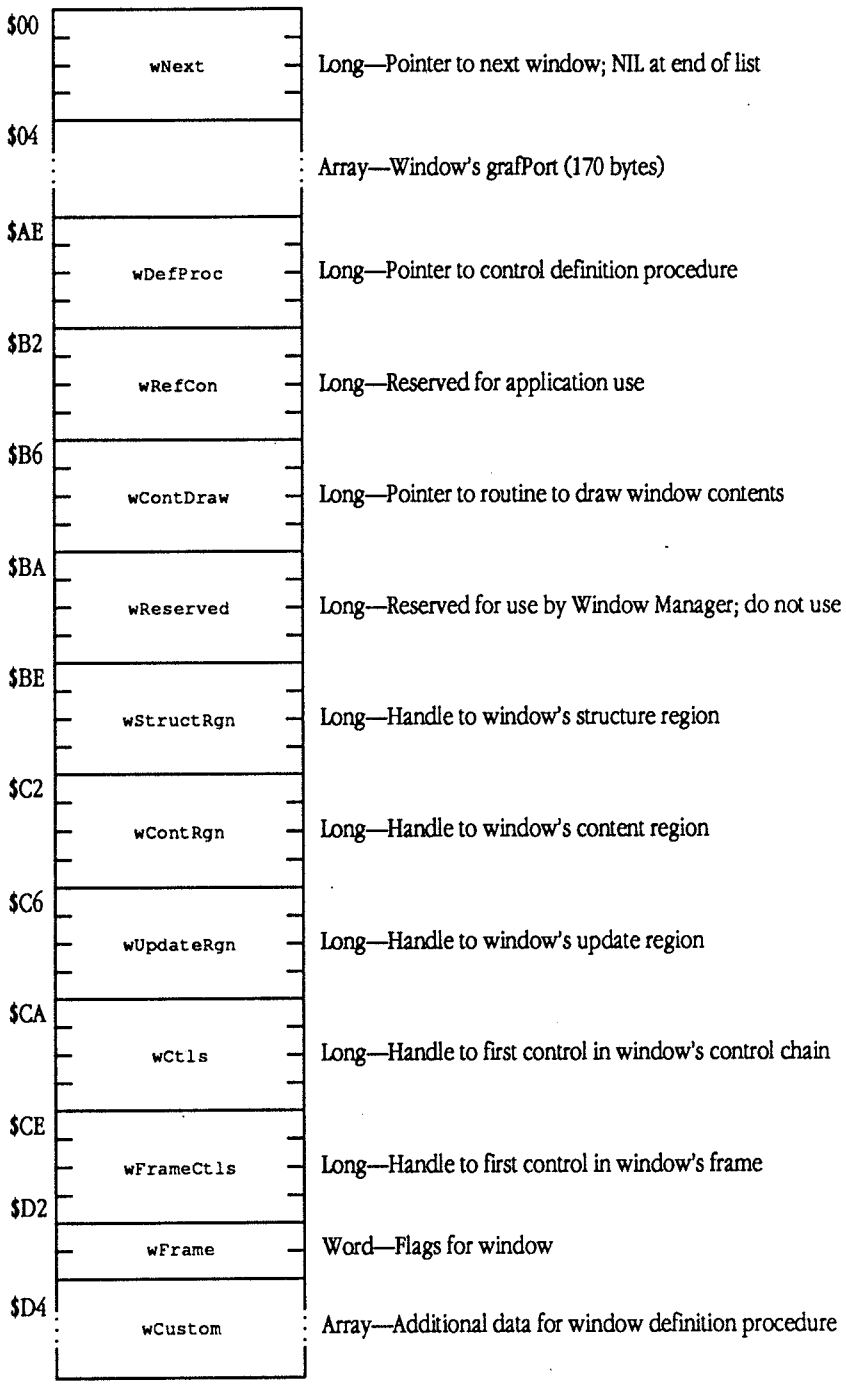
Window Manager data structures

This section discusses the format and content of changed Window Manager data structures.

Window record

The window record data structure has been redefined. The new definition is illustrated in Figure 52-1.

■ **Figure 52-1** Window record definition



`wReserved` A new data field reserved by Apple Computer, Inc. for future expansion.

`wFrame` A bit flag, containing flags specifying the window frame. All of the bits in this flag are described in Chapter 25, "Window Manager," in Volume 2 of the *Toolbox Reference*. Some of these bits may be used by window definition procedures. The following table lists the bits that may be used by window defProcs.

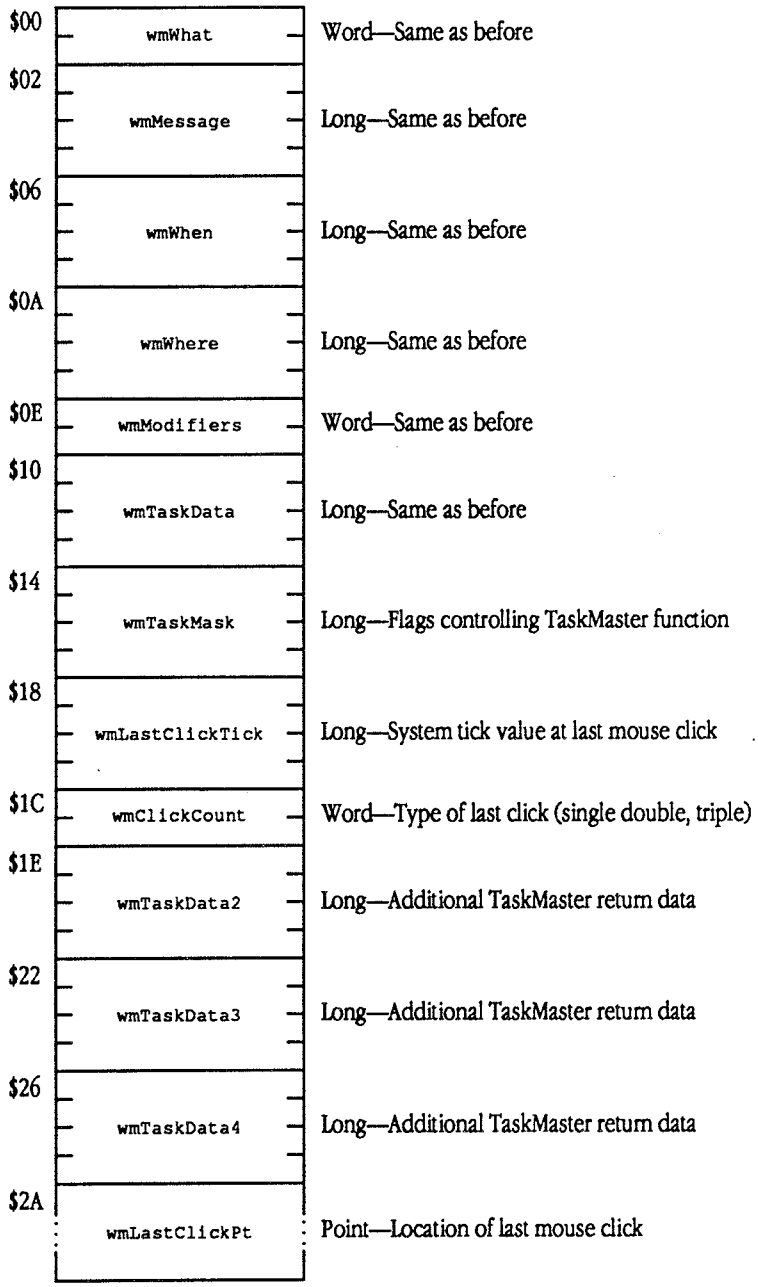
`fTitle`	bit 15
`fClose`	bit 14
`fAlert`	bit 13
`fRScroll`	bit 12
`fBScroll`	bit 11
`fGrow`	bit 10
`fFlex`	bit 9
`fZoom`	bit 8
`fMove`	bit 7
`fInfo`	bit 4
`fZoomed`	bit 1

Task record

Figure 52-2 defines the new format for the task record. This new record layout includes several new fields, each of which is set by TaskMaster every time your program calls TaskMaster. For information on the old fields, see Chapter 25, "Window Manager," in the *Toolbox Reference.*

TaskMaster will still accept old-format task records; however, if your program uses any of the new TaskMaster features (see `wmTaskMask`), it must use the new record layout.

■ **Figure 52-2** Task Record definition



`taskMask` Flag controlling TaskMaster function:

Reserved	bits 21–31	Must be set to 0
`tmIdleEvents`	bit 20	Controls whether TaskMaster sends idle events to the target control in the active window: 1 - Send idle events 0 - Do not send idle events
`tmMultiClick`	bit 19	Controls whether TaskMaster returns multiclick information in the Task Record: 1 - Return multiclick information 0 -Do not return multiclick information
`tmControlMenu`	bit 18	Controls whether TaskMaster passes menu events to controls in the active window: 1 - Pass menu events 0 - Do not pass menu events
`tmControlKey`	bit 17	Controls whether TaskMaster passes key events to controls in the active window: 1 - Pass key events 0 - Do not pass key events
`tmContentControls`	bit 16	Controls whether TaskMaster calls `FindControl` and `TrackControl` when `FindWindow` returns `wInContent` and the window is already selected: 1 - Track the control 0 - Do not track the control
`tmInfo`	bit 15	Controls whether TaskMaster activates the window when the user clicks in the info bar: 1 - Do not activate the window 0 - Activate the window
`tmInactive`	bit 14	Controls whether TaskMaster returns `wInactMenu` when the user selects an inactive menu item: 1 - Return `wInactMenu` 0 - Never return `wInactMenu`
`tmCRedraw`	bit 13	Controls whether TaskMaster redraws controls whenever an activate event occurs: 1 - Redraw controls 0 - Do not redraw controls
`tmSpecial`	bit 12	Controls whether TaskMaster handles special menu items (those with IDs < 256): 1 - Handle special menu items 0 - Do not handle special menu items

`tmScroll`	bit 11	Controls whether TaskMaster enables scrolling and activates inactive windows when the user clicks on the scroll bar: 1 - Enable scrolling 0 - Do not enable scrolling
`tmGrow`	bit 10	Controls whether TaskMaster calls `GrowWindow` when the user drags the size box: 1 - Call `GrowWindow` 0 - Do not call `GrowWindow`
`tmZoom`	bit 9	Controls whether TaskMaster calls `TrackZoom` when the user clicks in the zoom box: 1 - Call `TrackZoom` 0 - Do not call `TrackZoom`
`tmClose`	bit 8	Controls whether TaskMaster calls `TrackGoAway` when the user clicks in the close box: 1 - Call `TrackGoAway` 0 - Do not call `TrackGoAway`
`tmContent`	bit 7	Controls whether TaskMaster activates the window when the user clicks in the content region: 1 - Activate window 0 - Do not activate window
`tmDragW`	bit 6	Controls whether TaskMaster calls `DragWindow` when the user drags in the drag region: 1 - Call `DragWindow` 0 - Do not call `DragWindow`
`tmSysClick`	bit 5	Controls whether TaskMaster calls `SystemClick` when the user clicks in the system window: 1 - Call `SystemClick` 0 - Do not call `SystemClick`
`tmOpenNDA`	bit 4	Controls whether TaskMaster calls `OpenNDA` when the user selects a desk accessory: 1 - Call `OpenNDA` 0 - Do not call `OpenNDA`
`tmMenuSel`	bit 3	Controls whether TaskMaster calls `MenuSelect` when the user clicks in the menu bar: 1 - Call `MenuSelect` 0 - Do not call `MenuSelect`
`tmFindW`	bit 2	Controls whether TaskMaster calls `FindWindow` for mouse-down events: 1 - Call `FindWindow` 0 - Do not call `FindWindow`

`tmUpdate`	bit 1	Controls whether TaskMaster handles update events: 1 - Handle update events 0 - Do not handle update events
`tmMenuKey`	bit 0	Controls whether TaskMaster calls `MenuKey` to handle menu key equivalents: 1 - Call `MenuKey` 0 - Do not call `Menukey`

New Window Manager calls

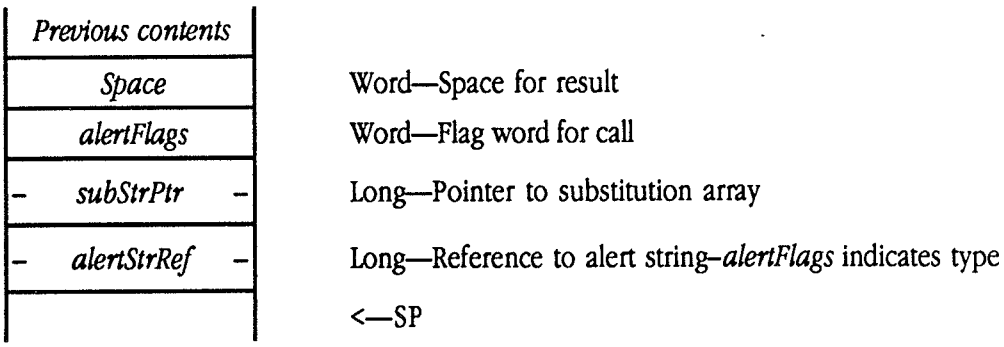
The following tool calls have been added to the Window Manager since publication of the first two volumes of the *Apple IIGS Toolbox Reference.*

AlertWindow $590E

Creates an alert window that displays a message referred to by *alertStrRef.* The message can be either a C or a Pascal string, as specified by *alertFlags.* The *subStrPtr* parameter points to an array of substitution strings for use with substitution characters. For more detailed information, see "Alert Windows" earlier in this chapter.

Parameters

Stack before call



Word—Space for result

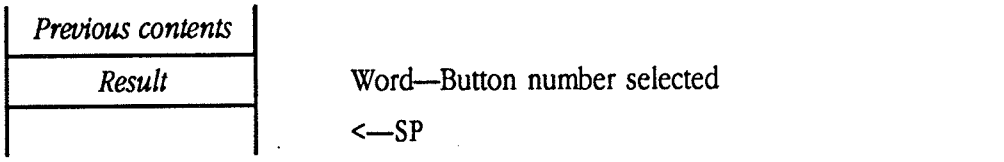
Word—Flag word for call

Long—Pointer to substitution array

Long—Reference to alert string–*alertFlags* indicates type

<—SP

Stack after call



Word—Button number selected

<—SP

Errors None

C

```
extern pascal Word AlertWindow(alertFlags,
                               subStrPtr, alertStrRef);

Word        alertFlags;
Pointer     subStrPtr;
Long        alertStrRef;
```

alertFlags Contains flags that indicate the type of strings referenced by *alertStrRef*, as well as the type of reference contained that field:

Reserved	bits 3–15	Must be set to 0
`referenceType`	bits 1–2	Indicate the type of reference stored in *alertStrRef*: 00 - *alertStrRef* is a pointer 01 - *alertStrRef* is a handle 10 - *alertStrRef* is a resource ID
`stringType`	bit 0	Indicates type of string referred to by *alertStrRef*: 0 - C string (null-terminated) 1 - Pascal string

`CompileText` $600E

Combines source text provided by your program with either custom or standard strings to compile a result text string. For successful calls, this call allocates and correctly sizes a handle to the result text string. That result string is a simple character array. Your program must extract length information for the string from the handle. Note that your program must dispose of this handle.

Control sequences in the source text direct the system to embed either custom or standard strings into the result text string. These control sequences consist of two ASCII characters: a flag character followed by a digit. The flag character indicates whether the desired substitution string is custom or standard.

For standard strings, the flag character is `#`. The digit following the flag character selects one of the following strings:

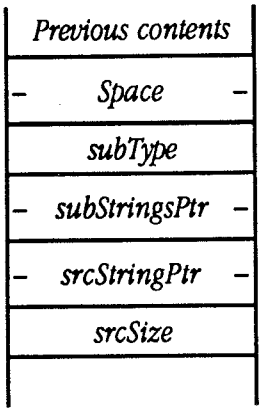
`#0`	OK
`#1`	Cancel
`#2`	Yes
`#3`	No
`#4`	Try Again
`#5`	Quit
`#6`	Continue

For custom strings, the flag character is `*`. `CompileText` obtains custom strings from a substitution array built by your program and provided to the system in the parameters for this call. The character following the flag character specifies which string to extract. Valid values for this character lie in the range `0` through `9`. Thus, a control sequence of `*0` would access the first string in your custom substitution array.

In order to include either of the flag characters as text in your compiled text, follow the flag character with a second flag character (for example, `**` results in `*` in the compiled text string).

Parameters

Stack before call



Long—Space for result

Word—Type of custom substitution strings

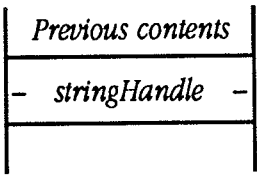
Long—Pointer to substitution array

Long—Pointer to source string

Word—Length of source string pointed to by *srcStringPtr*

<—SP

Stack after call



Long—Handle to result string

<—SP

Errors

$0E04	`compileTooLarge`	Compiled text is larger than 64k

C

```
extern pascal Handle CompileText(subType,
          subStringsPtr, srcStringPtr, srcSize);

Word      subType, srcSize;
Pointer   subStringsPtr, srcStringPtr;
```

subType Indicates the type of strings stored in the substitution array pointed to by *subStringsPtr*.

0	Array contains C strings
1	Array contains Pascal strings

Note that this field is ignored if your program does not use any custom substitution strings.

subStringsPtr Contains a pointer to your custom text substitution array. This array contains from 1 to 10 long pointers to either C or Pascal strings (use *subType* to indicate which type of string you have used). Embedded control sequences in your source text direct the system to extract a specific string from this array. Note that the system does not verify string specifications against the size of this array; be careful to define the correct number of string pointers in this array.

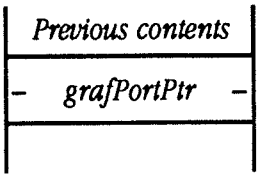
Note that this field is ignored if your program does not use any custom substitution strings.

DrawInfoBar $550E

Redraws the info bar of the window specified by *grafPortPtr*. The method used to redraw the info bar's interior is the routine specified by the *wInfoDefProc* field of the `paramList` passed to `NewWindow` when the window is created. The Window Manager will automatically clip info bar drawing to the dimensions of the info bar, and to the visible region of the window.

Parameters

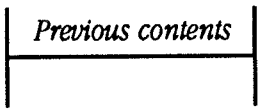
Stack before call



Long—Pointer to GrafPort for window

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void DrawInfoBar(grafPortPtr);

Pointer   grafPortPtr;
```

`EndFrameDrawing` $5B0E

Restores Window Manager variables after a call to `StartFrameDrawing`.

Parameters This call has no input or output parameters. The stack is unaffected.

Errors None

C

```
extern pascal void EndFrameDrawing();
```

ErrorWindow $620E

Creates a dialog box displaying an error message for a specified error code. GS/OS error codes are listed along with standard message text in "Error messages" later in this chapter.

Each error message is in alert string format and may require a substitution string (see "Alert windows" earlier in this chapter for message format and text substitution information). The system retrieves the error messages from a resource file of type $8020. The resource ID for each message is formed as follows:

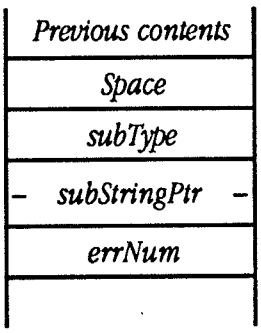
high-order word	$07FF
low-order word	error number

The default error messages are stored in the system resources file.You may assert custom error message text by defining and opening another resource file containing type $8020 resources with appropriate resource IDs assigned to each error message. Make sure that your resource file precedes the system resource file in the Resource Manager's search sequence. A custom error message resource file need not define substitute messages for all possible GS/OS errors; if the Resource Manager does not find a message in your file, it will continue through the standard resource search sequence.

If ErrorWindow receives an undefined error code, it displays a dialog box with the "Unknown Error" message ($72).

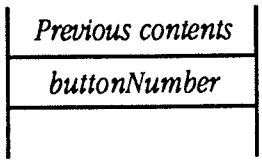
Parameters

Stack before call



Word—Space for result
Word—Type of custom substitution string
Long—Pointer to substitution string
Word—GS/OS error number
<—SP

Stack after call



Word—Number of button pressed by the user
<—SP

Errors Resource Manager errors returned unchanged

C

```
extern pascal Word ErrorWindow(subType,
          subStringPtr, errNum);

Word      subType, errNum;
Pointer   subStringPtr;
```

subType Indicates the type of string pointed to by *subStringPtr*.

0	C string
1	Pascal string

Note that this field is ignored if the specified error message does not use any substitution strings.

subStringPtr Contains a pointer to your custom text substitution string.

Note that this field is ignored if the specified error message does not use any substitution strings.

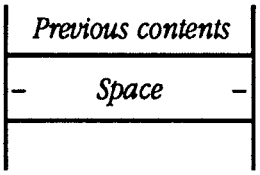
GetWindowMgrGlobals $580E

Returns a pointer to the Window Manager global data area.

▲ **Warning** An application should never make this call ▲

Parameters

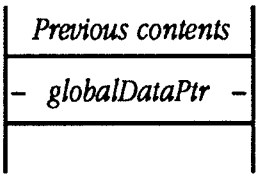
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to the global data area

<—SP

Errors None

C

```
extern pascal Pointer GetWindowMgrGlobals();
```

NewWindow2 $610E

Performs the same function as NewWindow, but allows you to specify the input window template as a resource (type rWindParam1 or rWindParam2). See Appendix E, "Resource Types," later in this book for complete descriptions of all resource types.

- *Note:* If you have specified the window template as a resource, then the references within that template to title, color table, and control list must also be resources (or NIL).

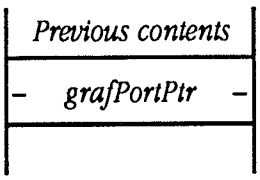
- *Note:* In order to create an InfoBar with NewWindow2 using a window template defined as a resource, you must specify a NIL infoDraw procedure in the input template and create an invisible window. After issuing the NewWindow2 call, set the infoDraw routine by calling SetInfoDraw, then make the window visible with the ShowWindow tool call.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
– *titlePtr* –	Long—Pointer to replacement title
– *refCon* –	Long—RefCon to replace value in template
– *contentDrawPtr* –	Long—Pointer to replacement content draw Routine
– *defProcPtr* –	Long—Pointer to replacement window definition procedure
paramTableDesc	Word—Indicates type of reference in *paramTableRef*
– *paramTableRef* –	Long—Reference to window template
resourceType	Word—Resource type of template referred to by *paramTableRef*
	<—SP

Stack after call



Long—Pointer to window GrafPort; NIL if unsuccessful

<—SP

Errors

Resource Manager errors	returned unchanged
Memory Manager errors	returned unchanged
Window Manager errors	returned unchanged from `NewWindow`
Control Manager errors	returned unchanged from `NewControl2`

C

```
extern pascal Pointer NewWindow2(titlePtr, refCon,
          contentDrawPtr, defProcPtr,
          paramTableDesc, paramTableRef,
          resourceType);

Word      paramTableDesc, resourceType;
Pointer   titlePtr, contentDrawPtr, defProcPtr;
Long      refCon, paramTableRef;
```

titlePtr, *refCon*, *contentDrawPtr*, *defProcPtr*

`NewWindow2` will replace the values supplied in the template referred to by *paramTableRef* with the contents from these fields, allowing you to use a standard template and tailor it to create different windows. To prevent `NewWindow2` from replacing the template values, supply NIL pointers in *titlePtr*, *contentDrawPtr*, and *defProcPtr*.

paramTableDesc Indicates the type of reference stored in *paramTableRef*:

$0000	*paramTableRef* contains a pointer to a window template
$0001	*paramTableRef* contains a handle to a window template
$0002	*paramTableRef* contains the resource ID of a window template

paramTableRef Reference to a window template. The *paramTableDesc* field defines the type of reference stored here. The *resourceType* field defines the resource type for the template. The template must comply with the format specification of resource type `rWindParam1` or `rWindParam2` (even if the template is not stored as a resource). See Appendix E, "Resource Types," in this book for information on the format and content of these resources.

resourceType Specifies the type of window template referred to by *paramTableRef*. This value should be set correctly even if *paramTableRef* does not contain a resource ID. Valid values are:

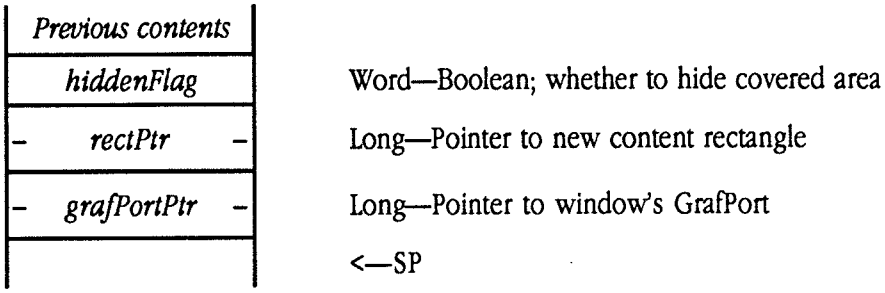
$800E	`rWindParam1`
$800F	`rWindParam2`

`ResizeWindow` $5C0E

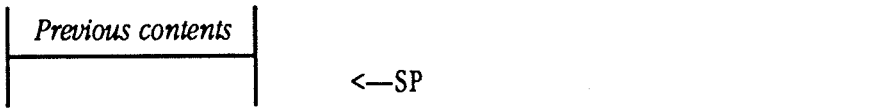
Moves, resizes, and draws the window specified by *grafPortPtr*. The *rectPtr* parameter is a pointer to the window's content region. The *hiddenFlag* parameter is a Boolean parameter; a TRUE value specifies that those portions of the window that are covered should not be drawn. If the value is FALSE, the entire window is drawn, covered or not.

Parameters

Stack before call



Stack after call



Errors None

C

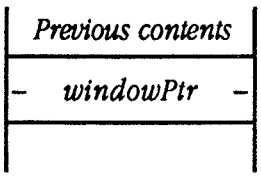
```
extern pascal void ResizeWindow(hiddenFlag, rectPtr,
                grafPortPtr);

Word      hiddenFlag;
Pointer   rectPtr, grafPortPtr;
```

`StartFrameDrawing` $5A0E

Sets up Window Manager data to draw a window frame. Should be called only by window definition procedures. Must be balanced by a call to `EndFrameDrawing` when drawing is completed.

Parameters

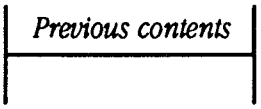
Stack before call



Long—Pointer to window to draw

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void StartFrameDrawing(windowPtr);

Pointer   windowPtr;
```

TaskMasterContent $5D0E

Internal routine that handles events inside the content region of a window. TaskMaster invokes this routine if the `tmContentControls` bit of the `taskMask` field of the task record is set to 1. Your program should never issue this call.

Pseudo-code:

```
if tmContentControls in wmTaskMask = 1
      if mousedown in content region of frontmost window
            set wmTaskData2, wmTaskData3, and wmTaskData4 to $00000000
            call FindControl
            put resulting partCode into low-order word of wmTaskData3
            put controlHandle into wmTaskData2
            if partCode <> 0
                  call GetCtlID
                  put resulting control ID into wmTaskData4
                  call TrackControl with actionProcPtr set to $FFFFFFFF
                  if result <> 0 or part code corresponds to scroll bar
                        put resulting partCode into high-order word of
                              wmTaskData3
                        if the control is a check box or radio button
                              Set or clear the value, as appropriate
                        endif
                        return(wInControl)
                  endif
                  set low word of wmTaskData = wInControl
                  return (nullEvt)
            endif
      else
            set wmTaskData = pointer to window
            return(wInContent)
      endif
endif
```

`TaskMasterContent` calls `FindControl`. If the user did not press the button in a control, then the routine returns a result code of `wInContent`, indicating that the mouse is in the content region of the window.

If the user did press the mouse button in a control, `TaskMasterContent` calls `TrackControl`, directing the Control Manager to use the appropriate action procedure for the control.

When `TrackControl` returns, `TaskMasterContent` examines the part code. If the part code is set to 0, then the user decided not to use the control (released the mouse button outside the control). `TaskMasterControl` returns a result code of `nullEvt` ($0000).

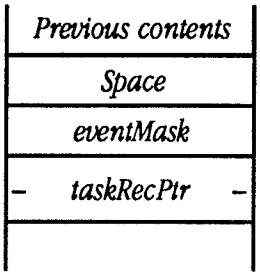
If the part code is non-zero, then the user released the mouse button within a control. `TaskMasterContent` returns a result code of `wInControl`, `wmTaskData2` contains the control handle, `wmTaskData3` (low-order word) contains the part code identifying the control in which the user pressed the mouse button, `wmTaskData3` (high-order word) contains the part code identifying the control where the user released the mouse button, and wm`TaskData4` contains the control ID (if there is one defined).

TaskMasterDA $5F0E

This call is the TaskMaster entry point for desk accessories. Your program passes event information obtained from the Desk Manager.

Parameters

Stack before call



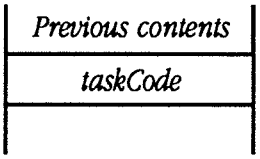
Word—Space for result

Word—Not used

Long—Pointer to extended Task Record

<—SP

Stack after call



Word—TaskMaster result code

<—SP

Errors None

C

```
extern pascal Word TaskMasterDA(eventMask,
                taskRecPtr);
                Pointer     taskRecPtr;

Word        eventMask;
```

TaskMasterKey $5E0E

Internal routine that handles keystroke events inside the content region of a window. Your program should never issue this call.

Pseudo-code:

```
if tmMenuKey in wmTaskMask =1
      if wmTaskData = 0        (menu did not take keystroke)
            if tmInactive in wmTaskMask =1
                  if high word of wmTaskData <> 0
                        set low word of wmTaskData = 0
                        set high word of wmTaskData = ID of selected
                              inactive menu item
                        return (wInActMenu)
                  endif
                  goto CheckControls
            endif
      else                     (menu did take keystroke)
            if low word of wmTaskData > 255
                  if tmControlMenu in wmTaskMask = 1
                        Call SendEventToCtl with targetOnlyFlag = TRUE
                        if result <> 0
                              set wmTaskData2 = handle of control that
                                    took keystroke
                              set wmTaskData3 = result code from defProc
                              set wmTaskData4 = ID of control that took
                                    keystroke
                              dim the menu title for selected menu item
                              set low word of wmTaskData =
                                    wInControlMenu
                              return (nullEvt)
                        endif
                        set low word of wmTaskData = ID of selected menu
                              item
                        set high word of wmTaskData = ID of menu from
                              which selection was made
                        return (wInMenuBar)
                  endif
```

```
elseif low word of wmTaskData < 250
      if tmOpenNDA in wmTaskMask = 0
            set low word of wmTaskData = ID of selected menu
                  item
            set high word of wmTaskData = ID of menu from
                  which selection was made
            return (wInDeskItem)
      endif
      call OpenNDA
      dim menu title for selected menu item
      set low word of wmTaskData = wInDeskItem
      return (nullEvt)
elseif tmSpecial of wmTaskMask = 0
      set low word of wmTaskData = ID of selected menu item
      set high word of wmTaskData = ID of menu from which
            selection was made
      return (wInSpecial)
elseif top window is an application window
      if tmControlMenu of wmTaskMask = 1
            call SendEventToCtl with targetOnlyFlag = TRUE
            if result <> 0
                  set wmTaskData2 = handle of control that
                        took keystroke
                  set wmTaskData3 = result code from defProc
                  set wmTaskData4 = ID of control that took
                        keystroke
                  dim the menu title for selected menu item
                  set low word of wmTaskData =
                        wInControlMenu
                  return (nullEvt)
            endif
      endif
      set low word of wmTaskData = ID of selected menu item
      set high word of wmTaskData = ID of menu from which
            item was selected
      return (wInSpecial)
```

```
            elseif low word of wmTaskData = 250, 251, 252, 253, or 254
                  call SystemEdit
                  if SystemEdit returns FALSE
                        set low word of wmTaskData = ID of selected menu
                              item
                        set high word of wmTaskData = ID of menu from
                              which item was selected
                        return (wInSpecial)
                  endif
                  dim menu title for menu item that was selected
                  set low word of wmTaskData = wCalledSysEdit
                  return (nullEvt)
            elseif low word of wmTaskData = 255
                  call CloseNDAbyWinPtr for top window
                  dim menu title for menu item that was just selected
                  set low word of wmTaskData = wClosedNDA
                  return (nullEvt)
            endif
      endif
endif

CheckControls:

if tmControlKey in wmTaskMask = 1
      set wmTaskData2, wmTaskData3, and wmTaskData4 = 0
      if there is a front window
            call SendEventToCtl with targetOnlyFlag = FALSE
            if result <> 0
                  set wmTaskData2 = handle of control that took the
                        keystroke
                  set wmTaskData3 = result from defProc
                  set wmTaskData4 = ID of control that took the
                        keystroke
                  set wmTaskData = window containing control
                  if control is a check box or radio button
                        set the ctlValue for the control
                  endif
                  return (wInControl)
            endif
      endif
      return (keyDownEvt or autoKeyEvt)
endif
```

`TaskMasterKey` first checks to see if menu keys are to be passed to the Menu Manager. If so, `TaskMasterKey` calls `MenuKey`. If the user entered a menu keystroke, `MenuKey` handles it and `TaskMasterKey` returns control to the calling application.

If the user did not enter a menu key equivalent or if keystrokes are not to be passed to the Menu Manager, `TaskMasterKey` looks for a control in the active window that wants the keystroke. If a control takes the event, `TaskMasterKey` returns `nullEvt` to the calling application. Otherwise, `TaskMasterKey` returns `keyDownEvt`, indicating that the keystroke is for the application.

`GDRPrivate` $540E

This is an internal Window Manager call; your program should never issue this call.

Error messages

This section documents the error numbers and accompanying messages produced by the `ErrorWindow` tool call. For each error number, the following table specifies the message text displayed in the dialog box, the icon shown, and the button(s) available for the user to press. Any required substitution strings are shown in the message text.

Error (Hex)	Message	Icon	Button
$00	`No error occurred.`	None	OK
$01	`Bad system call number.`	None	OK
$04	`Invalid parameter count.`	None	OK
$07	`GS/OS already active.`	None	OK
$10	`Device not found.`	None	OK
$11	`Invalid device number.`	None	OK
$20	`Bad request or demand.`	None	OK
$21	`Bad control or status code.`	None	OK
$22	`Bad call parameter.`	None	OK
$23	`Character device not open.`	None	OK
$24	`Character device already open.`	None	OK
$25	`Interrupt table full.`	None	OK
$26	`Resources not available.`	None	OK
$27	`I/O error.`	None	OK
$28	`Device not connected.`	None	OK
$29	`Driver is busy and not available.`	None	OK
$2B	`Device is write protected.`	None	OK
$2C	`Invalid byte count.`	None	OK
$2D	`Invalid block number.`	None	OK
$2E	`Disk has been switched.`	None	OK
$2F	`Device off-line/no media present.`	None	OK
$40	`Invalid pathname syntax.`	None	OK
$43	`Invalid reference number.`	None	OK
$44	`Subdirectory does not exist.`	None	OK
$45	`Volume not found.`	None	OK
$46	`File not found.`	None	OK
$47	`Duplicate pathname.`	None	OK
$48	`Volume full.`	None	OK
$49	`Volume directory full.`	None	OK

$4A	Version error.	None	OK
$4B	Bad storage type.	None	OK
$4C	End of file encountered.	None	OK
$4D	Position out of range.	None	OK
$4E	Access not allowed.	None	OK
$4F	Buffer too small.	None	OK
$50	File is already open.	None	OK
$51	Directory error.	None	OK
$52	Unknown volume type.	None	OK
$53	Parameter out of range.	None	OK
$54	Out of memory.	None	OK
$57	Duplicate volume name.	None	OK
$58	Not a block device.	None	OK
$59	Specified level is outside legal range	None	OK
$5A	Block number too large.	None	OK
$5B	Invalid pathnames for change_path.	None	OK
$5C	Not an executable file.	None	OK
$5D	Operating system not supported.	None	OK
$5F	Stack overflow.	None	OK
$60	Data unavailable.	None	OK
$61	End of directory has been reached.	None	OK
$62	Invalid FST call class.	None	OK
$63	File does not contain requested resource.	None	OK
$64	Specified FST is not present in system	None	OK
$65	FST does not handle this type of call	None	OK
$66	FST handled call, but result is weird	None	OK
$67	Internal error.	None	OK
$68	Device list is full.	None	OK
$69	Supervisor list is full.	None	OK
$70	Cannot expand file, resource already exists.	None	OK

$71	Cannot add resource fork to this type of file.	None	OK
$72	Unknown error: [*error string*].	None	Cancel
$80	Error creating the new directory: [*reason string*].	Stop	Cancel
$81	Error saving the file: [*reason string*].	Stop	Cancel
$82	Insufficient access privileges to open that folder.	Stop	OK
$83	The selected folder cannot be opened: [*reason string*].	Stop	Cancel
$84	You cannot replace a folder with a file.	Stop	Cancel
$85	That file already exists.	Stop	Cancel Replace
$86	Insufficient memory to perform that operation. About [*number string*]K additional needed	Stop	Cancel
$87	Initialization failed: Disk write protected.	Stop	Cancel
$88	The pathname is too long.	Stop	OK
$89	The disk is write protected.	Caution	Cancel
$8A	The disk is full.	Stop	Cancel
$8B	The disk directory is full.	Stop	Cancel
$8C	The file is copy-protected and can't be copied.	Stop	Cancel
$8D	Memory is full.	Stop	OK
$8E	There isn't enough memory remaining to complete this operation. Please close some windows and try again.	Stop	OK
$8F	The item is locked and can't be renamed.	Stop	Cancel
$90	An I/O error has occurred while using the disk.	Stop	Cancel
$91	This disk seems to be damaged	Stop	Cancel
$92	Not a ProDOS disk.	Stop	OK
$93	No on-line volumes can be found.	Stop	OK
$94	Insert the disk: [*name string*].	Swap	Cancel

Appendix E **Resource Types**

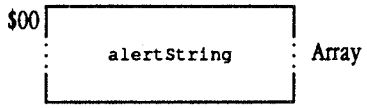
This appendix documents the format and content of standard resources used by the Apple IIGS toolbox. The resources are discussed in alphabetical order by resource type name.

rAlertString $8015

Figure E-1 defines the layout of resource type `rAlertString` ($8015). Resources of this type define the data for Alert Windows to be displayed by the `AlertWindow` Window Manager tool call. For more complete information on Alert Window definitions, see Chapter 52, "Window Manager Update," earlier in this book.

■ **Figure E–1** Alert string, type `rAlertString` ($8015)

`AlertWindow` accepts a reference to a string that contains its message, and a reference to an array of substitution strings. The substitution strings can be any of seven standard strings (such as "OK", "Continue", and so on) or can be specified by the application and stored in the buffer to which the substitution-string pointer refers.



`alertString` Defines the alert message to be displayed. Contents of this string must comply with the rules for Alert Window definitions documented in Chapter 52, "Window Manager Update," earlier in this book.

rControlList $8003

Figure E-2 defines the layout of resource type `rControlList` ($8003). The Control Manager stores lists of resource IDs in resources of this type.

■ **Figure E–2** Control List, type `rControlList` ($8003)

`ctlList` List of resource IDs for control template definitions. The last entry must be set to NIL.

rControlTemplate $8004

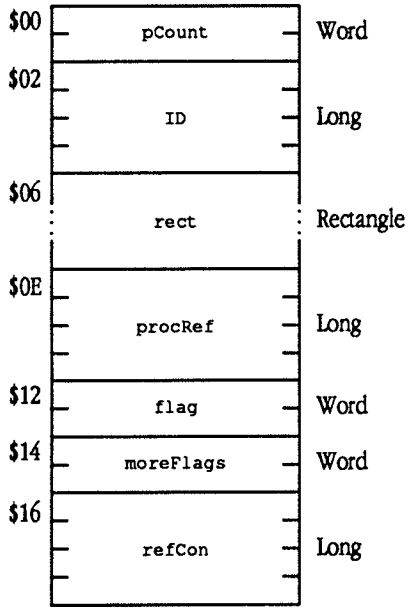
Resources of type `rControlTemplate` ($8004) define control templates, used with the Control Manager `NewControl2` tool call to create controls. You fill a type `rControlTemplate` resource according to the needs of the particular control you want to create. The system distinguishes between different control templates by examining the `procRef` field in the standard header portion that precedes each template.

Control template standard header

Each control template contains the standard header, which consists of seven fields. Following that header, some templates have additional fields, which further define the control to be created. The format and content of the standard template header is shown in Figure E-3.

Custom control definition procedures establish their own item template layout. The only restriction placed on these templates is that the standard header be present and well formed. Custom data for the control procedure may follow the standard header.

■ **Figure E-3** Control template standard header



`pCount` Count of parameters in the item template, not including the `pCount` field. Minimum value is 6, maximum value varies depending upon the type of control template.

`ID` Sets the `ctlID` field of the control record for the new control. The `ctlID` field may be used by the application to provide a straightforward mechanism for keeping track of controls. The control ID is a value assigned by your application, which the control "carries around" for your convenience. Your application can use the ID, which has a known value, to identify a particular control.

`rect` Sets the `ctlRect` field of the control record for the new control. Defines the boundary rectangle for the control.

`procRef` Sets the `ctlProc` field of the control record for the new control. This field contains a reference to the control definition procedure for the control. The value of this field is either a pointer to a control definition procedure, or the ID of a standard routine. The standard values are:

`simpleButtonControl`	$80000000	Simple button
`checkControl`	$82000000	Check box
`iconButtonControl`	$07FF0001	Icon button
`editLineControl`	$83000000	LineEdit
`listControl`	$89000000	List
`pictureControl`	$8D000000	Picture
`popUpControl`	$87000000	Pop-up
`radioControl`	$84000000	Radio control
`scrollBarControl`	$86000000	Scroll bar
`growControl`	$88000000	Size box
`statTextControl`	$81000000	Static Text
`editTextControl`	$85000000	TextEdit

`flag` A word used to set both `ctlHilite` and `ctlFlag` in the control record for the new control. Since this is a word, the bytes for `ctlHilite` and `ctlFlag` are reversed. The high-order byte of `flag` contains `ctlHilite`, while the low-order byte contains `ctlFlag`. The bits in `flag` are mapped as follows:

Highlight	bits 8–15	Indicates highlighting style: 0 Control active, no highlighted parts 1–254 Part code of highlighted part 255 Control inactive
Invisible	bit 7	Governs visibility of control: 0 - Control visible 1 - Control invisible
Variable	bits 0–6	Values and meaning depends upon control type

`moreFlags` Used to set the `ctlMoreFlags` field of the control record for the new control.

The high-order byte is used by the Control Manager to store its own control information. The low-order byte is used by the control definition procedure to define reference types.

The defined Control Manager flags are:

`fCtlTarget`	$8000	If set to 1, this control is currently the target of any typing or editing commands.
`fCtlCanBeTarget`	$4000	If set to 1 then this control can be made the target control.
`fCtlWantEvents`	$2000	If set to 1 then this control can be called when events are passed via the `SendEventToCtl` Control Manager call. Note that, if the `fCtlCanBeTarget` flag is set to 1, this control will receive events sent to it regardless of setting of this flag.
`fCtlProcRefNotPtr`	$1000	If set to 1, then Control Manager expects `ctlProc` to contain the ID of a standard control procedure. If set to 0, then `ctlProc` contains a pointer to the custom control procedure.
`fCtlTellAboutSize`	$0800	If set to 1, then this control needs to be notified when the size of the owning window has changed. This flag allows custom control procedures to resize their associated control images in response to changes in window size.
`fCtlIsMultiPart`	$0400	If set to 1, then this is a multipart control. This flag allows control definition procedures to manage multi-part controls (necessary since the Control Manager does not know about all the parts of a multi-part control).

The low-order byte uses the following convention to describe references to color tables and titles (note, though, that some control templates do not follow this convention):

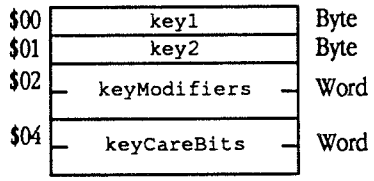
`titleIsPtr`	$00	Title reference is by pointer
`titleIsHandle`	$01	Title reference is by handle
`titleIsResource`	$02	Title reference is by resource ID
`colorTableIsPtr`	$00	Color table reference is by pointer
`colorTableIsHandle`	$04	Color table reference is by handle
`colorTableIsResource`	$08	Color table reference is by resource ID

`refCon` Used to set the `ctlRefCon` field of the control record for the new control. Reserved for application use.

Keystroke equivalent information

Many of these control templates allow you to specify keystroke equivalent information for the associated controls. Figure E-4 shows the standard format for that keystroke information.

■ **Figure E-4** Keystroke equivalent record layout



`key1` This is the ASCII code for the upper or lower case of the key equivalent.

`key2` This is the ASCII code for the lower or upper case of the key equivalent. Taken with `key1`, this field completely defines the values against which key equivalents will be tested. If only a single key code is valid, then set `key1` and `key2` to the same value.

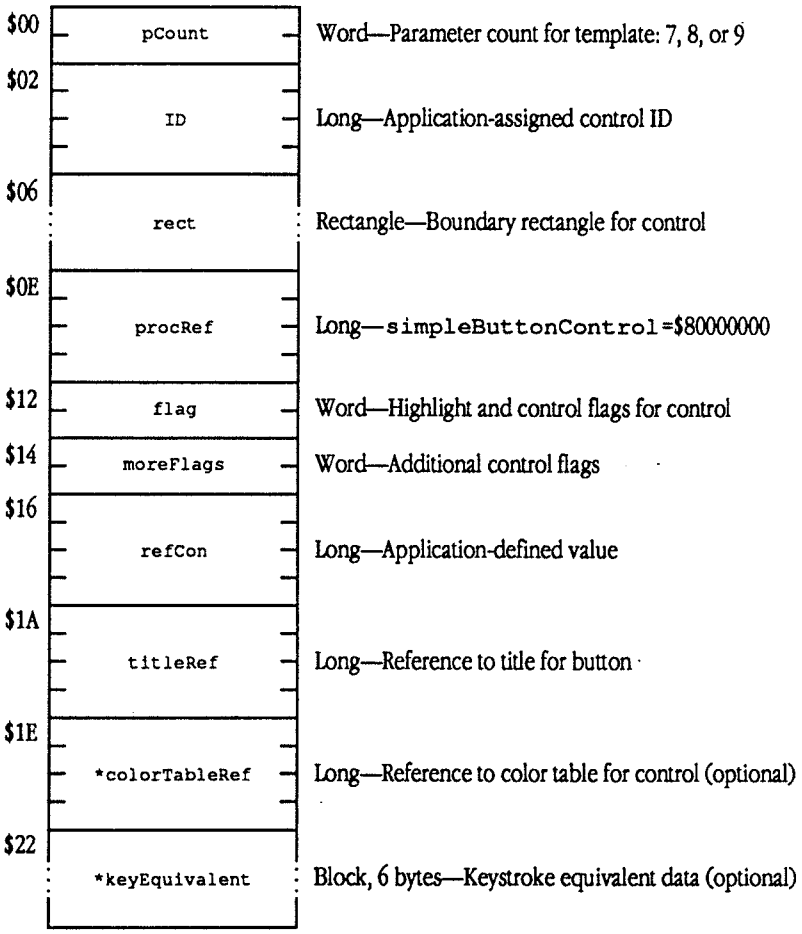
`keyModifiers` These are the modifiers that must be set to 1 in order for the equivalence test to pass. The format of this flag word corresponds to that defined for the event record in Chapter 7, "Event Manager," in Volume 1 of the *Toolbox Reference*. Note that only the modifiers in the high-order byte are used here.

`keyCareBits` These are the modifiers that must match for the equivalence test to pass. The format for this word corresponds to that for `keyModifiers`. This word allows you to discriminate between double-modified keystrokes. For example, if you want Control-7 to be an equivalent, but not Option-Control-7, you would set the *controlKey* bit in `keyModifiers` and both the *optionKey* and the *controlKey* bits in `keyCareBits` to 1. If you want Return and Enter to be treated the same, the *keyPad* bit should be set to 0.

Simple button control template

Figure E-5 shows the template that defines a simple button control.

■ **Figure E-5** Item template for simple button controls



Defined bits for `flag` are

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 2–6	Must be set to 0
Button type	bits 0–1	Describes button type: 0 = single-outlined round-cornered button 1 = bold-outlined round-cornered button 2 = single-outlined square-cornered button 3 = single-outlined square-cornered drop-shadowed button

Defined bits for `moreFlags` are

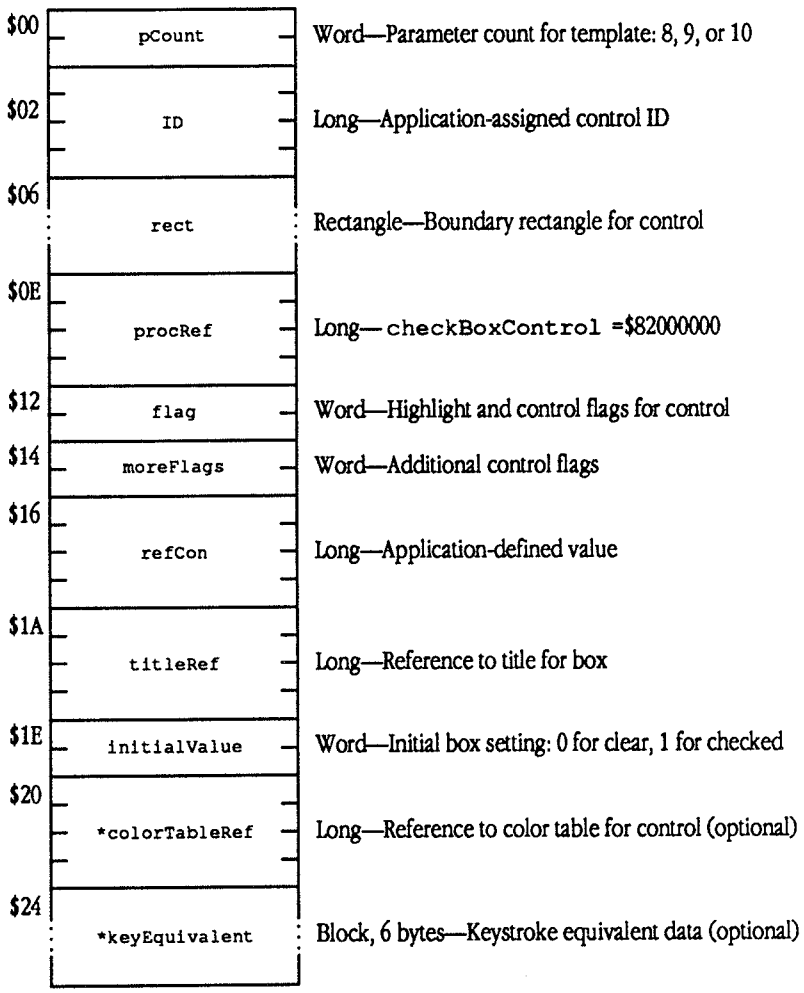
`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Set to 1 if button has keystroke equivalent
`fCtlProcNotPtr`	bit 12	Must be set to 1
fCtlTellAboutSize	bit 11	Must be set to 0
Reserved	bits 4–10	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `colorTableRef`. See Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* for the definition of the simple button color table. 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Title reference	bits 0–1	Defines type of title reference in `titleRef`: 00 - title reference is pointer 01 - title reference is handle 10 - title reference is resource ID 11 - invalid value

`keyEquivalent` Keystroke equivalent information stored at `keyEquivalent` is formatted as shown in Figure E–4.

Check box control template

Figure E-6 shows the template that defines a check box control.

Figure E-6 Control template for check box controls



Defined bits for `flag` are

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 0–6	Must be set to 0

Defined bits for `moreFlags` are

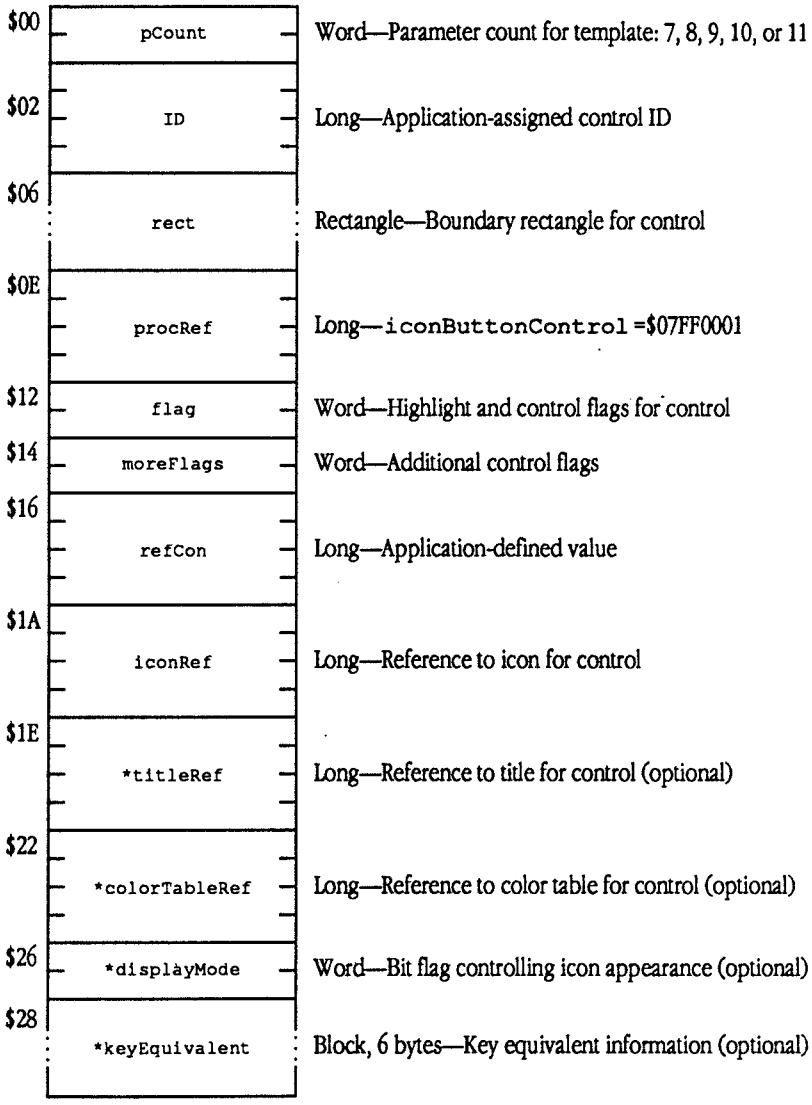
`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Set to 1 if check box has keystroke equivalent
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 4–10	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `colorTableRef` (see Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* for the definition of the check box color table) 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Title reference	bits 0–1	Defines type of title reference in `titleRef`: 00 - title reference is pointer 01 - title reference is handle 10 - title reference is resource ID 11 - invalid value

`keyEquivalent` Keystroke equivalent information stored at `keyEquivalent` is formatted as shown in Figure E-4.

Icon button control template

Figure E-7 shows the template that defines an icon button control. For more information about icon button controls, see "Icon button control" in Chapter 28, "Control Manager Update," earlier in this book.

■ **Figure E-7** Control template for icon button controls



Defined bits for `flag` are

`ctlHilite`	bits 8–15	Sets the `ctlHilite` field of the control record
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 3–6	Must be set to 0
`showBorder`	bit 2	1=No border, 0=Show border
`buttonType`	bits 0–1	Defines button type: 00 - single-outlined round-cornered button 01 - bold-outlined round-cornered button 10 - single-outlined square-cornered button 11 - single-outlined square-cornered and drop-shadowed button

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 0
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 6–10	Must be set to 0
Icon reference	bits 4–5	Defines type of icon reference in `iconRef`: 00 - icon reference is pointer 01 - icon reference is handle 10 - icon reference is resource ID 11 - invalid value
Color table reference	bits 2–3	Defines type of reference in `colorTableRef`; the color table for an icon button is the same as that for a simple button (see Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* for the definition of the simple button color table) 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Title reference	bits 0–1	Defines type of title reference in `titleRef`: 00 - title reference is pointer 01 - title reference is handle 10 - title reference is resource ID 11 - invalid value

`titleRef` Reference to the title string, which must be a Pascal string. If you are not using a title but are specifying other optional fields, set `moreFlags` bits 0 and 1 to 0, and set this field to zero.

`displayMode` Passed directly to the `DrawIcon` routine, and defines the display mode for the icon. The field is defined as follows (for more information on icons, see Chapter 17, "QuickDraw II Auxiliary," in Volume 2 of the *Toolbox Reference*):

Background Color	bits 12–15	Defines the background color to apply to black part of black-and-white icons.
Foreground Color	bits 8–11	Defines the foreground color to apply to white part of black-and-white icons.
Reserved	bits 3–7	Must be set to 0
`offLine`	bit 2	1=AND light-gray pattern to image being copied 0=Don't AND the image
`openIcon`	bit 1	1=Copy light-gray pattern instead of image 0=Don't copy light-gray pattern
`selectedIcon`	bit 0	1=Invert image before copying 0=Don't invert image

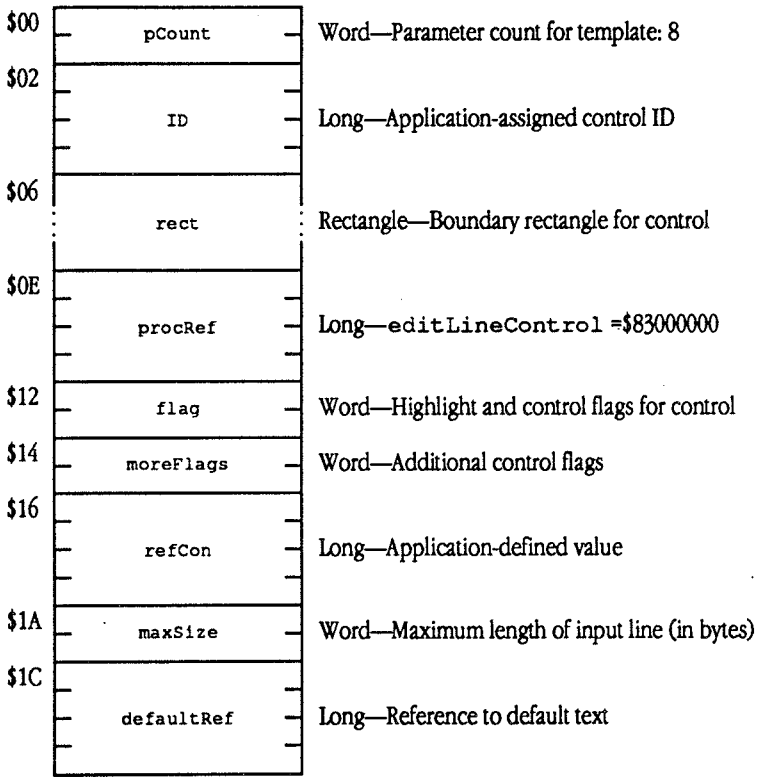
Color values (both foreground and background) are indexes into the current color table. See Chapter 16, "QuickDraw II," in Volume 2 of the *Toolbox Reference* for details about the format and content of these color tables.

`keyEquivalent` Keystroke equivalent information stored at `keyEquivalent` is formatted as shown in Figure E-4.

LineEdit control template

Figure E-8 shows the template that defines a LineEdit control. For more information about LineEdit controls, see "LineEdit control" in Chapter 28, "Control Manager Update," earlier in this book.

■ **Figure E-8** Control template for LineEdit controls



Defined bits for `flag` are

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 0–6	Must be set to 0

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 1
`fCtlWantsEvents`	bit 13	Must be set to 1
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 2–10	Must be set to 0
Text reference	bits 0–1	Defines type of text reference in `defaultRef` 00 - text reference is pointer 01 - text reference is handle 10 - text reference is resource ID 11 - invalid value

`maxSize` Specifies the maximum number of characters allowed in the LineEdit field. Valid values lie in the range from 1 to 255.

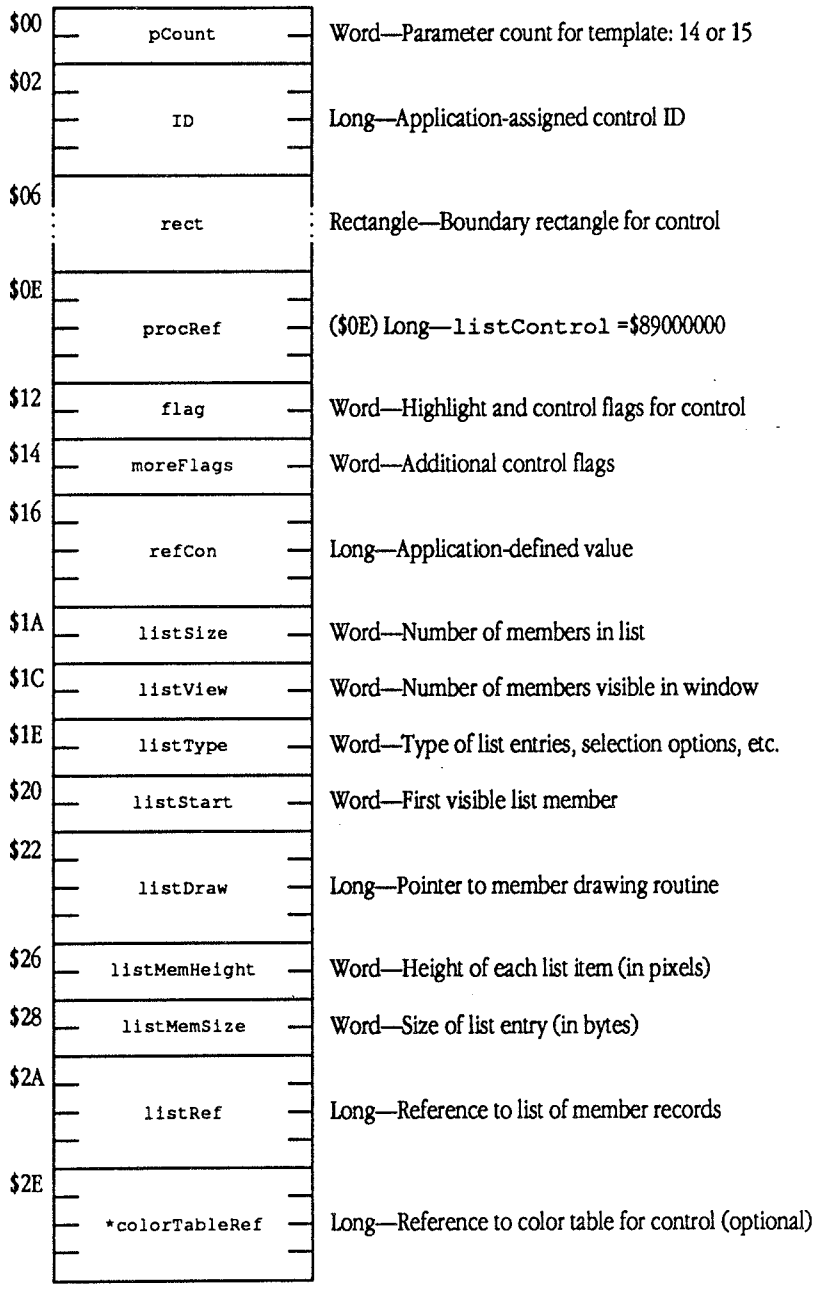
The high-order bit indicates whether the LineEdit field is a password field. Password fields protect user input by echoing asterisks, rather than the actual user input. If this bit is set to 1, then the LineEdit field is a password field.

Note that LineEdit controls do not support color tables.

List control template

Figure E-9 shows the template that defines a list control. For more information about list controls, see "List control" in Chapter 28, "Control Manager Update," earlier in this book.

■ **Figure E-9** Control template for list controls



Defined bits for `flag` are

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 0–6	Must be set to 0

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 0
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
`fCtlIsMultiPart`	bit 10	Must be set to 1
Reserved	bits 4–9	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `colorTableRef` (the color table for a List control is described in Chapter 11, "List Manager," in Volume 1 of the *Toolbox Reference*) 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
List reference	bits 0–1	Defines type of reference in `listRef` (the format for a list member record is described in Chapter 11, "List Manager," in Volume 1 of the *Toolbox Reference*) 00 - list reference is pointer 01 - list reference is handle 10 - list reference is resource ID 11 - invalid value

`listType` Valid values for `listType` are as follows:

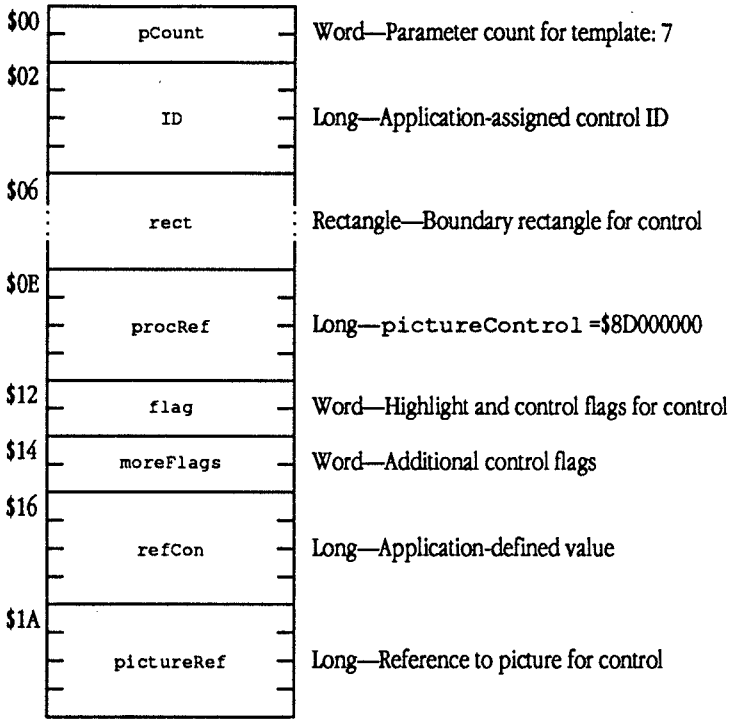
Reserved	bits3–15	Must be set to 0.
`fListScrollBar`	bit 2	Allows you to control where the scroll bar for the list is drawn: 1 - Scroll bar drawn on inside of boundary rectangle. The List Manager calculates space needed, adjusts dimensions of boundary rectangle, and resets this flag. 0 - Scroll bar drawn on outside of boundary rectangle.
`fListSelect`	bit 1	Controls type of selection options available to the user: 1 - Only single selection allowed 0 - Arbitrary and range selection allowed
`fListString`	bit 0	Defines the type of strings used to define list items: 1 - C-strings ($00-terminated) 0 - Pascal strings

For details on the remaining custom fields in this template, see the discussion of "List Controls and List Records" in Chapter 11, "List Manager," of Volume 1 of the *Toolbox Reference.*

Picture control template

Figure E-10 shows the template that defines a picture control. For more information about picture controls, see "Picture control" in Chapter 28, "Control Manager Update," earlier in this book.

■ **Figure E-10** Control template for picture controls



Defined bits for `flag` are

`ctlHilite`	bits 8–15	Specifies whether the control wants to receive mouse selection events; the values for `ctlHilite` are as follows: 0 Control is active 255 Control is inactive
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 0–6	Must be set to 0

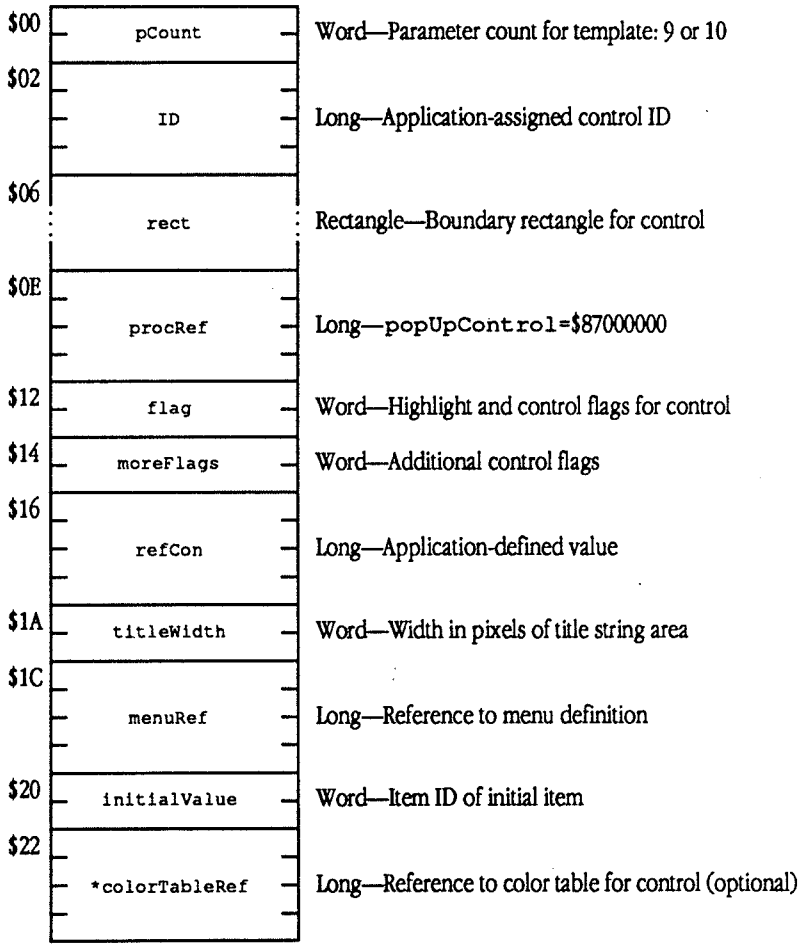
Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 0
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`		bit 11 Must be set to 0
Reserved	bits 2–10	Must be set to 0
Picture reference	bits 0–1	Define type of picture reference in `pictureRef`: 00 - invalid value 01 - reference is handle 10 - reference is resource ID 11 - invalid value

Pop-up control template

Figure E-11 shows the template that defines a pop-up control. For more information about pop-up controls, see "Pop-up control" in Chapter 28, "Control Manager Update," earlier in this book.

■ **Figure E-11** Control template for pop-up controls



Defined bits for `flag` are

`ctlHilite`	bits 8–15	Specifies whether the control wants to receive mouse selection events; the values for `ctlHilite` are as follows: 0 Control is active 255 Control is inactive
`ctlInvis`	bit 7	1=invisible, 0=visible
`fType2PopUp`	bit 6	Tells the Control Manager whether to create a pop-up menu with white space for scrolling (see Chapter 37, "Menu Manager Update," for details on Type 2 pop-up menus): 1 - Draw pop-up with white space (Type 2) 0 - Draw normal pop-up
`fDontHiliteTitle`	bit 5	Controls highlighting of the control title: 1 - Do not highlight title when control is popped up 0 - Highlight title
`fDontDrawTitle`	bit 4	Allows you to prevent the title from being drawn (note that you must supply a title in the menu definition, whether or not it will be displayed); if `titleWidth` is defined and this bit is set to 1, then the entire menu is offset to the right by `titleWidth` pixels: 1 - Do not draw the title 0 - Draw the title
`fDontDrawResult`	bit 3	Allows you to control whether the selection is drawn in the pop-up rectangle: 1 - Do not draw the result in the result area after a selection 0 - Draw the result
`fInWindowOnly`	bit 2	Controls the extent to which the pop-up menu can grow; this is particularly relevant to Type 2 pop-ups (see Chapter 37, "Menu Manager Update," for details on Type 2 pop-up menus): 1 - Keep the pop-up in the current window 0 - Allow the pop-up to grow to screen size
`fRightJustifyTitle`		

	bit 1	Controls title justification: 1 - Right justify the title; note that if the title is right justified, then the control rectangle is adjusted to eliminate unneeded pixels, the value for `titleWidth` is also adjusted 0 - Left justify the title
`fRightJustifyResult`	bit 0	Controls result justification: 1 - Right justify the selection 0 - Left justify the selection `titleWidth` pixels from the left of the pop-up rectangle.

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 1 if the pop-up has any keystroke equivalents defined
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 5–10	Must be set to 0
Color table reference	bits 3–4	Defines type of reference in `colorTableRef` (the color table for a menu is described in Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference*) 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
`fMenuDefIsText`	bit 2	Defines type of data referred to by `menuRef`: 1 - `menuRef` is a pointer to a text stream in `NewMenu` format (see Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference* for details) 0 - `menuRef` is a reference to a Menu Template (again, see Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference* for details on format and content of a Menu Template)

Menu reference	bits 0–1	Defines type of menu reference in `menuRef` (if `fMenuDefIsText` is set to 1, then these bits are ignored): 00 - menu reference is pointer 01 - menu reference is handle 10 - menu reference is resource ID 11 - invalid value

`rect` Defines the boundary rectangle for the pop-up and its title, before the menu has been "popped" by the user. The Menu Manager will calculate the lower, right coordinates of the rectangle for you, if you specify those coordinates as (0,0).

`initialValue` The initial value to be displayed for the menu. The initial value is the default value for the menu, and is displayed in the pop-up rectangle of "unpopped" menus. You specify an item by its ID, that is, its relative position within the array of items for the menu (see Chapter 37, "Menu Manager Update," for information on the layout and content of the pop-up menu template). If you pass an invalid item ID then no item is displayed in the pop-up rectangle.

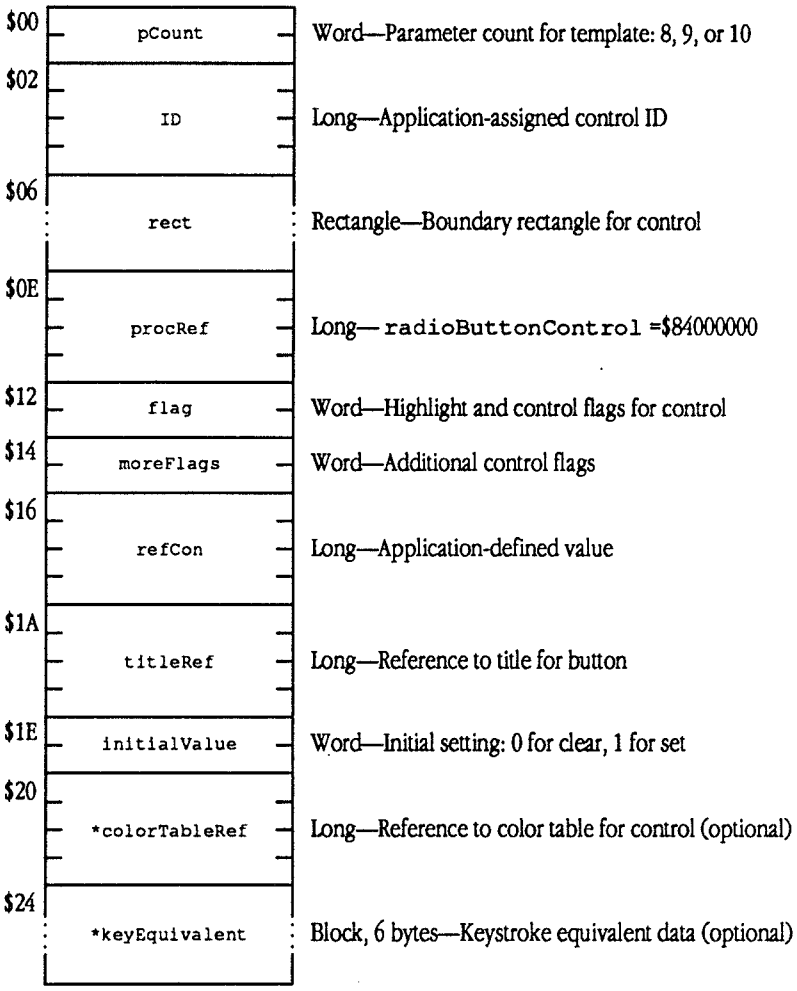
`titleWidth` Provides you with additional control over placement of the menu on the screen. The `titleWidth` field defines an offset from the left edge of the control (boundary) rectangle to the left edge of the pop-up rectangle. If you are creating a series of pop-up menus and you want them to be vertically aligned, you can do this by giving all menus the same x1 coordinate and `titleWidth` value. You may use `titleWidth` for this even if you are not going to display the title (`fDontDrawTitle` flag is set to 1 in `flag`). If you set `titleWidth` to 0, then the Menu Manager determines its value based upon the length of the menu title, and the pop-up rectangle immediately follows the title string. If the actual width of your title exceeds the value of `titleWidth`, results are unpredictable.

`menuRef` Reference to menu definition (see Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference* and Chapter 37, "Menu Manager Update," in this book for details on menu templates). The type of reference contained in `menuRef` is defined by the menu reference bits in `moreFlags`.

Radio button control template

Figure E-12 shows the template that defines a radio button control:

■ **Figure E-12** Control template for radio button controls



Defined bits for `flag` are

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible

Family number	bits 0–6	Family numbers define associated groups of radio buttons; radio buttons in the same family are logically linked, that is, setting one radio button in a family clears all other buttons in the same family

Defined bits for `moreFlags` are as follows:

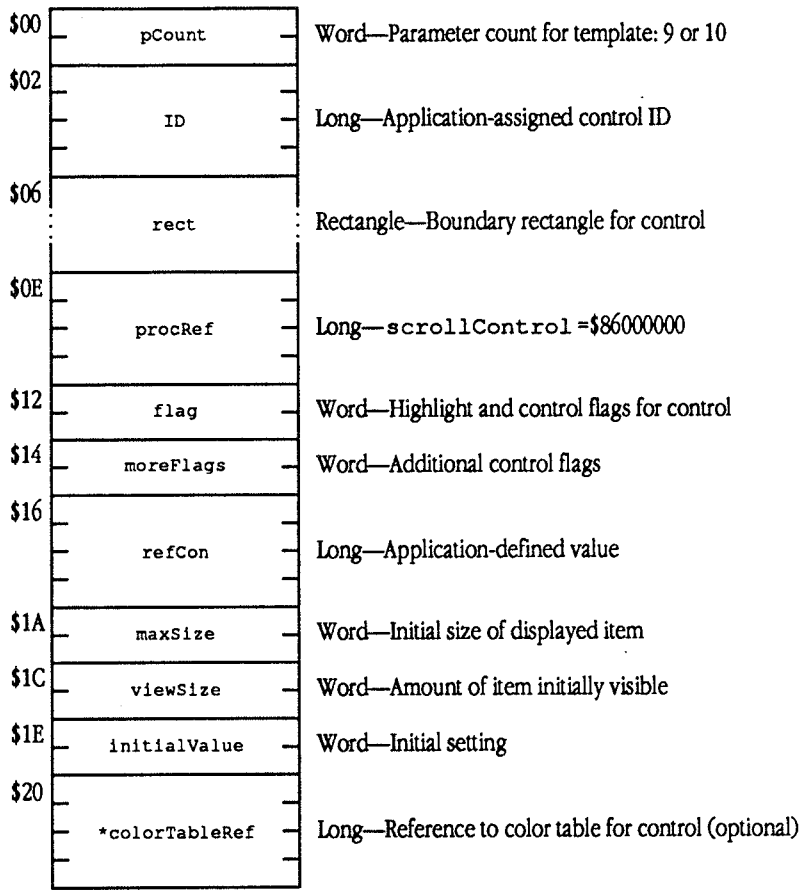
`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Set to 1 if button has keystroke equivalent
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 4–10	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `colorTableRef` (see Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* for the definition of the radio button color table) 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Title reference	bits 0–1	Defines type of title reference in `titleRef`: 00 - title reference is pointer 01 - title reference is handle 10 - title reference is resource ID 11 - invalid value

`keyEquivalent` Keystroke equivalent information stored at `keyEquivalent` is formatted as shown in Figure E-4.

Scroll bar control template

Figure E-13 shows the template that defines a scroll bar control:

■ **Figure E-13** Control template for scroll bar controls



Defined bits for `flag` are

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 5–6	Must be set to 0
`horScroll`	bit 4	1=horizontal scroll bar, 0=vertical scroll bar
`rightFlag`	bit 3	1=bar has right arrow, 0=bar has no right arrow
`leftFlag`	bit 2	1=bar has left arrow, 0=bar has no left arrow
`downFlag`	bit 1	1=bar has down arrow, 0=bar has no down arrow
`upFlag`	bit 0	1=bar has up arrow, 0=bar has no up arrow

Note that extraneous flag bits are ignored, based upon state of `horScroll` flag. For example, for vertical scroll bars, `rightFlag` and `leftFlag` are ignored.

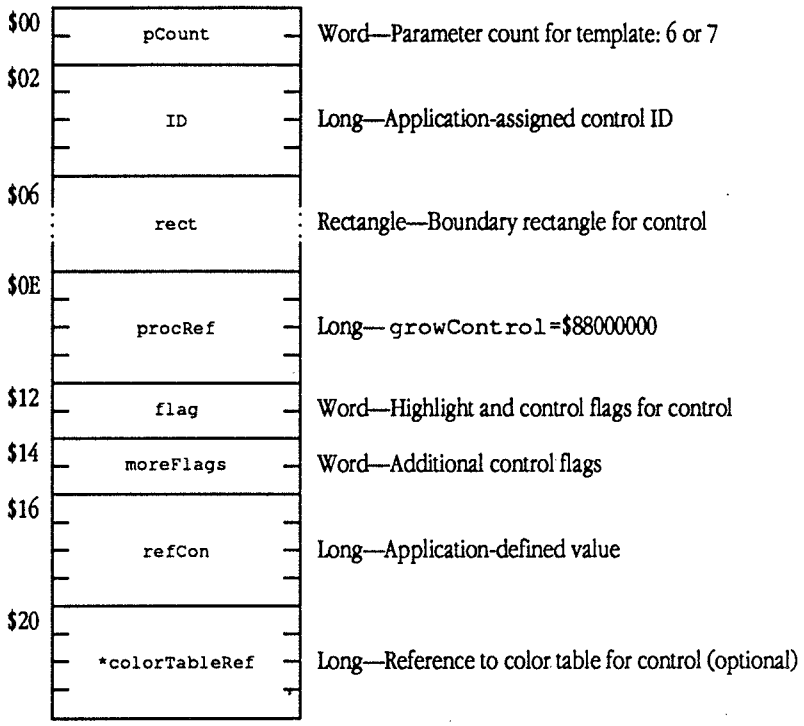
Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 0
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 4–10	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in *colorTableRef* (see Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* and "Clarifications" in Chapter 28, "Control Manager Update," earlier in this book for the definition of the scroll bar color table) 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Reserved	bits 0–1	Must be set to 0

Size box control template

Figure E-14 shows the template that defines a size box control:

■ **Figure E-14** Control template for size box controls



Defined bits for `flag` are

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 1–6	Must be set to 0
`fCallWindowMgr`	bit 0	1=call `GrowWindow` and `SizeWindow` to track this control 0=just highlight control

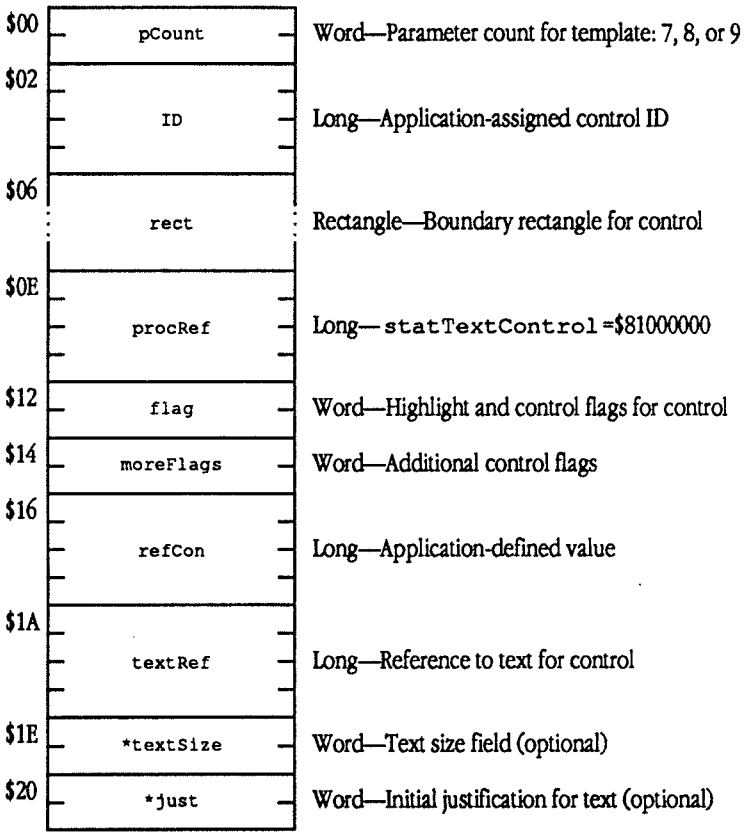
Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 0
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 4–10	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `colorTableRef` (see "Error Corrections" in Chapter 28, "Control Manager Update," earlier in this book for the definition of the size box color table) 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Reserved	bits 0–1	Must be set to 0

Static text control template

Figure E-15 shows the template that defines a static text control. For more information about static text controls, see "Static text control" in Chapter 28, "Control Manager Update," earlier in this book.

■ **Figure E-15** Control template for static text controls



Defined bits for `flag` are

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 2–6	Must be set to 0
`fSubstituteText`	bit 1	0=no text substitution to perform 1=there is text substitution to perform
`fSubTextType`	bit 0	0=C strings 1=Pascal strings

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 0
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 2–10	Must be set to 0
Text Reference	bits 0–1	Defines type of text reference in `textRef`: 00 - text reference is pointer 01 - text reference is handle 10 - text reference is resource ID 11 - invalid value

`textSize` The size of the referenced text in characters, but only if the text reference in `textRef` is a pointer. If the text reference is either a handle or a resource ID, then the Control Manager can extract the length from the handle.

`just` The justification word is passed on to `LETextBox2` (see Chapter 10, "LineEdit Tool Set," in Volume 1 of the *Toolbox Reference* for details on the `LETextBox2` tool call), and is used to set the initial justification for the text being drawn. Valid values for `just` are

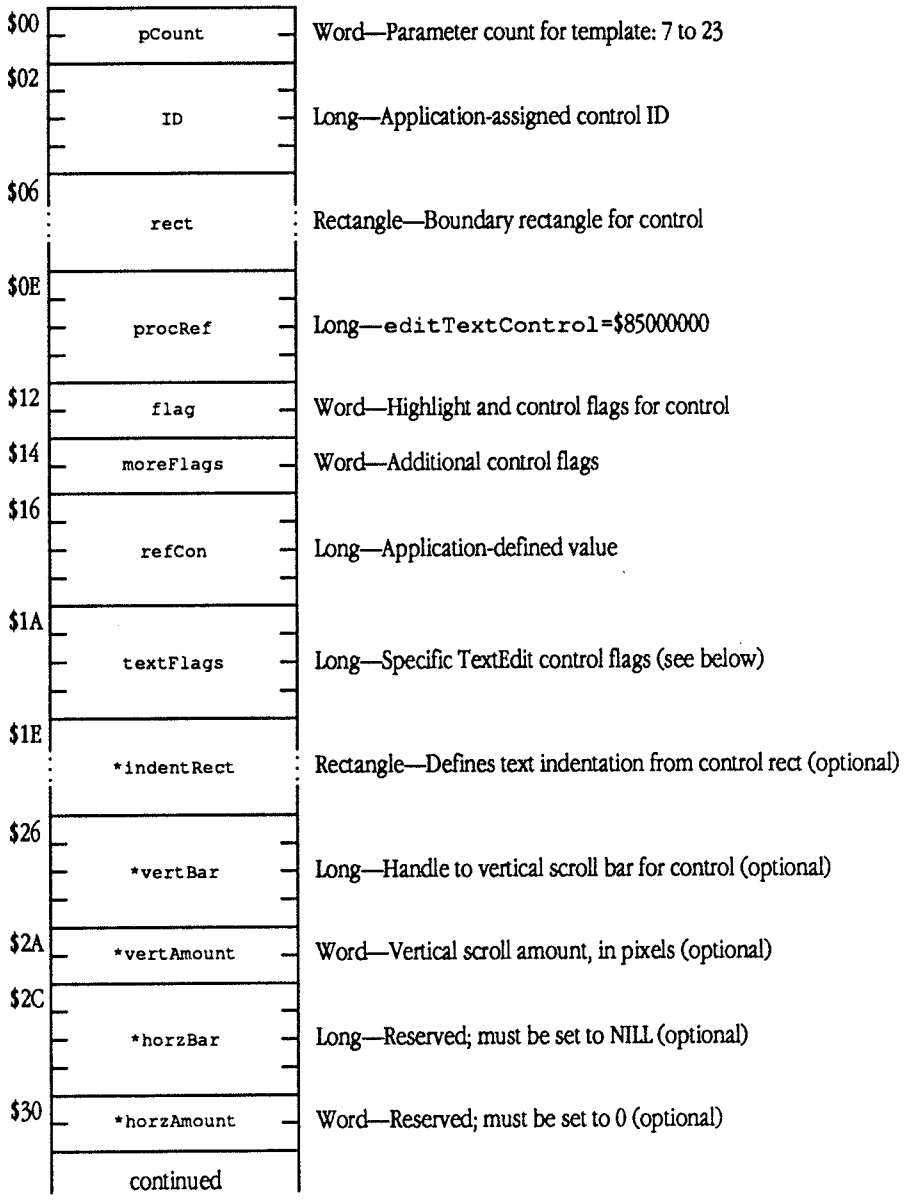
`leftJustify`	0	Text is left justified in the display window
`centerJustify`	1	Text is centered in the display window
`rightJustify`	-1	Text is right justified in the display window
`fullJustify`	2	Text is fully justified (both left and right) in the display window

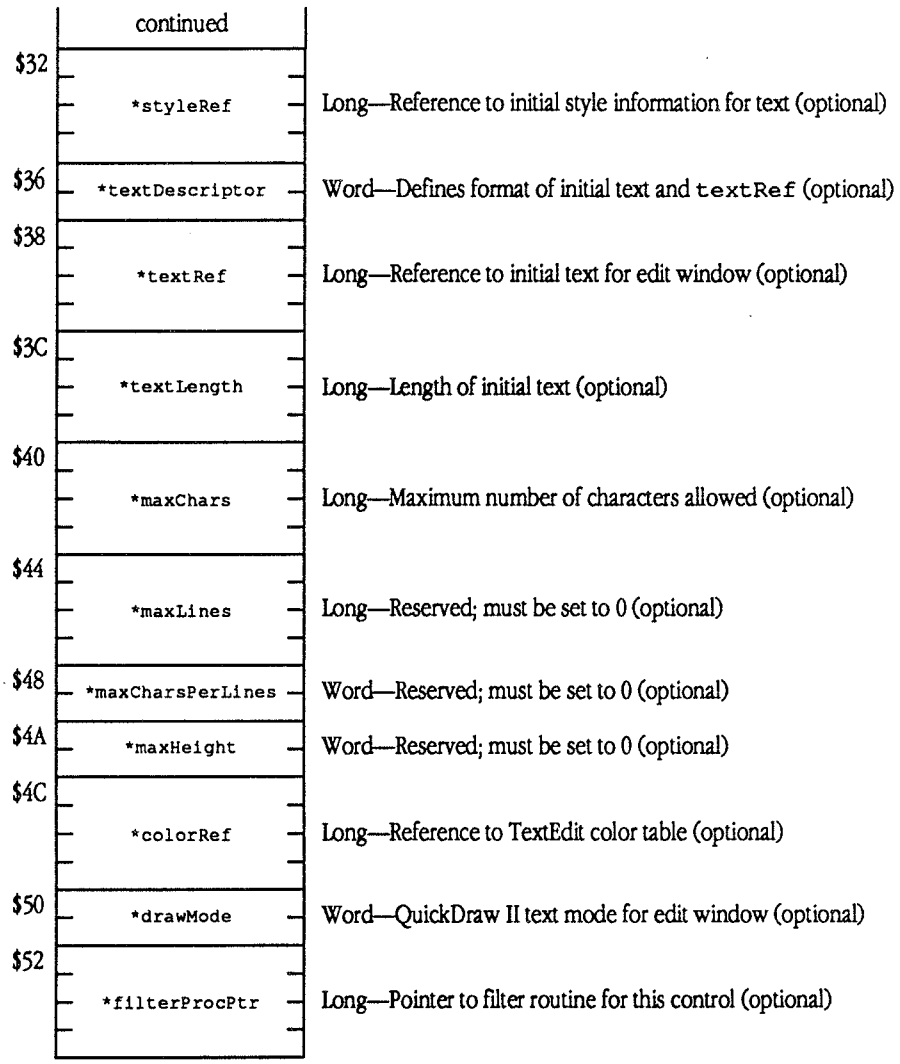
Static text controls do not support color tables. In order to display text of different color, you must embed the appropriate commands into the text string you are displaying. See the discussion of `LETextBox2` in Chapter 10, "LineEdit Tool Set," in Volume 1 of the *Toolbox Reference* for details on command format and syntax.

TextEdit control template

Figure E-16 shows the template that defines a TextEdit control. For more information about TextEdit controls, see "TextEdit control" in Chapter 28, "Control Manager Update," earlier in this book.

■ **Figure E-16** Control template for TextEdit controls





Defined bits for `flag` are

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 0–6	Must be set to 0

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 1
`fCtlWantsEvents`	bit 13	Must be set to 1
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fTellAboutSize`	bit 11	If set to 1, a size box will be created in the lower-right corner of the window. Whenever the control window is resized, the edit text will be resized and redrawn.
`fCtlIsMultiPart`	bit 10	Must be set to 1
Reserved	bits 4–9	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `colorRef`; the color table for a TextEdit control (`TEColorTable`) is described in Chapter 49, "TextEdit," in this book: 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Style reference	bits 0–1	Defines type of style reference in `styleRef`; the format for a TextEdit style descriptor is described in Chapter 49, "TextEdit," in this book: 00 - style reference is pointer 01 - style reference is handle 10 - style reference is resource ID 11 - invalid value

△ **Important** Do not set `fTellAboutSize` to 1 unless the control also has a vertical scroll bar. △

Valid values for `textFlags` are

`fNotControl`	bit 31	Must be set to 0
`fSingleFormat`	bit 30	Must be set to 1
`fSingleStyle`	bit 29	Allows you to restrict the style options available to the user: 1 - Allow only one style in the text 0 - Do not restrict the number of styles in the text
`fNoWordWrap`	bit 28	Allows you to control TextEdit word wrap behavior: 1 - Do not word wrap the text; only break lines on CR ($0D) characters 0 - Perform word wrap to fit the ruler

`fNoScroll`	bit 27	Controls user access to scrolling: 1 - Do not allow either manual or auto-scrolling 0 - Scrolling permitted
`fReadOnly`	bit 26	Restricts the text in the window to read-only operations (copying from the window will still be allowed): 1 - No editing allowed 0 - Editing permitted
`fSmartCutPaste`	bit 25	Controls TextEdit support for smart cut and paste (see Chapter 49, "TextEdit," for details on smart cut and paste support): 1 - Use smart cut and paste 0 - Do not use smart cut and paste
`fTabSwitch`	bit 24	Defines behavior of the Tab key (see Chapter 49, "TextEdit," for details): 1 - Tab to next control in the window 0 - Tab inserted in TextEdit document
`fDrawBounds`	bit 23	Tells TextEdit whether to draw a box around the edit window, just inside `rect`; the pen for this box is two pixels wide and one pixel high 1 - Draw rectangle 0 - Do not draw rectangle
`fColorHilight`	bit 22	Must be set to 0.
`fGrowRuler`	bit 21	Tells TextEdit whether to resize the ruler in response to the user resizing the edit window; if set to 1, TextEdit will automatically adjust the right margin value for the ruler: 1 - Resize the ruler 0 - Do not resize the ruler
`fDisableSelection`	bit 20	Controls whether user can select text: 1 - User cannot select text 0 - User can select text
`fDrawInactiveSelection`	bit 19	Controls how inactive selected text is displayed: 1 - TextEdit draws a box around inactive selections 0 - TextEdit does not display inactive selections
Reserved	bits 0–18	Must be set to 0

`indentRect` Each coordinate of this rectangle specifies the amount of white space to leave between the boundary rectangle for the control and the text itself, in pixels. Default values are (2,6,2,4) in 640 mode and (2,4,2,2) in 320 mode. Each indentation coordinate may be specified individually. In order to assert the default for any coordinate, specify its value as $FFFF.

`vertBar` Handle of the vertical scroll bar to use for the TextEdit window. If you do not want a scroll bar at all, then set this field to NIL. If you want TextEdit to create a scroll bar for you, just inside the right edge of the boundary rectangle for the control, then set this field to $FFFFFFFF.

`vertAmount` Specifies the number of pixels to scroll whenever the user presses the up or down arrow on the vertical scroll bar. In order to use the default value (9 pixels), set this field to $0000.

`horzBar` Must be set to NIL.

`horzAmount` Must be set to 0.

`styleRef` Reference to initial style information for the text. See the description of the `TEFormat` record in Chapter 49, "TextEdit," for information about the format and content of a style descriptor. Bits 1 and 0 of `moreFlags` define the type of reference (pointer, handle, resource ID). To use the default style and ruler information, set this field to NULL.

`textDescriptor` Input text descriptor that defines the reference type for the initial text (which is in `textRef`) and the format of that text. See Chapter 49, "TextEdit," for detailed information on text and reference formats.

`textRef` Reference to initial text for the edit window. If you are not supplying any initial text, then set this field to NULL.

`textLength` If `textRef` is a pointer to the initial text, then this field must contain the length of the initial text. For other reference types, TextEdit extracts the length from the reference itself.

◆ *Note:* You must specify or omit the `textDescriptor`, `textRef`, and `textLength` fields as a group.

`maxChars` Maximum number of characters allowed in the text. If you do not want to define any limit to the number of characters, then set this field to NULL.

`maxLines` Must be set to 0.

`maxCharsPerLines` Must be set to NULL.

`maxHeight` Must be set to 0.

`colorRef` Reference to the color table for the text. This is a Text Edit color table (see Chapter 49, "TextEdit," for format and content of `TEColorTable`). Bits 2 and 3 of `moreFlags` define the type of reference stored here.

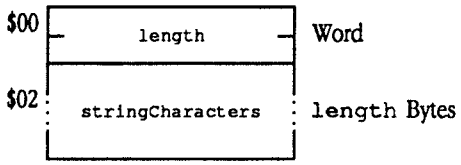
`drawMode` This is the text mode used by QuickDraw II for drawing text. See Chapter 16, "QuickDraw II," in Volume 2 of the *Toolbox Reference* for details on valid text modes.

`filterProcPtr` Pointer to a filter routine for the control. See Chapter 49, "TextEdit," for details on TextEdit generic filter routines. If you do not want to use a filter routine for the control, set this field to NIL.

`rC1InputString` $8005

Figure E-17 defines the layout of resource type `rC1InputString` ($8005). Resources of this type contain GS/OS Class 1 input strings (length word followed by data).

■ **Figure E–17** GS/OS class1 input string, type `rC1InputString` ($8005)



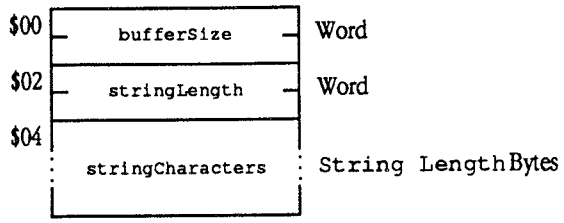
`length` Indicates the number of bytes stored at `stringCharacters`. This is an unsigned integer; valid values lie in the range from 1 to 65535.

`stringCharacters`
Array of `length` characters.

`rC1OutputString` $8023

Figure E-18 defines the layout of resource type `rC1OutputString` ($8023). Resources of this type contain GS/OS class 1 output strings (buffer size word and string length word followed by data).

■ **Figure E–18** GS/OS class1 output string, type `rC1OutputString` ($8023)



`bufferSize` Indicates the number of bytes in the entire structure, including `bufferSize`.

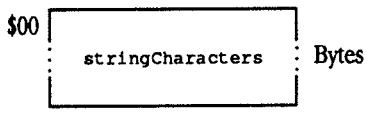
`stringLength` Indicates the number of bytes stored at `stringCharacters`. This is an unsigned integer; valid values lie in the range from 1 to 65535. If the returned string will not fit in the buffer, this field indicates the length of the string the system wants to return. Your program should add four to that value (to account for `bufferSize` and `stringLength`, resize the buffer, and reissue the call).

`stringCharacters` Array of `stringLength` characters.

`rCString` $801D

Figure E-19 defines the layout of resource type `rCString` ($801D). Resources of this type contain C strings (null-terminated character array).

■ **Figure E–19** C string, type `rCString` ($801D)

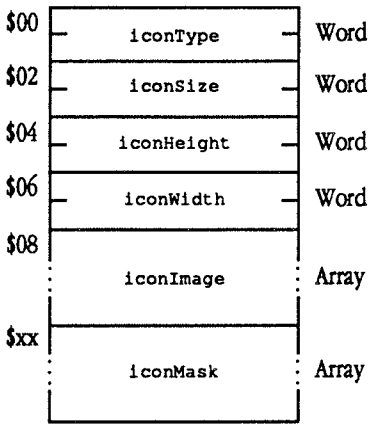


`stringCharacters`
Array of characters; last character must be a null ($00). The string may contain up to 65,535 characters, including the null terminator.

rIcon $8001

Figure E-20 defines the layout of resource type `rIcon` ($8001).

■ **Figure E–20** Icon, type `rIcon` ($8001)

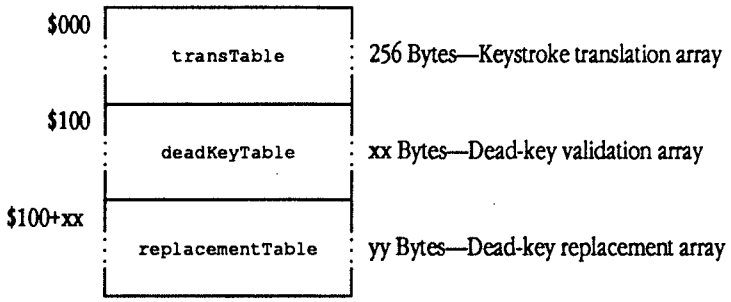


`iconType`	Contains flags defining the type of icon stored in the icon record.	
Color Indicator	Bit 15	Indicates whether the icon contains a color or black-and-white image 1 - Icon is color 0 - Icon is black and white
`iconSize`	Specifies the size of the icon image stored at `iconImage`, in bytes.	
`iconHeight`	Specifies the height of the icon, in pixels.	
`iconWidth`	Specifies the width of the icon, in pixels.	
`iconImage`	Contains `iconSize` bytes of icon image data.	
`iconMask`	Contains `iconSize` bytes of mask data to be applied to the image located at `iconImage`.	

`rKTransTable` $8021

Figure E-21 defines the layout of resource type `rKTransTable` ($8021). Resources of this type define keystroke translation tables for use by the Event Manager (see Chapter 31, "Event Manager Update," earlier in this book for complete information on the format and content of resources of this type).

■ **Figure E–21** Keystroke translation table, type `rKTransTable` ($8021)



`transTable` This is a packed array of bytes used to map the ASCII codes produced by the keyboard into the character value to be generated. Each cell in the array directly corresponds to the ASCII code that is equivalent to the cell offset. For example, the `transTable` cell at offset $0D (13 decimal) contains the character replacement value for keyboard code $0D, which, for a straight ASCII translation table, is a Return character (CR). The `transTable` cells from 128 to 255 ($80 to $FF) contain values for Option-key sequences (such as Option-S).

`deadKeyTable` This table contains entries used to validate dead keys. Dead key refers to keystrokes used to introduce multikey sequences that result in single characters. For example, pressing Option-u followed by e yields an e with an umlaut. There is one entry in `deadKeyTable` for each defined dead key. The last entry must be set to $0000. Each entry must be formatted as follows:

`deadKey`	Byte—Character code for dead key
`offset`	Byte—Offset from `deadKeyTable` into `replacementTable`

`deadKey` Contains the character code for the dead key. The system uses this value to check for user input of a dead key. The system compares this value with the first user keystroke.

`offset` Byte offset from beginning of `deadKeyTable` into relevant subarray in `replacementTable`, divided by 2. The system uses this value to access the valid replacement values for the dead key in question.

`replacementTable`
This table contains the valid replacement values for each dead key combination. This table is made up of a series of variable-length subarrays, each relevant to a particular dead key. The last entry in each sub-array must be set to $0000. Each entry in the `replacementTable` must be formatted as follows:

`deadKey`	Byte—Character code for dead key
`offset`	Byte—Offset from `deadKeyTable` into `replacementTable`

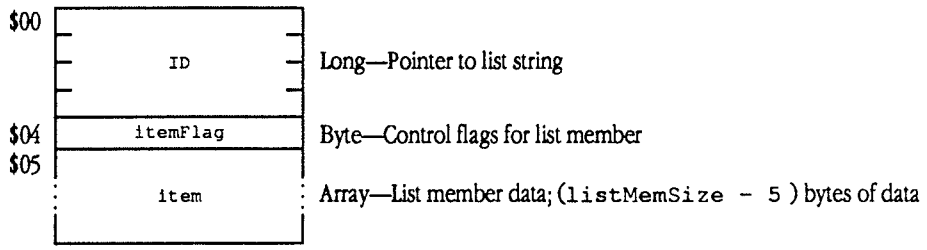
`scanKey` Contains a valid character code for dead key replacement. The system uses this field to determine whether the user entered a valid dead key combination. The system compares this value with the second user keystroke.

`replaceValue` Contains the replacement value for the character specified in `scanKey` for this entry. The system delivers this value as the replacement for a valid dead key combination.

`rListRef` $801C

Figure E-22 defines the layout of the array element that comprises resource type `rListRef` ($801C). Resources of this type define members of list controls (see Chapter 28, "Control Manager Update," earlier in this book for more information on list controls). A single `rListRef` resource may contain more than one of these elements; you concatenate the elements to form the resource.

■ **Figure E–22** List member reference array element, type `rListRef` ($801C)



`itemPtr` Pointer to the list member string.

`itemFlag` Control flags for the member.

`memSelect` Bits 6–7 Indicates whether the item is selected
00 - Item is enabled but not selected
01 - Item is disabled (cannot be selected)
10 - Item is selected
11 - Invalid value

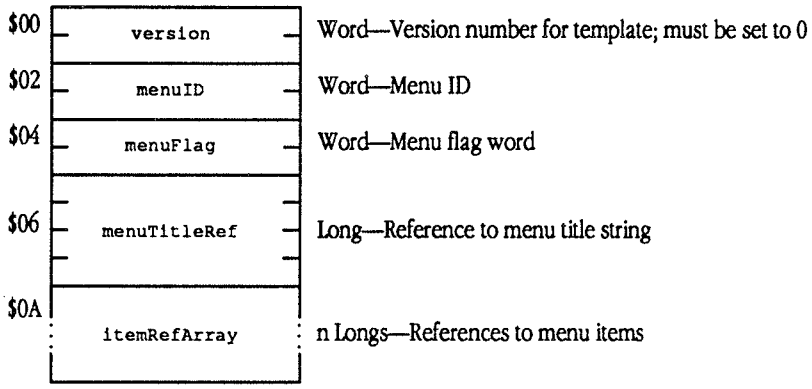
Reserved Bits 0–5 Must be set to 0

`item` Application-specific data for the list member. The `listMemSize` field of the list control template specifies the size of this field, plus 5. For example, in order to assign a two-byte tag to each list member, you would set `listMemSize` to 7 (5+2) and place the tag value at `item` in each list member.

rMenu $8009

Figure E-23 defines the layout of resource type `rMenu` ($8009). Resources of this type define parameters to some new Menu Manager tool calls. See Chapter 37, "Menu Manager Update," earlier in this book for more information.

■ **Figure E–23** Menu template, type `rMenu` ($8009)



`version` Identifies the version of the menu template. The Menu Manager uses this field to distinguish between different revisions of the template. Must be set to 0.

`menuID` Unique identifier for the menu. See Chapter 13, "Menu Manager," in the *Toolbox Reference* for information on valid values for `menuID`.

`menuFlag` Bit flags controlling the display and processing attributes of the menu. Valid values for `menuFlag` are:

`titleRefType`	bits 14–15	Defines the type of reference in `menuTitleRef`: 00 - Reference is pointer 01 - Reference is handle 10 - Reference is resource ID 11 - Invalid value
`itemRefType`	bits 12–13	Defines the type of reference in each entry of `itemRefArray` (all array entries must be of the same type): 00 - References are pointers 01 - References are handles 10 - References are resource IDs 11 - Invalid value
Reserved	bits 9–11	Must be set to 0
`alwaysCallmChoose`	bit 8	Causes the Menu Manager to call a custom menu defProc mChoose routine even when the mouse is not in the menu rectangle (supported tear-off menus): 0 - Do not always call mChoose routine 1 - Always call mChoose routine
`disabled`	bit 7	Enables or disables the menu: 0 - Menu enabled 1 - Menu disabled
Reserved	bit 6	Must be set to 0
`XOR`	bit 5	Controls how selection highlighting is performed: 0 - Do not use XOR to highlight 1 - Use XOR to highlight item
`custom`	bit 4	Indicates whether custom or standard menu: 0 - Standard menu 1 - Custom menu
`allowCache`	bit 3	Controls menu caching: 0 - Do not cache menu 1 - Menu caching allowed
Reserved	bits 0–2	Must be set to 0

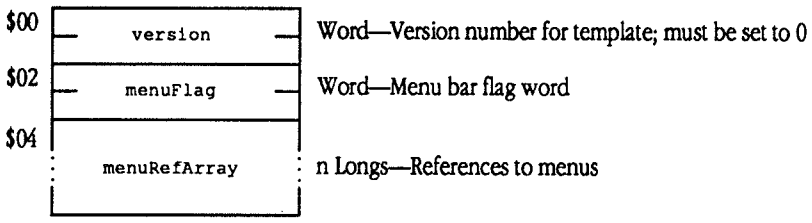
`menuTitleRef` Reference to title string for menu. The `titleRefType` bits in `menuFlag` indicate whether `menuTitleRef` contains a pointer, a handle, or a resource ID. If `menuTitleRef` is a pointer, then the title string must be a Pascal string. Otherwise, the Menu Manager can retrieve the string length from control information in the handle.

`itemRefArray` Array of references to the menu items for the menu. The `itemRefType` bits in `menuFlag` indicate whether the entries in the array are pointers, handles, or resource IDs. Note that all array entries must contain the same reference type. The last entry in the array must be set to $00000000.

rMenuBar $8008

Figure E-24 defines the layout of resource type `rMenuBar` ($8008). Resources of this type define the characteristics of a menu bar for new Menu Manager tool calls. For more information, see Chapter 37, "Menu Manager Update," earlier in this book.

■ **Figure E-24** Menu bar record, type `rMenuBar` ($8008)



`version` Identifies the version of the menu bar template. The Menu Manager uses this field to distinguish between different revisions of the template. Must be set to 0.

`menuBarFlag` Bit flags controlling the display and processing attributes of the menu bar. Valid values for `menuBarFlag` are:

`menuRefType` bits 14–15 Defines the type of reference in each entry of `menuRefArray` (all array entries must be of the same type):
00 - References are pointers
01 - References are handles
10 - References are resource IDs
11 - Invalid value

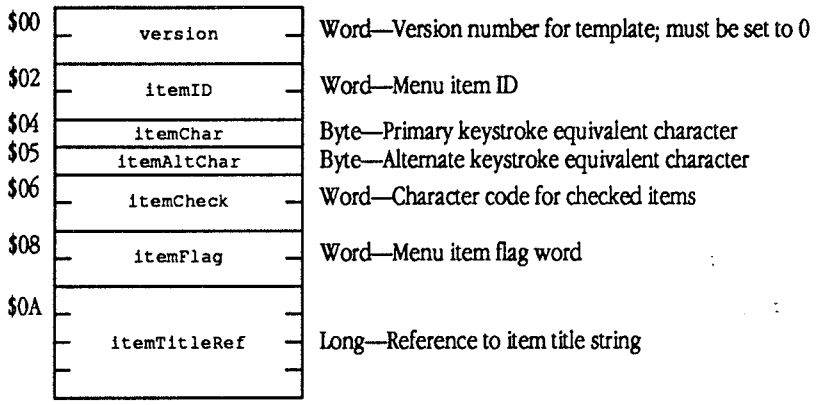
Reserved bits 0–13 Must be set to 0

`menuRefArray` Array of references to the menus for the menu bar. The `menuRefType` bits in `menuBarFlag` indicate whether the entries in the array are pointers, handles, or resource IDs. Note that all array entries must contain the same reference type. The last entry in the array must be set to $00000000.

rMenuItem $800A

Figure E-25 defines the layout of resource type `rMenuItem` ($800A). Resources of this type define menu items to some new Menu Manager tool calls. See Chapter 37, "Menu Manager Update," earlier in this book for more information.

■ **Figure E-25** Menu item template, type `rMenuItem` ($800A)



`version` Identifies the version of the menu item template. The Menu Manager uses this field to distinguish between different revisions of the menu item template. Must be set to 0.

`itemID` Unique identifier for the menu item. See Chapter 13, "Menu Manager," in the *Toolbox Reference* for information on valid values for `itemID`.

`itemChar`, `itemAltChar`
These fields define the keystroke equivalents for the menu item. The user can select the menu item by pressing the Command key along with the key corresponding to one of these fields. Typically, these fields contain the upper and lower case ASCII codes for a particular character. If you only have a single key equivalence, set both fields with that value.

`itemCheck` Defines the character to be displayed next to the item when it is checked.

`itemFlag` Bit flags controlling the display attributes of the menu item. Valid values for `itemFlag` are:

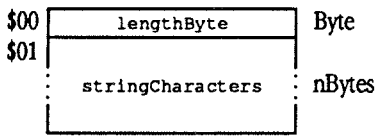
`titleRefType`	bits 14–15	Defines the type of reference in `itemTitleRef`: 00 - Reference is pointer 01 - Reference is handle 10 - Reference is resource ID 11 - Invalid value
Reserved	bit 13	Must be set to 0
`shadow`	bit 12	Indicates item shadowing: 0 - No shadow 1 - Shadow
`outline`	bit 11	Indicates item outlining 0 - Not outlined 1 - Outlined
Reserved	bits 8–10	Must be set to 0
`disabled`	bit 7	Enables or disables the menu item: 0 - Item enabled 1 - Item disabled
`divider`	bit 6	Controls drawing divider below item: 0 - No divider bar 1 - Divider bar
`XOR`	bit 5	Controls how highlighting is performed: 0 - Do not use XOR to highlight 1 - Use XOR to highlight item
Reserved	bits 3–4	Must be set to 0
`underline`	bit 2	Controls item underlining: 0 - Do not underline item 1 - Underline item
`italic`	bit 1	Indicates whether item is italicized 0 - Not italicized 1 - Italicized
`bold`	bit 0	Indicates whether item is drawn bold: 0 - Not bold 1 - Bold

`itemTitleRef` Reference to title string for menu item. The `titleRefType` bits in `itemFlag` indicate whether `itemTitleRef` contains a pointer, a handle, or a resource ID. If `itemTitleRef` is a pointer, then the title string must be a Pascal string. Otherwise, the Menu Manager can retrieve the string length from control information in the handle.

`rPString` $8006

Figure E-26 defines the layout of resource type `rPString` ($8006). Resources of this type contain Pascal strings.

■ **Figure E–26** Pascal string, type `rPString` ($8006)



`lengthByte` Number of bytes of data stored in `stringCharacters` array.

`stringCharacters`
Array of `lengthByte` characters.

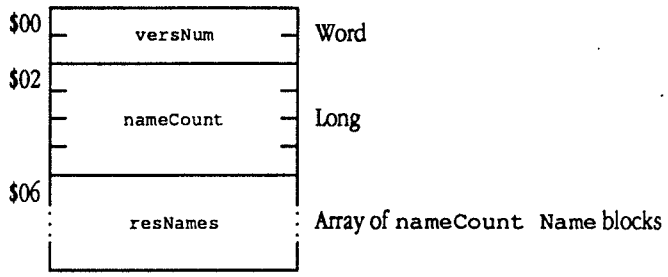
rResName $8014

Figure E-27 defines the layout of resource type `rResName` ($8014). Resources of this type define name strings for resources of a given type and ID. The resource ID value assigned to an `rResName` resource must be formed as follows:

$0001xxxx where xxxx corresponds to the resource type for resources whose names are defined in this resource

Within the `rResName` resource you define name strings corresponding to resources with specified resource IDs. Names are stored in Pascal strings, and must be unique within the appropriate resource type. Resource names are not required, so you may specify names for only a few resources within a given type.

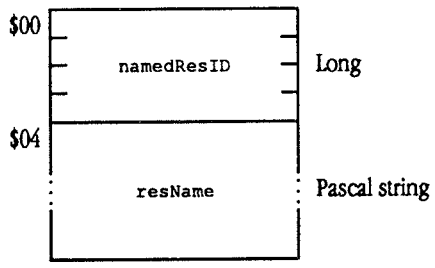
■ **Figure E–27** Resource name array, type `rResName` ($8014)



`versNum` Specifies the resource template version. Must be set to 1.

`nameCount` Count of entries in the `resNames` name definition array.

`resNames` Array of name strings. Each entry must be formatted as follows:



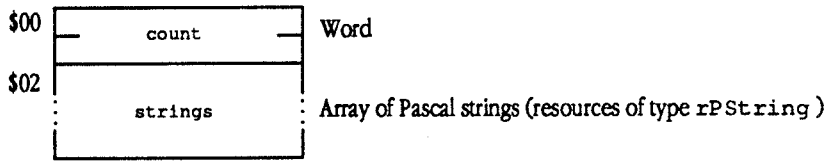
`namedResID` ID of the resource for this name.

`resName` Name string for the resource.

`rStringList` $8007

Figure E-28 defines the layout of resource type `rStringList` ($8007). Resources of this type contain an array of Pascal strings.

■ **Figure E–28** Pascal string array, type `rStringList` ($8007)

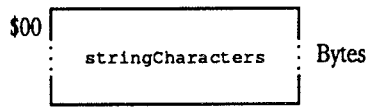


`count`	Indicates the number of Pascal strings stored at `strings`.
`strings`	An array of `count` Pascal strings.

`rText` $8016

Figure E-29 defines the layout of resource type `rText` ($8016). Resources of this type contain text blocks (data arrays with no embedded length information; block length must be indicated in other fields).

■ **Figure E–29** Text block, type `rText` ($8016)

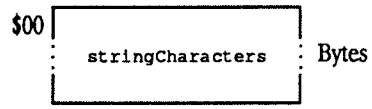


`stringCharacters`
Array of up to 65,535 characters. Any length information is contained in a separately maintained field.

`rTextBlock` $8011

Figure E-30 defines the layout of resource type `rTextBlock` ($8011). Resources of this type contain text blocks (data arrays with no embedded length information; block length must be indicated in other fields).

■ **Figure E–30** Text block, type `rTextBlock` ($8011)

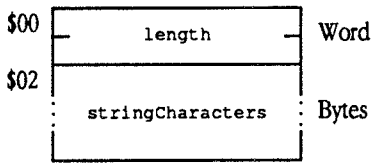


`stringCharacters`
Array of up to 65535 characters. Any length information is contained in a separately maintained field.

`rTextBox2` $800B

Figure E-31 defines the layout of resource type `rTextBox2` ($800B). Resources of this type contain data formatted as input to the `LETextBox2` LineEdit tool call (see Chapter 10, "LineEdit Tool Set," in the *Toolbox Reference* for details).

■ **Figure E–31** `LETextBox2` input text, type `rTextBox2` ($800B)



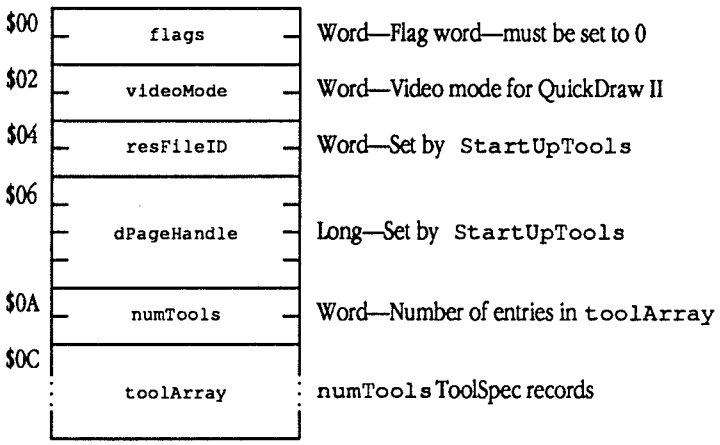
`length` Indicates the number of bytes stored at `stringCharacters`. Valid values lie in the range from 1 to 32767.

`stringCharacters`
Array of up to 32767 characters. Formatting information is embedded in the character array, and is included in the value of `length`. See Chapter 10, "LineEdit Tool Set," in the *Toolbox Reference* for complete information on the syntax for this embedded appearance information.

rToolStartup $8013

Figure E-32 defines the layout of resource type `rToolStartup` ($8013). Resources of this type define tool set start up records for use with the Tool Locator `StartUpTools` and `ShutDownTools` tool calls.(see Chapter 51, "Tool Locator Update," earlier in this book for more information).

■ **Figure E–32** Tool start stop record, type `rToolStartup` ($8013)

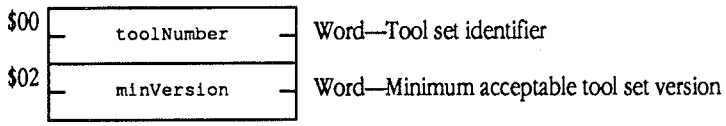


`videoMode` Defines the video mode for QuickDraw II. See Chapter 16, "QuickDraw II," in the *Toolbox Reference* for valid values.

`resFileID` The `StartUpTools` call sets this field. `ShutDownTools` requires it as input.

`dPageHandle` The `StartUpTools` call sets this field. `ShutDownTools` requires it as input.

`toolArray` Each entry defines a tool set to be started. The `numTools` field specifies the number of entries in this array. Each entry is formatted as follows:



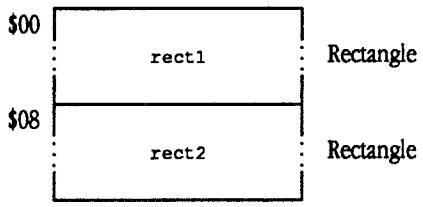
`toolNumber` Specifies the tool set to be loaded Valid tools set numbers are discussed in Chapter 51, "Tool Locator Update," earlier in this book.

`toolVersion` Specifies the minimum acceptable version for the tool set. See Chapter 24, "Tool Locator," in the *Toolbox Reference* for the format of this field.

`rTwoRects` $801A

Figure E-33 defines the layout of resource type `rTwoRects` ($801A).

■ **Figure E–33** Two rectangles, type `rTwoRects` ($801A)

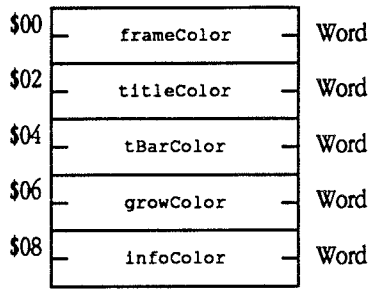


`rect1`	First rectangle.
`rect2`	Second rectangle.

rWindColor $8010

Figure E-34 defines the layout of resource type rWindColor ($8010). Resources of this type define window color tables for the Window Manager.

- **Figure E-34** Window color table, type rWindColor ($8010)



frameColor Color of the window frame and of the alert frame.

Reserved	bits 8–15	Must be set to 0
windowFrame	bits 4–7	Color of window frame—value is an index into the active color table
Reserved	bits 0–3	Must be set to 0

titleColor Colors of inactive title bar, inactive title, and active title:

Reserved	bits 12–15	Must be set to 0
inactiveTitleBar	bits 8–11	Color of inactive title bars—value is an index into the active color table
inactiveTitle	bits 4–7	Color of inactive titles—value is an index into the active color table
activeTitle	bits 0–3	Color of active titles, close box, and zoom box—value is an index into the active color table.

tBarColor Color and pattern information for active title bar:

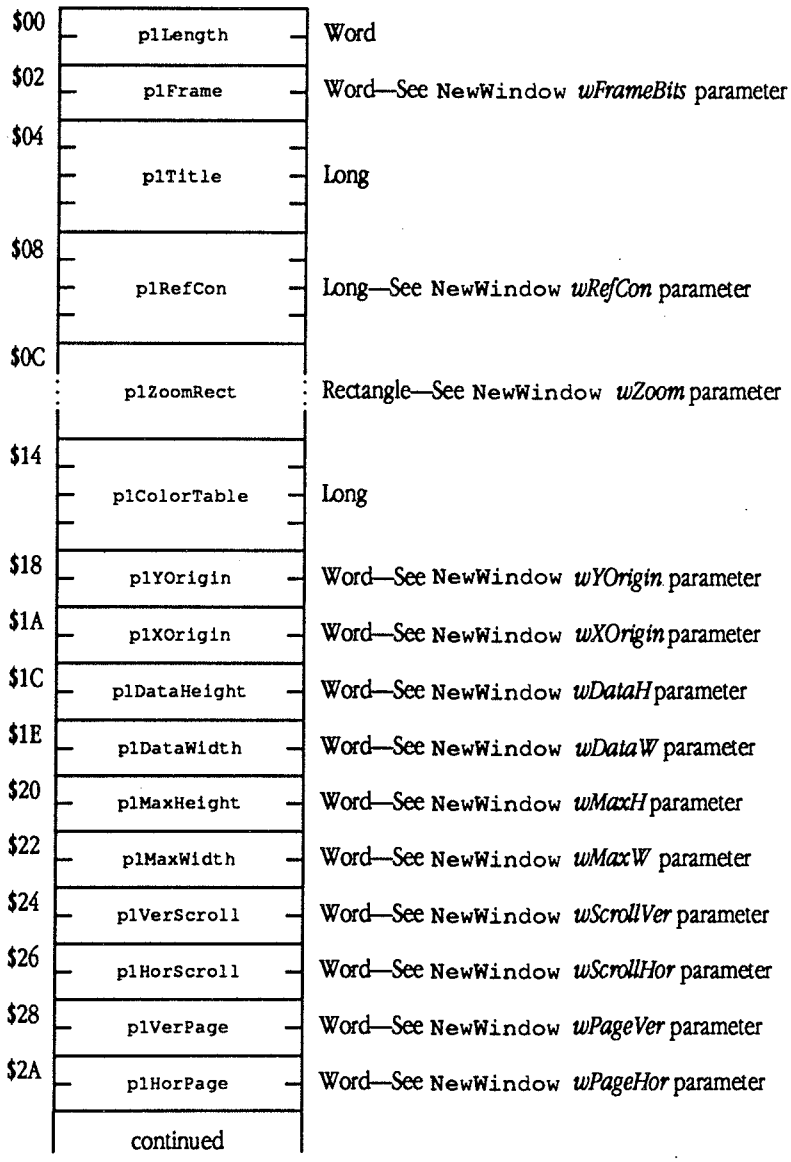
pattern	bits 8–15	Defines pattern for title bar: $00 - Solid $01 - Dither $02 - Lined
patternColor	bits 4–7	Color for pattern—value is an index into the active color table
backColor	bits 0–3	Background color—value is an index into the active color table.

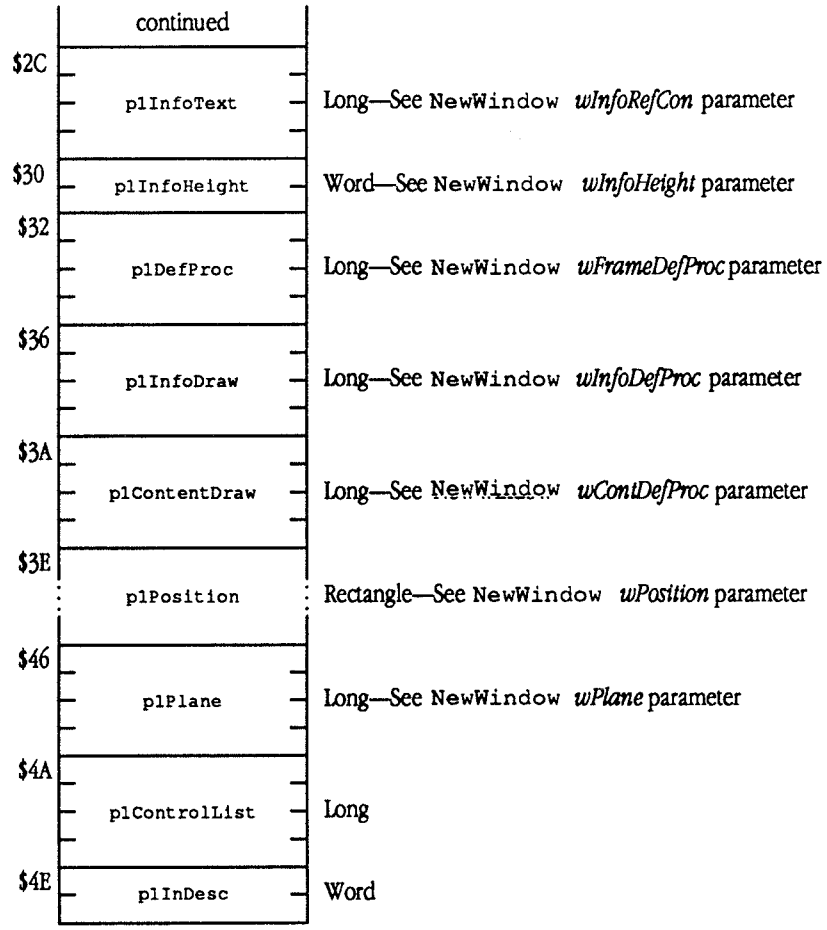
`growColor`	Color of size box and alert frame's middle outline:	
`alertMidFrame`	bits 12–15	Color of alert frame middle outline—value is an index into the active color table
Reserved	bits 8–11	Must be set to 0
`sizeUnselected`	bits 4–7	Color for unselected size box—value is an index into the active color table
`sizeSelected`	bits 0–3	Color for selected size box—value is an index into the active color table.
`infoColor`	Color of information bar and alert frame's inside outline:	
`alertMidFrame`	bits 12–15	Color of alert frame inside outline—value is an index into the active color table
Reserved	bits 8–11	Must be set to 0
`infoBar`	bits 4–7	Color for information bar—value is an index into the active color table
Reserved	bits 0–3	Must be set to 0

rWindParam1 $800E

Figure E-35 defines the layout of resource type `rWindParam1` ($800E). This resource defines a template used to create windows with the `NewWindow2` Window Manager tool call (see Chapter 52, "Window Manager Update," earlier in this book). Most of these fields correspond to fields in the `NewWindow` parameter list (defined in the *Toolbox Reference*).

■ **Figure E-35** Window template, type `rWindParam1` ($800E)





`plLength` Specifies the number of bytes in the template, including the length of `plLength`. Must be set to $50.

`plTitle` Reference to title string for the window. The contents of `plInDesc` specify the type of reference stored here. The title must be stored in a Pascal string, containing both a leading and a trailing space.

If `plTitle` is set to NIL, the Window Manager will create a window without a title bar. If your program is creating a window with a title bar, you must specify a title of some sort. In order to create a window without a title, make `plTitle` (or *titlePtr* on the `NewWindow2` call) refer to a null string.

Note that the Window Manager creates a copy of the title string, so that your program can free the memory for this string after issuing the `NewWindow2` call.

If you specify a non-NIL value for *titlePtr* on the `NewWindow2` call, this field is ignored.

`plColorTable` Reference to the color table for the window. The contents of `plInDesc` specify the type of reference stored here. If `plColorTable` is set to NIL, the Window Manager assumes that there is no color table for the window.

The format of the color table is defined in Chapter 25, "Window Manager," in the *Toolbox Reference*. If `plColorTable` refers to a resource, then the color table must be defined in a resource of type `rWindColor`.

`plControlList` Reference to the template or templates defining controls for the window. The Window Manager passes this value to the `NewControl2` Control Manager tool call as the *reference* parameter. Note that `plInDesc` contains the data for the `NewControl2` *referenceDesc* parameter. Refer to Chapter 28, "Control Manager Update," in this book for more information about `NewControl2`.

If this field is set to NIL, then the Window Manager assumes that there is no control list for the window and does not call `NewControl2`.

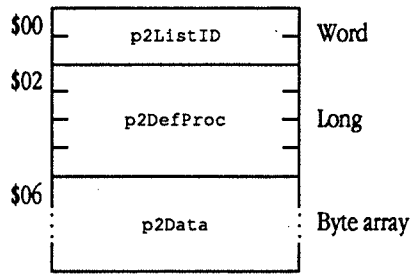
`plInDesc` Defines the type of reference stored in `plColorTable` and `plTitle`. Also contains the *referenceDesc* value for `NewControl2` that defines the contents of `plControlList`:

Reserved	bits 12–15	Must be set to 0
`colorTableRef`	bits 10–11	Define the type of reference stored in `plColorTable`: 00 - Reference is pointer to color table 01 - Reference is handle to color table 10 - Reference is resource ID of `rWindColor` resource 11 - Invalid value
`titleRef`	bits 8–9	Define the type of reference stored in `plTitle`: 00 - Reference is pointer to Pascal string 01 - Reference is handle to Pascal string 10 - Reference is resource ID of `rPString` resource 11 - Invalid value
`controlRef`	bits 0–7	Define the type of reference stored in `plControlList`. Passed directly to the `NewControl2` Control Manager tool call as the *referenceDesc* parameter. For valid values, see the description of the `NewControl2` tool call in Chapter 28, "Control Manager Update," earlier in this book.

rWindParam2 $800F

Figure E-36 defines the layout of resource type `rWindParam2` ($800F). This resource defines a template used to create windows with the `NewWindow2` Window Manager tool call (see Chapter 52, "Window Manager Update," earlier in this book). Use this template for custom windows.

■ **Figure E-36** Window template, type `rWindParam2` ($800F)



`p2ListID` Specifies the resource template version. Must be set to NIL.

`p2DefProc` Pointer to the definition procedure for the window. When using the `rWindParam2` window template, you must pass a pointer to a valid definition procedure, either in the template or with the *defProcPtr* parameter to the `NewWindow2` Window Manager tool call. On disk, this field does not contain a valid value.

`p2Data` Window definition data required by the routine pointed to by `p2DefProc`. The format and content of this field is determined by the window definition procedure.

Appendix F **Delta Guide**

This appendix collects all information that corrects errors or clarifies ambiguities in Volumes 1 and 2 of the *Toolbox Reference*. This information was derived from the "Error corrections" and "Clarifications" sections of each chapter in this book. This appendix contains a separate major section for tool set to be addressed; the sections are presented alphabetically, by tool set name.

Apple Desktop Bus

The following sections correct errors or omissions in Chapter 3, "Apple Desktop Bus Tool Set," in Volume 1 of the *Toolbox Reference.*

Error correction

The parameter table for the `ReadKeyMicroData` tool call ($0A09) in Volume 1 of the *Toolbox Reference* incorrectly describes the format for the `readConfig` command ($0B). The description should be as follows.

Command	*dataLength*	Name	Action
$0B	3	`readConfig`	Read configuration; *dataPtr* refers to a 3-byte data structure: Byte — ADB keyboard and mouse addresses low nibble - keyboard high nibble - mouse Byte — Keyboard layout and display language low nibble - keyboard layout high nibble - display language Byte — Repeat rate and delay low nibble - repeat rate high nibble - repeat delay

The description of this configuration record is also wrong in the tool set summary. The following table shows the correct information.

Name	Offset	Type	Definition
`ReadConfigRec` (configuration record for `ReadKeyMicroData`)			
`rcADBAddr`	$0000	Byte	ADB keyboard and mouse addresses low nibble - keyboard high nibble - mouse
`rcLayoutOrLang`		$0001	Byte Keyboard layout and display language low nibble - keyboard layout high nibble - display language
`rcRepeatDelay`	$0002	Byte	Repeat rate and delay low nibble - repeat rate high nibble - repeat delay

Clarification

This section presents new information about the `AsyncADBReceive` call.

You can call `AsyncADBReceive` to poll a device using register 2, and it will return certain useful information about the status of the keyboard. The call returns the following information in the specified bits of register 2:

Bit 5: 0-Caps Lock key down
1-Caps Lock key up

Bit 3: 0-Control key down
1-Control key up

Bit 2: 0-Shift key down
1-Shift key up

Bit 1: 0-Option key down
1-Option key up

Bit 0: 0-Command key down
1-Command key up

Control Manager

The following sections correct errors or omissions in Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference.*

Error corrections

This section documents errors in Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference.*

- The color table for the size box control in the *Toolbox Reference* is incorrect. The correct table follows, with new information in boldface.

`growOutline` word		Color of size box's outline
	Bits 8–15	= zero
	Bits 4–7	= outline color
	Bits 0–3	= zero
`growNorBack` word		Color of interior when not highlighted
	Bits 8–15	= zero
	Bits 4–7	= background color
	Bits 0–3	= icon color
`growSelBack` word		**Color of interior when highlighted**
	Bits 8–15	**= zero**
	Bits 4–7	**= background color**
	Bits 0–3	**= icon color**

- On page 4-76 of the *Toolbox Reference,* in the section that covers the `SetCtlParams` call, it states that the call "Sets new parameters to the control's definition procedure . . ." This description is misleading; the call does not directly set the parameters. Rather, it *sends* the new parameters to the control's definition procedure, unlike `SetCtlValue`, which actually sets the appropriate value in the control record and then passes the value on to the definition procedure.

Clarifications

The following items provide additional information about features previously described in the *Toolbox Reference.*

- The `barArrowBack` entry in the scroll bar table was never implemented as first intended, and is now no longer used.
- The Control Manager preserves the current port across Control Manager calls, including those that are passed through other tools, such as the Dialog Manager.
- The Control Manager preserves the following fields in the port of a window that contains controls:

`bkPat`	background pattern
`pnLoc`	pen location
`pnSize`	pen size
`pnMode`	pen mode
`pnPat`	pen pattern
`pnMask`	pen mask
`pnVis`	pen visibility
`fontHandle`	handle of current font
`fontID`	ID of current font
`fontFlags`	font flags
`txSize`	text size
`txFace`	text face
`txMode`	text mode
`spExtra`	value of space extra
`chExtra`	value of character extra
`fgColor`	foreground color
`bgColor`	background color

- The control definition procedures for simple buttons, check boxes, and radio buttons can now compute the size of their boundary rectangles automatically. The computed size is based on the size of the title string for the button.
- To ensure predictable color behavior, you should always align color table–based controls on an even pixel boundary in 640 mode. If you do not do so, the control will not appear in the colors you specify, due to the effect of dithering.

Dialog Manager

The following sections correct errors or omissions in Chapter 6, "Dialog Manager," in Volume 1 of the *Toolbox Reference.*

Error corrections

This section explains changes that have been made to the Dialog Manager's documentation in the *Apple IIGS Toolbox Reference.*

- The documentation for `SetDItemType` on page 6-82 of the *Toolbox Reference* says that the call is used to change a dialog item to a different type. In fact, `SetDItemType` should be used only to change the *state* of an item from enabled to disabled or vice versa.
- The Dialog Manager does not support dialog item type values of `picItem` or `iconItem`, contrary to what the *Toolbox Reference* states in Table 6-3 on page 6-12.

Integer Math Tool Set

The following section describes a bug that has been fixed in the Integer Math Tool Set.

Clarifications

- The `Long2Dec` Integer Math tool call now correctly handles input long values that have the low-order three bytes set to zero. Previously, if the input long had its low-order three bytes set to zero, `Long2Dec` would always return a zero value, even if the high-order byte was non-zero.

List Manager

The following sections correct errors or omissions in Chapter 11, "List Manager," in Volume 1 of the *Toolbox Reference.*

Clarifications

- The *Apple IIGS Toolbox Reference* states that a disabled item of a list cannot be selected. In fact, a disabled item can be selected, but it cannot be highlighted. The List Manager provides the ability to select disabled (dimmed) items so that it is possible, for instance, for a user to select a disabled menu choice as part of a help dialog. To make an item unselectable, set it inactive (see "List Manager definitions" later in this chapter).
- Any List Manager tool call that draws will change fields in the GrafPort. If you are using List Manager tool calls you must set up the GrafPort correctly and save any valuable GrafPort data before issuing the call.
- Member text is now drawn in 16 colors in both 320 and 640 mode.
- Previous versions of List Manager documentation do not clearly define the relationship between the `listView`, `listMemHeight`, and `listRect` fields in the list record. To clarify this point, note that the following formula must be true for values in any list record:

```
listView * listMemHeight + 2 = listRect.v2 - listRect.v1
```

If you set `listView` to 0, the List Manager will automatically adjust the `listRect.v2` field and set the `listView` field so that this formula holds. Note that if you pass a 0 value for `listView` the bottom boundary of `listRect` may change slightly.

List Manager definitions

The following terms define the valid states for a list item.

inactive	Bit 5 of the list item's `memFlag` field is set to 1. Inactive items appear dimmed and cannot be highlighted or selected.
disabled	Bit 6 of the list item's `memFlag` field is set to 1. Disabled items appear dimmed and cannot be highlighted.

enabled	Bit 6 of the list item's `memFlag` field is set to 0. Enabled items appear normal and can be highlighted.
selected	Bit 7 of the list item's `memFlag` field is set to 1. This bit is set when a user clicks on the list item, or the item is within a range of selected items. A selected item appears highlighted only if it is also enabled.
highlighted	A member of a list appears highlighted only when it is both selected and enabled. This means that bit 7 of the `memFlag` field is set to 1 and bit 6 is set to 0. A highlighted member is drawn in the highlight colors.

Memory Manager

The following sections correct errors or omissions in Chapter 12, "Memory Manager," in Volume 1 of the *Toolbox Reference.*

Error correction

On page 12-10 of the *Toolbox Reference*, Figure 12-7 shows the low-order bit of the User ID is reserved. This is not correct. The figure should show that the `mainID` field comprises bits 0–7, and that the `mainID` value of $00 is reserved.

Clarification

The *Toolbox Reference* documentation of the `SetHandleSize` call ($1902) states "If you need more room to lengthen a block, you may compact memory or purge blocks." This is misleading. In fact, to satisfy a request the Memory Manager will compact memory or purge blocks in order to free sufficient contiguous memory. Therefore, the sentence should read "If your request requires more memory than is available, the Memory Manager may compact memory or purge blocks, as needed."

Menu Manager

The following sections correct errors or omissions in Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference.*

Error corrections

- In the description of the `SetSysBar` tool call (pages 13-86 and 13-3), the *Toolbox Reference* states that, after an application issues this call, the new system menu bar becomes the current menu bar. This is incorrect. Your application must issue the `SetMenuBar` tool call to make the new menu bar the current menu bar.
- In the definition of the menu bar record (pages 13-17–18), the *Toolbox Reference* shows that bits 0–5 of the `ctlFlag` field are used to indicate the starting position for the first title in the menu bar. This is incorrect. The `ctlHilite` field defines the starting position for the first title. Note further that the entire `ctlHilite` field is used in this manner. The documented purpose of the `ctlHilite` field (number of highlighted titles) is not supported by the Menu Bar record.

Clarifications

- The `SetBarColors` tool call changes the color table for all menu bars in a window. If you want to use separate color tables for different menu bars, your application must build a menu bar color table and modify the `ctlColor` field of the appropriate control record to point to this custom color table. See "SetBarColor" in Chapter 13, "Menu Manager," of the *Toolbox Reference* for the format and contents of a menu bar color table.
- The description of the `InsertMenu` tool call should also note that your application must call `FixMenuBar` before calling `DrawMenuBar` in order to display the modified menu bar.
- The description of the `InitPalette` tool call in the *Toolbox Reference* should also note that the call changes color tables 1 through 6 to correspond to the colors needed for drawing the Apple logo in its standard colors.
- The `CalcMenuSize` call uses the *newWidth* and *newHeight* parameters to compute a menu's size. These parameters may contain the width and height of the menu, or may contain the values $0000 or $FFFF. A value of $0000 tells `CalcMenuSize` to calculate the parameter automatically. A value of $FFFF tells it to calculate the parameter only if the current setting is 0.

The effect of all three uses:

- **Pass the new value.** The value passed will become the menu's size. Use this method when a specific menu size is needed.
- **Pass $0000.** The size value will be automatically computed. This option is useful if menu items are added or deleted, rendering the menu's size incorrect. The menu's height and width can be automatically adjusted by calling `CalcMenuSize` with *newWidth* and *newHeight* equal to $0000.
- **Pass $FFFF.** The width and height of a menu is 0 when it is created. `FixMenuBar` calls `CalcMenuSize` with *newWidth* and *newHeight* equal to $FFFF to calculate the sizes of those menus with heights and widths of 0.

Miscellaneous Tool Set

The following sections correct errors or omissions in Chapter 14, "Miscellaneous Tool Set," in Volume 1 of the *Toolbox Reference*.

Error corrections

- On page 14-58 of the *Toolbox Reference*, Figure 14-3 shows the low-order bit of the User ID is reserved. This is not correct. The figure should show that the `mainID` field comprises bits 0–7, and that the `mainID` value of $00 is reserved.
- The sample code on page 14-28 contains an error. In the code to clear the 1 second IRQ source, the second instruction reads

  ```
  TSB    $C032
  ```

  This instruction should read

  ```
  TRB    $C032
  ```

- The descriptions of the `PackBytes` and `UnPackBytes` tool calls in unclear with respect to the *startHandle* parameter to each call. The stack diagrams correctly describe the parameter as a pointer to a pointer. However, the C sample code for each call defines *startHandle* as a handle. In both cases, *startHandle* is not a Memory Manager handle, but is a pointer to a pointer. Creating *startHandle* as a handle will cause unpredictable system behavior.

Print Manager

The following sections correct errors or omissions in Chapter 15, "Print Manager," in Volume 1 of the *Toolbox Reference.*

Error corrections

This section documents errors in the *Toolbox Reference.*

- The diagram for the job subrecord, figure 15-10 on page 15-14 of the *Toolbox Reference,* shows that the `fFromUsr` field is a word. This is incorrect. The `fFromUsr` field is actually a byte. Note that the offsets for all fields following this one are incorrect, as a result. This error is also reflected in the tool set summary at the end of the chapter.
- The description of the `PrJobDialog` tool call states that "The initial settings displayed in the dialog box are taken from the printer driver . . .." This is incorrect. The sentence should begin "The initial settings displayed in the dialog box are taken from the print record . . .."

Clarifications

- The existing *Toolbox Reference* documentation for the `PrPicFile` tool call does not mention that your program may pass a NIL value for *statusRecPtr.* Passing a NIL pointer causes the system to allocate and manage the status record internally.
- The `PrPixelMap` call (documented in Volume 1 of the *Toolbox Reference*) provides an easy way to print a bitmap. It does much of the required processing, and an application need not make the calls normally required to start and end the print loop. The *srcLocPtr* parameter must be a pointer to a *locInfo* record (see Figure 16-3 in Chapter 16, "QuickDraw II," in the *Toolbox Reference* for the layout of the *locInfo* record).

QuickDraw II

The following sections correct errors or omissions in Chapter 16, "QuickDraw II," in Volume 2 of the *Toolbox Reference.*

Error corrections

The following items provide corrections to the documentation for QuickDraw II in the *Apple IIGS Toolbox Reference:*

- The documentation in the *Toolbox Reference* that explains pen modes is somewhat misleading. There are, in fact, 8 drawing modes, and you may set the pen to draw lines and other elements of graphics in any of these modes. There are also 16 modes used for drawing text, and they are completely independent of the graphic pen modes. The 8 drawing modes listed in Table 16-9 on page 16-235 are valid modes for either the text pen or the graphics pen. You can set either pen to any of these modes by using the appropriate calls. You can also set the text pen to 8 other modes. These modes are listed in the table on page 16-260 of the *Toolbox Reference.* The `SetPenMode` call sets the mode used by the graphics pen; the `SetTextMode` call sets the mode used by the text pen. Setting either one does not affect the other.
- There are two versions of the Apple IIGS standard 640-mode color tables, one on page 16-36 and one on page 16-159. The two tables are different; Table 16-7 on page 16-159 is correct.
- In the QuickDraw II chapter, the *Apple IIGS Toolbox Reference* states that the coordinates passed to the `LineTo` and `MoveTo` calls should be expressed as global coordinates. In fact, the coordinates must be local coordinates, and must refer to the GrafPort in which the drawing or moving takes place.

Sound Tool Set

The following sections correct errors or omissions in Chapter 20, "Sound Tool Set," in Volume 2 of the *Toolbox Reference.*

Error corrections

This section provides corrections to the documentation of the Sound Tool Set in the *Apple IIGS Toolbox Reference.*

- The documentation of the `FFSoundDoneStatus` call includes an error. You will note that the paragraph that describes the call does not agree with the "Stack after call" diagram. The text states that the call returns TRUE if the specified sound is still playing, whereas the diagram states that it returns FALSE if still playing. The diagram, not the text, is correct.
- There is an undocumented distinction between a generator that is playing a sound and one that is active. A generator that is playing a sound returns FALSE in response to an `FFSoundDoneStatus` call. One that is active may or may not be playing a sound; the value of the flag returned by `FFSoundStatus` is TRUE. Active generators are those that are allocated to a voice. At any given moment the generator may be playing a sound, and so the `FFSoundDoneStatus` returns FALSE—or it may be silent between notes, in which case `FFSoundDoneStatus` returns TRUE.

Clarification

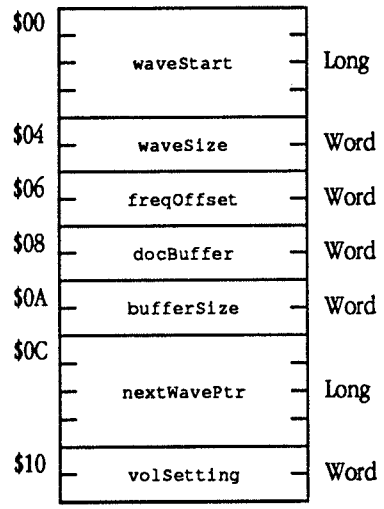
This section presents more complete information about the `FFStartSound` tool call, including further explanation of its parameters, a new error code, an example procedure for moving a sound from the Macintosh to the Apple IIGS and some sample code demonstrating the use of the call. The original documentation for this call is in Chapter 20, "Sound Tool Set," in Volume 2 of the *Toolbox Reference.*

FFStartSound

The freeform synthsizer is designed to play back long wave forms. In order to handle longer waveforms the synthesizer uses two buffers (which must be the same size), alternating its input from one to the other. When the synthesizer exhausts a buffer, it generates an interrupt and then starts reading data from the other buffer. The Sound Tool Set services the interrupt and begins refilling the empty buffer. This process continues until the waveform has been completely played.

Note that all synthesizer input buffers must be buffer-size aligned. That is, if you have allocated 4K buffers, then those buffers must be aligned on 4K memory boundaries.

Parameter Block



`waveStart` The starting address of the wave to be played, not in DOC RAM but in Apple IIGS system RAM. The Sound Tool Set loads the waveform data into DOC RAM as it is played.

`waveSize` The size in pages of the wave to be played. A value of 1 indicates that the wave is one page (256 bytes) in size; a value of 2 indicates that it is two pages (512 bytes) in size—and so on as you might expect. The only anomaly is that a value of 0 specifies that the wave is 65,536 pages in size.

`freqOffset`	This parameter is copied directly into the Frequency High and Frequency Low registers of the DOC. See the previous discussion of those registers for more complete information.
`docBuffer`	Contains the address in Sound RAM where buffers are to be allocated. This value is written to the DOC WaveForm Table Pointer register. The low-order byte is not used, and should always be set to 0.
`bufferSize`	The lowest three bits set the values for the table-size and resolution portions of the DOC Bank-Select/Table-size/Resolution register. See the previous discussion of that register for details.
`nextWavePtr`	This is the address of the next waveform to be played. If the field's value is 0, then the current waveform is the last waveform to be played.
`volSetting`	The low byte of the `volSetting` field is copied directly into the Volume register of the DOC. All possible byte values are valid.

New error code

$0817	`IRQNotAssignedErr`	No master IRQ was assigned.

Moving a sound from the Macintosh to the Apple IIGS

To move a digitized sound from the Macintosh to the Apple IIGS and play the sound, you will have to perform the following steps

1. Save the sound as a pure data file on the Macintosh.
2. Transfer the file to the Apple IIGS (using Apple File Exchange, for example).
3. Filter all the zero sample bytes out of the file by replacing them with bytes set to $01. This is very important, because the Apple IIGS interprets zero bytes as the end of a sample.
4. Load the sound into memory with GS/OS calls.
5. Play the sound with the `FFStartSound` tool call.

   Set the `freqOffset` parameter to $01B7 in order to play the sound at the same tempo as the Macintosh.

Sample code

This assembly-language code sample demonstrates the use of the `FFStartSound` tool call.

```
            PushWord    chanGenType     ; set generator for FFSynth
            PushLong    #STParamBlk     ; address of parm block
            _FFStartSound               ; start free-from synth

ChanGenType DC.W $0201                  ; generator 2, FFSynth

STParamBlk  DS.L 1                      ; store the address of the
                                        ;  sound in system memory here
            Entry       WaveSize
WaveSize    DS.W 1                      ; store the number of pages to
                                        ;  play here
Freq        DC.W $200                   ; A9 set for each sample once
Start       DC.W $8000                  ; start at beginning
Size        DC.W $6                     ; 16k buffers
Nxtwave     DC.L $0                     ; no new param block
Vol         DC.W $FF                    ; maximum volume
```

Window Manager

The following sections correct errors or omissions in Chapter 25, "Window Manager," in Volume 2 of the *Toolbox Reference.*

Error corrections

This section corrects some errors in the Window Manager documentation in the *Apple IIGS Toolbox Reference.*

- The manual's description of `SetZoomRect` is incorrect. The correct description is as follows:

  Sets the `fZoomed` bit of the window's `wFrame` record to 0. The rectangle passed to `SetZoomRect` then becomes the window's zoom rectangle. The window's size and position when `SetZoomRect` is called becomes the window's unzoomed size and position, regardless of what the unzoomed characteristics were before `SetZoomRect` was called.

- *Apple IIGS Toolbox Reference* page 25-126, third line:

  If `wmTaskMask` bit `tmInfo` (bit 15) = 1

  should read:

  If `wmTaskMask` bit `tmInfo` (bit 15) = 0

- When used with a window that does not have scroll bars, the call `WindNewRes` calls the window's defProc to recompute window regions. A call to `SizeWindow` is not necessary under these circumstances.

Glossary

ACIA: See **Asynchronous Communications Interface Adapter**.

Adaptive Differential Pulse Code Modulation (ADPCM): An algorithm for digitizing audio samples. Used in the Apple IIGS Audio Compression and Expansion Tool Set for compressing audio samples.

ADPCM: See **Adaptive Differential Pulse Code Modulation**.

ADSR: Acronym for **attack**, **decay**, **sustain**, **release**. These terms describe the paradigm for representing sounds in terms of a sound **envelope**.

alert window: Similar to a modal dialog box; used to present urgent or important information to the user. You create alert windows with the `AlertWindow` Window Manager tool call.

Asynchronous Communications Interface Adapter (ACIA): Adapter card that allows the Apple IIGS to support asynchronous communications protocols with external devices.

attack: That portion of a sound **envelope** where the sound is increasing from silence to its peak loudness. See also **ADSR**.

control template: Structure containing the information necessary for the `NewControl2` Control Manager tool call to create a new control.

decay: That portion of a sound **envelope** where the sound falls off from its peak loudness to a sustained level. See also **ADSR**.

Digital Oscillator Chip (DOC): Integrated circuit that supports the sound capabilities of the Apple IIGS.

DOC: See **Digital Oscillator Chip**.

drop sample tuning: Describes a technique for changing the pitch of a played sound that relies on skipping (or dropping) sound samples on playback. By dropping samples at a fixed rate, the pitch of a sound can be raised in octave increments.

envelope: A graphical representation of a sound's loudness over time. The envelope typically consists of segments identified as **attack**, **decay**, **sustain** and **release**, or **ADSR**.

extended controls: Controls created with the `NewControl2` Control Manager tool call, rather than the `NewControl` call. Extended controls have new-style control records that contain more information than those created by `NewControl`.

keystroke equivalent: A keystroke that activates a control just as if the user had clicked in the control.

menu template: Data structure used to define menus, menu items, and menu bars to the Menu Manager.

MIDI: See **Musical Instrument Digital Interface**.

Musical Instrument Digital Interface (MIDI): An interface specification that allows external devices to control electronic musical instruments.

out-of-memory queue: A queue maintained by the Memory Manager. Queue elements (**out-of-memory routines**) refer to code to be executed when the Memory Manager detects an out-of-memory condition.

out-of-memory routines: Code executed by the Memory Manager when it detects an out-of-memory condition. The **out-of-memory queue** consists of a list of these routines.

password field: Do not echo user input, allowing protected data entry. Your program can specify the echo character; the default echo character is asterisk (*).

pop-up menu: A menu that "pops" out of its display rectangle when selected by the user. The

two types of pop-up menus, **type 1** and **type 2** pop-up menus, have different limits on their maximum size.

reference type: Indicates whether a storage location contains a pointer, a handle, or a resource ID for an object.

release: That portion of a sound **envelope** where the note dies away to silence. See also **ADSR**.

resource: Collection of data managed by the Resource Manager for other applications.

resource file: A collection of one or more **resources**. The Resource Manager provides routines for accessing and updating resources in a resource file.

resource ID: Uniquely identifies a **resource** within the context of its **resource type**. The Resource Manager provides facilities to assign unique resource IDs. Compare **resource name**.

resource name: Uniquely identifies a **resource** within the context of its **resource type**. Note that resource names are not maintained by the system; it is your program's responsibility to assign and manage them. Compare **resource ID**.

resource type: Identifies a class of **resources** that share a common data layout. Individual instances of **resources** of a given **type** are identified by their unique **resource ID** or **resource name**.

run item: An element in the **run queue**. Run items specify program code to be executed by the Desk Manager at regular intervals.

run queue: A queue maintained by the Desk Manager that contains elements (**run items**) that specify code to be executed at regular intervals.

sample rate: Specifies the number of sound samples the Apple IIGS **DOC** plays per second.

sustain: That portion of a sound **envelope** where the note maintains a fairly constant loudness, before it dies away. See also **ADSR**.

target control: That control which is currently the recepient of user actions (keystrokes and menu items).

TextEdit record: Describes a TextEdit user session, whether or not that session is managed as a control.

type 1 pop-up menu: A **pop-up menu** that will not grow beyond its window constraints. Compare **type 2 pop-up menu**.

type 2 pop-up menu: A **pop-up menu** that will grow beyond its window if necessary to display its menu items. Compare **type 1 pop-up menu**.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh® computers and Microsoft® Word software. Proof and final pages were created on the Apple LaserWriter® printers. Line art was created using Adobe Illustrator™. POSTSCRIPT®, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type and display type are Apple's corporate font, a condensed version of Garamond. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.