Apple IIGS Toolbox Changes for System Software 6.0

(ERS version 2.5)

by David A. Lyons

February 26, 1992

TABLE OF CONTENTS

ABOUT THIS DOCUMENT

This document describes enhancements to the Apple IIGS toolbox for System Software 6.0.

REVISION HISTORY

17..26-Feb-92	Cleaned up for final release. Incorporated changes from 6.0d53 through 6.0d75.

Taking Advantage of New System 6.0 Toolbox Features

Some System 6.0 toolbox features are "free"—they automatically work without requiring any change to your application. `AlertWindow` is one example. You automatically gain keyboard control with System 6.0.

Smooth application launching is another free feature. If your desktop application's auxiliary type bits are set as specified in Apple II File Type Note $B3 and you use `StartUpTools` and `ShutDownTools`, the user sees a smooth transition from the Finder (for example) to your application and back.

Other features are not automatic, but you can take advantage of them by making small changes to your application.

For example, adding a call to `HandleDiskInsert` in your main event loop makes your application automatically recognize disks as they are inserted, giving the user a chance to format them if necessary (see `HandleDiskInsert` in the Window Manager chapter).

Also, if you have any extended List controls, providing keyboard navigation through the lists is often as simple as turning on some bits in the controls' `moreFlags` fields (see Control Manager and List Manager).

New Resource Types

New resource types defined for System Software 6.0 and HyperCard IIGS 1.1 are listed below. For information on other resource types defined by Apple, see:

- Apple IIGS Toolbox Reference, Volume 3
- Apple IIGS Technical Note #76, Miscellaneous Resource Formats
- HyperCard IIGS Script Language Guide

$801B `rFileType`

For your convenience, `rFileType` resources are defined as having the same format as Filetype Descriptor files (see Apple II File Type Note $42). There is no direct support in the system for resources of this type.

$8028 `rItemStruct`

Supports menu items with icons attached. See the Menu Manager chapter.

$8029 `rVersion`

Any file with an Apple IIGS-format resource fork can have an `rVersion` resource with ID=1. The Finder displays this information in Icon Info windows.

`rVersion` format:
`+000:` Version Number (see format below)
`+004:` Country Code (0=USA)
`+006:` Pascal String giving name of product (may be empty)
`+xxx:` Pascal String giving additional info, like a copyright notice (may be empty)

Version Number Format

The version number format is as described in Macintosh Technical Note #189, Version Territory, except that the bytes are in least-significant-first order and we have a “final” stage before the actual “release” stage. (Note: It is incorrect to have a nonzero release field when the stage is “released.”)

An important property of Version longwords is that you can do an unsigned long comparison to see which version is later.

```
Version = 32 bits:

                 ___________________________________________________________
bit 31  |M|M|M|M|M|M|M|M|m|m|m|m|b|b|b|b|s|s|s|0|0|0|0|0|r|r|r|r|r|r|r|r|  bit
0
        ------------------------------------------------------------------
                        |               |               |

          M = major version (2 digits, BCD)
          m = minor version
          b = bug version
          s = stage (001=development, 010=alpha, 011=beta, 100=final,
101=released)
          r = release (2 digits, BCD)
```

Examples:

```
$06002021 = 6.0d21
$1234A000 = 12.3.4
$05042002 = 5.0.4d2
$01234001 = 1.2.3a1
$01236099 = 1.2.3b99
$01008001 = 1.0f1
```

$802A `rComment`

An `rComment` resource consists of unformatted text. Any file with an Apple IIGS-format resource fork can have an `rComment` resource with ID=1 containing information about the file. The Finder displays this text and lets the user edit it.

The Finder displays a file's `rComment`(2) resource if it can't be launched. This text can explain, for example, what the file is for (for files that aren't intended to be launched), or what application created it.

All other resource IDs within `rComment` are reserved for future definition by Apple.

$802B `rBundle`

This is defined by Finder 6.0.

$802C `rFinderPath`

This is defined by Finder 6.0.

$802D `rPaletteWindow`

Used by HyperCard IIGS 1.1.

$802E rTaggedStrings

An rTaggedStrings resource contains a number of <Word, Pascal String> pairs.

```
/*------------------------- rTaggedStrings ---------------------------------*/

type rTaggedStrings {
             integer = $$Countof(StringArray);
             array StringArray {
                          hex integer;        /* Key integer */
                          pstring;            /* String */
             };
};
```

$802F rPatternList

An rPatternList resource contains zero or more QuickDraw II patterns. (This type is defined for your convenience; the toolbox does not use it directly.)

```
/*------------------------- rPatternList -----------------------------------*/

type rPatternList {
      array {
             array[32] {
                    hex byte;
             };
      };
};
```

Additions to Sys.Resources

Added new rCtlDefProc resources—see the Control Manager chapter.

Added some rIcon (type = $8001) resources (for convenience, some of them have resource names):

$07FF0058	640-mode “X” icon to overlay ShowBootInfo icons
$07FF0002	640-mode Stop icon (name = “Stop”)
$07FF0003	640-mode Note icon (name = “Note”)
$07FF0004	640-mode Caution icon (name = “Caution”)
$07FF0005	640-mode Disk icon (name = “Disk”)
$07FF0006	640-mode Disk Swap icon (name = “Disk Swap”)
$07FF0102	320-mode Stop icon
$07FF0103	320-mode Note icon
$07FF0104	320-mode Caution icon
$07FF0105	320-mode Disk icon
$07FF0106	320-mode Disk Swap icon

Added some rCursor (type = $8027) resources:

`$07FF0001` 640-mode I-Beam cursor
`$07FF0002` 640-mode Cross cursor
`$07FF0003` 640-mode Plus cursor
`$07FF0101` 320-mode I-Beam cursor
`$07FF0102` 320-mode Cross cursor
`$07FF0103` 320-mode Plus cursor

Added new `rErrorString` (type = $8020) resources for `ErrorWindow`:

`$07FF006A` = "Generic FST error ($6A)."
`$07FF0042` = "Cannot open file. Too many files are open on the server." (Stop icon)
`$07FF0096` = "GS/OS can't read this disk (in device *0). Do you want to initialize it?" **Eject**/Initialize (Caution icon)
`$07FF0097` = "GS/OS does not recognize the file system on this disk (in device *0). Do you want to initialize it?" **Eject**/Initialize (Caution icon)
`$07FF0098` = "Font size must be a number from 1 to 255." **Continue** (Stop icon)
`$07FF0099` = "The disk could not be formatted. (blank line) 800K disks can't be formatted as 1440K, and 1440K disks can't be formatted as 800K." **Continue** (Stop icon)

Most `rErrorString` resources now use Stop or Caution icons, and they use a Continue button rather than an OK button. From the user's point of view, things are definitely not OK when one of these errors occurs!

Added one `rWindColor` (type = $8010) resource:

`$07FF0001` black-and-white lined-pattern title bar

Added an `rVersion` (type = $8029) ID=1 resource for the System Software version. Product name = "System", string2 = "Copyright 1983-1992, Apple Computer, Inc."

009 Apple Desktop Bus Update

For ROM 1, `ADBVersion` now returns version 3.0 for consistency with ROM 3. There are no other changes.

029 Audio Compression and Expansion Update

New Audio Compression and Expansion Calls

Added two new calls for dealing with pieces of sounds. These calls are useful when working with AIFF-C files.

`GetACEExpState` $0D1D

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|    theBuffer    |   Long—Pointer to buffer
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                   <—SP
```

Errors `$1D03` `aceNotActive`

```
C           extern pascal void GetACEExpState(theBuffer);
            Ptr theBuffer;
```

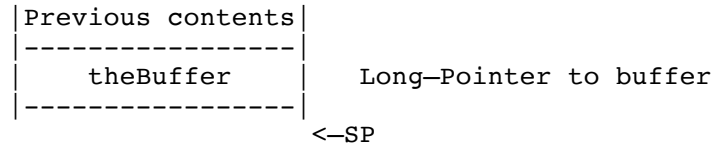
The buffer is 16 bytes long. Currently only the first 4 bytes are used, and the last 12 are returned as zero.

By setting the Expansion State appropriately, you can pick up an expansion part-way through some compressed sound data.

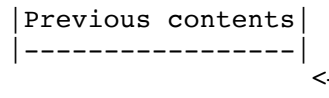
SetACEExpState $0E1D

Parameters

Stack before call



Stack after call



Errors $1D03 aceNotActive

```
C           extern pascal void SetACEExpState(theBuffer);
            Ptr theBuffer;
```

The buffer is 16 bytes long. Currently only the first 4 bytes are used, and the last 12 are returned as zero.

By setting the Expansion State appropriately, you can pick up an expansion part-way through some compressed sound data.

016 Control Manager Update

New Features of the Control Manager

New Control Types

Thermometer controls and Rectangle controls are described below.

Pop-Up Menu Controls

For enhancements to Pop-Up Menu controls, see the Menu Manager chapter.

Edit Line Controls

The parameter count in an edit line control template can now be 8 or 9 (previously, 8 was the only valid value). If the paramter count is 9, the edit line is "password" style, and 9th parameter is a one-word field giving the password character to display for each character entered.

Icon Button Controls

Bit 3 of the Icon Button Flags parameter makes the button **not** track or return hits when you click in it.

List Controls

To get the full functionality of the `memNever` bit (bit 5 of the member flags) in an item in a list control, you must set the `testMemNever` bit (bit 6) in the List Control's `ctlFlag` field. (This has been true since System Software 5.0.)

Scroll Bar Controls

If your program changes color tables, it is best to change them before creating controls that will be used while those color tables are active. For example, scroll bar controls examine the current color tables looking for a suitable gray pattern for the page regions. The old checkerboard pattern is used if no suitable gray is available.

If you change color tables while scroll bars already exist, you should call `CtlNewRes` so the Control Manager has a chance to notice and use an appropriate gray or checkerboard pattern for any scroll bars that exist.

Scroll bars use `WaitUntil` in the Miscellaneous Tools to limit the scrolling speed to 15 control value changes per second.

Static Text Controls

Setting bit 2 (value $0004, `fBlastText`) in the `ctlFlag` field of a static text control makes the control draw much faster but puts the following restrictions on the control:
•No string substitutions are performed
•No imbedded formatting characters are allowed
•No word wrapping is attempted
•The control is not clipped to its bounding rectangle, so you need to be sure the text fits
•The system does not erase the unused part of the control rectangle for you, as it does when you do not set this bit.

Setting bit 3 (value $0008, `fTextCanDim`) in the `ctlFlag` field of a static text control makes the text gray out when the control is inactive (either because its hilite value is $00FF, or because the window is inactive). Setting `fTextCanDim` is not recommended for large amounts of text, because system draws the text normally before graying it out. It's okay to use `fTextCanDim` in conjunction with `fBlastText`.

Resource-based Controls

Since `LoadResource` now re-locks handles, one section of Apple IIGS Technical Note #81 is now obsolete. Custom control defprocs no longer need to worry about getting called while their code is purgeable.

`HiliteControl`

`HiliteControl` now uses `WaitUntil` in the Miscellaneous Tools to limit how fast a control can blink on and off. (When `HiliteControl` sets the hilite state of the most-recently-hilited control to zero, it enforces a minimum wait of 4 ticks since the first hilite.)

`SendEventToCtl`

`SendEventToCtl` used to offer events to all extended controls. Now it ignores controls that are invisible.

`MakeNextCtlTarget`

`MakeNextCtlTarget` is responsible for cycling to the next targetable control when the user hits Tab. If the Command key is down, `MakeNextCtlTarget` now cycles in the opposite direction.

(If you have written a targetable custom control, you should call `MakeNextCtlTarget` for Command-Tab as well as Tab, if you aren't already doing so.)

NIL Window Pointers

Several calls let you pass NIL to act on the front window. These include `GetCtlHandleFromID`, `SendEventToCtl`, `NotifyCtls`, `FindCursorCtl` (in the Window Manager), `FindRadioButton`, `GetLETextByID`, and `SetLETextByID`.

`MakeNextCtlTarget` always acts on the front window.

New Control type: Rectangle

The Rectangle control defproc is resource ID $07FF0003 in Sys.Resources.

One use for the Rectangle control is to draw boxes around groups of related controls. You can detect mouse clicks within a Rectangle control if you want, but typically you will set the hilite value to $FF to disable hit testing.

You may want to punch some some text through the top of the rectangle. Just make sure your text control appears earlier than the Rectangle control in your control list. Since controls are drawn in the order opposite from how they were created, the rectangle will draw first, and then the text will punch a hole in the rectangle.

By the way, if you make the height or width of the rectangle very small (but not zero!), you can use the Rectangle Control to put horizontal or vertical lines in your windows.

Rectangle control template:

```
    |-----------------|
$00 |     pCount      |   Word—parameter count (6, 8, 9, or 10)
    |-----------------|
$02 |       ID        |   Long—Application-assigned control ID
    |-----------------|
$06 |      rect       |   Rect—boundary rectangle for control
    |-----------------|
$0E |     procRef     |   Long—Rectangle control = $87FF0003
    |-----------------|
$12 |      flag       |   Word—flags (see below)
    |-----------------|
$14 |    moreFlags    |   Word—additional control flags (see below)
    |-----------------|
$16 |     refCon      |   Long—application-assigned constant
    |-----------------|
$1A |    penHeight    |   *Word—pen height
    |-----------------|
$1C |    penWidth     |   *Word—pen width in 640-mode pixels
    |-----------------|
$1E |    penMask      |   *8 bytes—pen mask to draw rectangle with
    |-----------------|
$26 |   penPattern    |   *32 bytes—pen pattern to draw rectangle with
    |-----------------|
```

`flag`:

bit 7: 1= invisible

bits 1-0: 00=transparent control (doesn't draw anything, but can still do hit testing)
01=gray pattern
10=black pattern
11=reserved

`moreFlags` `fCtlTarget`, bit 15: must be 0
`fCtlCanBeTarget`, bit 14: must be 0
`fCtlWantEvent`, bit 13: must be 0
`fCtlProcRefNotPtr`, bit 12: must be 1
`fCtlTellAboutSize`, bit 11: must be 0
`fCtlIsMultiPart`, bit 10: must be 0
bits 9-0: reserved, must be 0

The default pen height is 1. The default pen width is 2 640-mode pixels. The pen width is always cut in half for 320 mode. You should include or omit the `penHeight` and `penWidth` parameters as a group (a parameter count of 7 is invalid).

If you provide the `penPattern` parameter, it overrides the pattern specified in the flags (but you should set the flags to 1 so it the control won't be transparent).

New Control type: Thermometer

The Thermometer control defproc is resource ID $07FF0002 in Sys.Resources.

A thermometer control is a rectangle the gradually fills as an operation completes. At convenient intervals, your application calls `SetCtlValue` on the thermometer control, passing values from 0 up to the data value you pass in the template.

The default color table provides a white rectangle, outlined in black, which fills with red. (A value of 0 is completely white, and a value equal to data is completely red.)

Thermometer control template:

```
    |-----------------|
$00 |     pCount      |   Word—parameter count (8 or 9)
    |-----------------|
$02 |       ID        |   Long—Application-assigned control ID
    |-----------------|
$06 |      rect       |   Rect—boundary rectangle for control
    |-----------------|
$0E |     procRef     |   Long—Thermometer control = $87FF0002
    |-----------------|
$12 |      flag       |   Word—flags (see below)
    |-----------------|
$14 |    moreFlags    |   Word—additional control flags (see below)
    |-----------------|
$16 |     refCon      |   Long—application-assigned constant
    |-----------------|
$1A |     value       |   Word—determines position of mercury
    |-----------------|
$1C |      data       |   Word—determines scale
    |-----------------|
$20 | *colorTableRef  |   Long—color table reference
    |-----------------|
```

```
flag:  bit 0:  0=vertical, 1=horizontal
       bits 1-15: reserved
```

`moreFlags` `fCtlTarget`, bit 15: must be 0
`fCtlCanBeTarget`, bit 14: must be 0
`fCtlWantEvent`, bit 13: must be 0
`fCtlProcRefNotPtr`, bit 12: must be 1
`fCtlTellAboutSize`, bit 11: must be 0
`fCtlIsMultiPart`, bit 10: must be 0
bits 9-2: reserved, must be 0
Color table reference, bits 1-0: Defines type of reference in `colorTableRef` : 00=pointer, 01=handle, 10=rCtlColorTbl resource

The color table is 4 words long in this format:
`$000w` w is the outline color

`$000x`	x is the interior color
`$000y`	y is the foreground mercury color for a dotted pattern
`$p00z`	z is the mercury color; p is 0 for solid, 8 for dotted pattern (that is, set bit 15 for a dotted mercury pattern)

The default color table is:

`$0000`	black outline
`$000F`	white interior
`$0000`	
`$0004`	solid red mercury

Note: `GetCtlTitle` and `SetCtlTitle` deal with the `ctlData` field of the control record, so you can use them to examine and change the scale of a thermometer control you have already created. Only the low word of the value is significant (the high word is reserved).

FindRadioButton $3910

Note: FindRadioButton is very similar to the fmdWhichRadio call in the DTS Libraries and Tools. See the section For fakeModalDialog Users earlier in this chapter.

Returns a value indicating which radio button is selected in a given family. The value returned is the low word of the selected radio button's control ID minus the low word of the lowest radio button control ID in the family.

For example, if four radio buttons are in a window with control ID values of $00013600, $00013601, $00013602, and $0001360F, respectively, and the second radio button is currently selected, FindRadioButton returns $0001 ($3601-$3600). If the fourth radio button is selected, FindRadioButton returns $000F ($360F-$3600).

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Word—space for result
|-----------------|
|     windPtr     |   Long—window containing the radio buttons
|-----------------|
|     famNum      |   Word—Family number for the radio buttons to check
|-----------------|
                  <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|     radioNum    |   Word—value indicating selected radio button
|-----------------|
                  <—SP
```

Errors none

```
C           extern pascal unsigned int FindRadioButton(windPtr,famNum);
            WindowPtr   windPtr;
            Word        famNum;
```

windPtr Pointer to the window containing the radio buttons to check. (FindRadioButton works with any window, not just windows used

with `DoModalWindow`.) You may pass NIL to work with the front window.

`famNum` Family number of the radio buttons to check.

`radioNum` Calculated value indicating which radio button is selected in the indicated family. If there is no active radio button in the specified family, radioNum is $FFFF.

GetLETextByID $3B10

Note: `GetLETextByID` is very similar to the `fmdLEGetText` call in the DTS Libraries and Tools. See the section For fakeModalDialog Users earlier in this chapter.

Returns the text of an Edit Line control into a buffer supplied by the caller.

`GetLETextByID` saves you the trouble of calling `GetCtlHandleFromID` to get the Edit Line control handle, retrieving the Line Edit record from the control (using `GetCtlTitle`) and then making Line Edit tool calls to actually retrieve the text.

The text is returned with a length byte at the beginning and a zero byte at the end. You can use the text as a Pascal-style string starting at the buffer's beginning or as a C-style string starting at the buffer's second byte. Pascal strings of 256 bytes will have a length byte of zero, but are still retrievable as C strings.

Important: `GetLETextByID` does no checking for buffer sizes; it simply assumes that there is enough memory at the specified address to hold all the text from the Edit Line control (the maximum number of characters possible in the Edit Line control, plus the Pascal length byte and C terminating zero byte).

`GetLETextByID` also does no checking to insure that the control ID specified belongs to an Edit Line control. Specifying the control ID of anything other than an Edit Line control is a bad thing.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|     windPtr     |   Long—Pointer to the window containing the control
|-----------------|
|     LECtlID     |   Long—Control ID for the Edit Line control
|-----------------|
|     textPtr     |   Long—Pointer to result buffer
|-----------------|
                  <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                  <—SP
```

Errors Control Manager errors returned unchanged

```
C           extern pascal void GetLETextByID(windPtr, LECtlID,
textPtr);
            WindowPtr   windPtr;
            Long        LECtlID;
            StringPtr   textPtr;
```

`windPtr` Pointer to the window that contains the Edit Line control. (`GetLETextByID` can be used to retrieve from any window, not just the active one.) You may pass NIL to work with the front window.

`LECtlID` The control ID of the Edit Line control from which to retrive the text.

`textPtr` Pointer to a buffer where the text will be returned. The text is preceded by a length byte and terminated by a zero byte.

Note: This call is in the Control Manager instead of Line Edit because it works with Edit Line controls and **not** Line Edit records.

SetLETextByID $3A10

Note: SetLETextByID is very similar to the fmdLESetText call in the DTS Libraries and Tools. See the section For fakeModalDialog Users earlier in this chapter.

Sets the text of an Edit Line control to a string supplied by the caller, selects all of the text, and invalidates the viewRect of the Line Edit record referenced in the control. This normally causes the new text to be redrawn on the next update event.

SetLETextByID saves you the trouble of calling GetCtlHandleFromID to get the Edit Line control handle, retrieving the Line Edit record from the control (using GetCtlTitle) and then making Line Edit tool calls to actually set the text and the selection.

Important: SetLETextByID does no checking to insure that the control ID specified belongs to an Edit Line control. Specifying the control ID of anything other than an Edit Line control is a bad thing.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|     windPtr     |   Long—Pointer to the window containing the control
|-----------------|
|     LECtlID     |   Long—Control ID for the Edit Line control
|-----------------|
|     textPtr     |   Long—Pointer to Pascal string
|-----------------|
                  <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                  <—SP
```

Errors Control Manager errors returned unchanged

```
C           extern pascal void SetLETextByID(windPtr, LECtlID,
textPtr);
            WindowPtr   windPtr;
            Long        LECtlID;
            StringPtr   textPtr;
```

`windPtr` Pointer to the window that contains the Edit Line control to receive the text. (`SetLETextByID` can be used to set text in any window, not just the active one.) You may pass NIL to work with the front window.

`LECtlID` The control ID of the Edit Line control to receive the text.

`textPtr` Pointer to a Pascal-style string to be used as the text.

Note: This call is in the Control Manager instead of Line Edit because it works with Edit Line controls and **not** Line Edit records.

005 Desk Manager Update

New Features of the Desk Manager

Classic Desk Accessory changes

•If bit 0 of battery-RAM location $5F is set, the Desk Manager sorts the CDA menu alphabetically. However, `Control Panel` always remains at the top, and `Quit` always remains at the bottom. (There is a check box in the General Control Panel to enable and disable this feature. By default, sorting is enabled.)

•Typing a non-control key at the CDA menu moves the selection bar to the next CDA name that begins with that character, if any. Upper and lowercase letters are considered the same, and the search wraps around to the beginning. If you type a letter that no CDA name starts with, the system calls `SysBeep2`($8008).

•The following keyboard shortcuts in the CDA menu are not new, but they were not documented before: Command-up-arrow moves up one page of CDAs (or to the top); Command-down-arrow moves down one page (or to the bottom); Esc moves to the `Quit` item at the bottom.

•CDA names are now allowed to contain Control-N. Any part of the name following the Control-N is displayed as normal text even when the CDA is hilited in inverse. (This worked before System 5.0 but was not supported; now it works again and is supported.)

New Desk Accessory changes

•If bit 0 of battery-RAM location $5F is set, `FixAppleMenu` inserts NDA menu items into the menu in alphabetical order.

•`DeskStartUp` checks to see if sufficient tools are already started up. If not, it returns without doing anything.

•`FixAppleMenu` checks to see if the Desk Manager was successfully started. If not, it tries to start it up again. (Some applications start tools in a poor order, and the cooperation between `DeskStartUp` and `FixAppleMenu` solves many compatibility problems—by the time the application calls `FixAppleMenu`, the proper tools have been started.)

•A successful `DeskStartUp` calls `SendRequest` $0502, `systemSaysDeskStartUp`; `DeskShutDown` calls `SendRequest` $0503, `systemSaysDeskShutDown` (`dataIn` and `dataOut` are reserved). This gives any part of the system a chance to take action at `DeskStartUp` or `DeskShutDown` time—this was previously easy only for desk accessories.

•`FixAppleMenu` calls `SendRequest` $051E, `systemSaysFixedAppleMenu` (`dataIn` and `dataOut` are reserved). At this point, it is possible for an NDA to add an icon to its Apple-menu item, by calling `SetMItemStruct` and `SetMItemIcon`. (The NDA needs to look in its own NDA header to determine what menu item ID it has been assigned.)

•`SystemEvent` intercepts key-down and auto-key events for Command-W when a System window is in front, and it calls `CloseNDAByWinPtr` on the front window. NDAs and applications never see Command-W presses when a System window is in front, and the user can always close an NDA by typing Command-W.

Note: Before `SystemEvent` calls `CloseNDAByWinPtr` on the system window to be closed, it offers `optionalCloseAction` ($000B) to the NDA's action procedure (see `CallDeskAcc`). This gives the NDA a chance to ask the user if they want to save changes, and even to abort the close operation. To tell `SystemEvent` that everything is taken care of, the action procedure stores a $0001 at the word pointed to by the data value passed.

•When `SystemClick` detects a click in a System window's (frame) grow box, is calls `GrowWindow` and normally enforces a minimum width of 78 and a minimum height of 34. If special minimum-width and minimum-height values are present in the window's auxiliary window information record, `SystemClick` uses those values instead (see `GetAuxWindInfo` in the Window Manager chapter). `SystemClick` does not do anything for grow box controls in a window's content, as created by `NewControl2`.

•The `qContent` window frame bit now works for system windows. When `SystemClick` detects a click in a system window that is not frontmost, it has always called `SelectWindow` to bring it to the front. Now it continues by checking the `qContent` bit in the window's frame. If `qContent` is set, `SystemClick` processes the click as if the window was already in front.

•The Desk Manager now knows how to handle System windows which were not returned from any NDA's Open procedure. If the window pointer is not found in the table of open NDA windows, the Desk Manager calls `GetAuxWindInfo` and looks at offset +024 for a pointer to an structure with the following format:

`+000 Word status`	Use $0000 (reserved for the Desk Manager)
`+002 Long openProc`	reserved (use 0)
`+006 Long closeProc`	pointer to the NDA-style Close routine
`+010 Long actionProc`	pointer to the NDA-style Action routine
`+014 Long initProc`	reserved (use 0)
`+018 Word period`	reserved (use 0)
`+020 Word eventMask`	event mask, just like for an NDA
`+022 Long lastServiced`	reserved (use 0)
`+026 Long windowPtr`	reserved (use 0)
`+030 Long ndaHandle`	reserved (use 0)
`+034 Word memoryID`	Your memory ID (very important for resource-app switching!)

This allows NDAs to have more than one (modeless) window. It also allows Finder Extensions or other things other than NDAs to create system windows and handle events in them.

`CloseNDAbyWinPtr`

`CloseNDAbyWinPtr` works for any system window, not just NDA windows.

When to use `SetSysWindow`

`SetSysWindow` marks a window as a "system window," which dramatically changes how the system handles events for that window.

When a system window is in front, many events are handled at a low level—during a `GetNextEvent` call, `SystemEvent` takes the event and feeds it to the NDA or other code responsible for that window.

If you are handling your window modally (your code keeps control until the window is dismissed), do not call `SetSysWindow`.

`SetSysWindow` should only be used for non-application windows that remain open while the application continues to run.

How to override `SystemClick`

You can now override any `SystemClick` features you don't like (for your window only). For example, if the user clicks on your system window's zoom box, you may want to toggle between two different window sizes without changing the window's location; the normal `SystemClick` response is to call `TrackZoom` and `ZoomWindow`, which doesn't do what you want.

Before `SystemClick` does anything else, it uses `CallDeskAcc` to send you a newly-defined action code. If `CallDeskAcc` is unable to send you the new action code, or if you decline to handle it, the `SystemClick` behaves as before. If you accept the action, `SystemClick` exits, taking no further action.

The action code is `sysClickAction`, code 10 ($000A). The data value is a pointer to the following structure:

```
     |-----------------|
+000 |     result      |   Word—space for result
     |-----------------|
+002 |     fwValue     |   Word—value returned from FindWindow
     |-----------------|
+004 |    windowPtr    |   Long—window pointer
     |-----------------|
+008 |   eventRecPtr   |   Long—event record pointer
     |-----------------|
```

The `fwValue`, `windowPtr`, and `eventRecPtr` fields are copies of the corresponding `SystemClick` parameters (except that bit 15, indicating a system window, is already masked off of `fwValue` for you).

If you handle the action, change the result field to $0001 (it is pre-zeroed for your convenience), and `SystemClick` exits, taking no further action.

CallDeskAcc $2405

`CallDeskAcc` calls a New Desk Accessory's Action or Init routine with the specified values. You can specify which NDA to call by index number or, if the NDA is open, by window pointer.

Warning: Be careful passing events to a desk accessory that is not open. Most action codes make sense only for an open desk accessory.

All undefined action codes are reserved for definition by Apple.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Word—space for result
|-----------------|
|      flags      |   Word—flags (defined below)
|-----------------|
|   daReference   |   Long—specifies which DA to call
|-----------------|
|     action      |   Word—action code to pass to DA (in A)
|-----------------|
|      data       |   Long—data value to pass to DA (in XY)
|-----------------|
                  <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|     result      |   Word—result returned from desk accessory
|-----------------|
                  <—SP
```

Errors $0520 `deskBadSelector` selector out of range

```
C           extern pascal Word CallDeskAcc(flags, daReference, action,
                                           data);
            Word flags, action;
            Long daReference, data;
```

`flags` bit 15: 1 = reserved for calling a CDA
0 = call an NDA
bits 14-2: reserved (use 0)

bit 1: 1 = call the NDA's Init routine
0 = call the NDA's Action routine
bit 0: 1 = daReference is a window pointer
0 = daReference is an index number (in the range 1..GetNumNDAs)

`daRefrence` Either a window pointer of an open System window, or an index number (in the range 1..`GetNumNDAs`), depending on bit 0 of flags.

`action` Value to pass to the DA in the A register (an action code if flags bit 1 is 0).

Action codes greater than 9 are sent to a DA's Action routine only if its event mask is `$Axxx`—otherwise error $0520 is returned.

An NDA with `$Axxx` for its `eventMask` promises to safely ignore any Action codes it does not recognize.

`data` Value to pass to the DA in the X and Y registers (for example, an event record pointer).

daReference Note

`GetNumNDAs` returns the number of NDAs installed. This number is also the `daReference` number of the most recently installed NDA, suitable for passing to `CallDeskAcc`.

GetDeskAccInfo $2305

GetDeskAccInfo provides safe access to certain information about Desk Accessories currently installed in the system.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      flags      |   Word—flags (defined below)
|-----------------|
|   daReference   |   Long—specifies which DA to get info on
|-----------------|
|    buffSize     |   Word—size of result buffer
|-----------------|
|    bufferPtr    |   Long—pointer to result buffer
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                   <—SP
```

Errors $0520 deskBadSelector selector out of range

C
```
extern pascal void GetDeskAccInfo(flags, daReference,
                buffSize, bufferPtr);
word  flags, buffSize;
Long  daReference;
Ptr   bufferPtr;
```

flags
bit 15: 1 = get information on an CDA
0 = get information on a NDA
bits 14-1: reserved (use 0)
bit 0: 1 = daReference is a window pointer
0 = daReference is an index number

daRefrence Either a window pointer of an open System window, or an index number (in the range 1..GetNumNDAs), depending on bit 0 of flags.

buffSize Number of bytes the result buffer can hold, not including the first two bytes (the first two bytes returned indicate the number of bytes of data following).

For information on a CDA, buffSize must be at least 4.

`bufferPtr` Pointer to the result buffer, in the following format.

Information on a CDA:

`+000` returned data size (for example, 4)
`+002` handle to CDA (4 bytes)

Information on an NDA:

`+000` returned data size
`+002` NDA status (zero if closed, nonzero if open)
`+004` pointer to NDA's Open routine
`+008` pointer to NDA's Close routine
`+012` pointer to NDA's Action routine
`+016` pointer to NDA's Init routine
`+020` NDA's period
`+022` NDA's event mask
`+024` tick count of last Run event sent to NDA
`+028` NDA's main window pointer, if any
`+032` handle to NDA
`+036` NDA's Memory Manager user ID

GetDeskGlobal $2505

GetDeskGlobal retrieves information from the Desk Manager. Only one value is currently defined.

Pass $0000 to GetDeskGlobal to get the pointer to the last window that the Desk Manager examined. This should be used inside NDA-style procedures called by the Desk Manager to determine what window is being handled.

This allows the same NDA-style procedures to be shared among several system windows.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Long—Space for result
|-----------------|
|     selector    |   Word—selects which value to retrieve
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|      value      |   Long—the selected value
|-----------------|
                   <—SP
```

Errors $0520 deskBadSelector selector out of range

```
C           extern pascal Long GetDeskGlobal(selector);
            word selector;
```

021 Dialog Manager Update

New Features of the Dialog Manager

•The standard icons for `NoteAlert`, `StopAlert`, and `CautionAlert` are now colorful.

•Setting bit 30 of the `filterProcPtr` parameter to `ModalDialog` or `ModalDialog2` causes the Dialog Manager to automatically change the cursor into an I-Beam when it is positioned over an `editLine` item. (If `ModalDialog` or `ModalDialog2` has left the cursor set to an I-Beam, `CloseDialog` restores it to an arrow.)

•`ModalDialog` no longer steals application events on ROM 3 machines (fixed by patching `GetNextEvent`).

The default `ErrorSound` procedure calls `SysBeep2` (in the Miscellaneous Tools) with one of the following codes:

Alert stage 0	`SysBeep2`($C000)
Alert stage 1	`SysBeep2`($4001)
Alert stage 2	`SysBeep2`($4002)
Alert stage 3	`SysBeep2`($4003)
click outside window	`SysBeep2`($0004)

006 Event Manager Update

New Features of the Event Manager

Patched `GetNextEvent` to fix ROM 3 ModalDialog bug, preventing `ModalDialog` from stealing app1 through app4 events.

The `GetNextEvent` patch also dispatches any deferred `SysBeep2` request (see `SysBeep2` in the Miscellaneous Tools).

`EMStartUp` and `EMShutDown` are patched to preserve the cursor location in the event of a smooth application launch, when the super-hires screen remains visible the whole time.

`EMShutDown` creates message number 6 containing the cursor location in 640-scale coordinates. `EMStartUp` uses this message to position the mouse. `QDStartUp` destroys message 6 if the super-hires screen is not already turned on.

027 Font Manager Update

New Features of the Font Manager

•`ChooseFont` can now display up to 24 font sizes within a family. (Previously the limit was 12.)

•The human interface for `ChooseFont` is improved:

- The family list and size lists are targetable controls, and Tab moves you between these and the "other size" edit line field.
- You can navigate in the lists using the up and down arrows, and you can begin typing a family name to move to that family.
- There are Command- key equivalents to toggle the Style check boxes. Escape and Command-period are tied to the Cancel button.
- If you uncheck all the style checkboxes, the Plain box automatically rechecks.

•The Font Manager can now load fonts from disk even if they are larger than 64K.

•The Font Manager can deal with font file names as long as 32 characters now. (This is not the same as family names. Family names are still limited to 25 characters.)

•Fonts are now scaled correctly even if the `owTOffset` field is in the range `$xx8000` to `$xxFFFF`. This didn't work before.

•`FMStartUp` now returns error $1B0D (`fmBadParmErr`) if you pass zero for the User ID or the direct-page address.

011 Integer Math Update

For ROM 1, `IMVersion` now returns version 3.0 for consistency with ROM 3. There are no other changes.

020 Line Edit Update

New Features of Line Edit

•Changed Edit Line Control to allow shift-clicking in Line Edit controls. Shift-clicking now extends the selection (this always worked in Line Edit records, but it didn't work for Edit Line controls).

•Fixed a problem where `LETextBox` would strip a random amount of stuff from the stack if called with a Length parameter of zero.

•Refer to the Control Manager section for new features of extended Edit Line controls.

•The default Password character is now a hollow diamond instead of an asterisk.

•Line Edit fields now scroll horizontally as you type or drag the mouse.

•The `leHiliteHook` and `leCaretHook` features now work properly even if the supplied routine starts at the beginning of a bank (`$xx0000`).

028 List Manager Update

New Features of the List Manager

Speed Improvements

The List Manager no longer bothers calling the member draw routine for members that will be completely clipped out (the technique in Apple IIGS Technical Note #74 is now obsolete, since the List Manager is doing something equivalent).

Standard `listDraw` routine

The standard `listDraw` routine draws characters closer together (using `SetCharExtra`) if the Pascal or C string being drawn is too wide to be displayed completely.

`SortList` and `SortList2`

•If bit 31 of `compareProc` is set for `SortList` or `SortList2`, the compare procedure is expected to return the result on the stack rather than in the carry flag. This makes it easier to write custom compare procedures in Pascal and C.

The system provides a word of result space just deeper than the RTL address. The compare procedure must set bit 0 when an old-style compare procedure would have returned with the carry flag set.

•If you pass $00000001 for `compareProc` to `SortList` or `SortList2`, the List Manager does a case-insensitive sort for you (NIL is still a case-sensitive sort). In addition to ignoring case, the sort also ignores accent marks on characters and treats certain typographical characters as their similar ASCII counterparts. See `CompareStrings` later in this chapter for a complete list of translations.

`NewList2`

•You can now pass $FFFFFFFF as the `NewList2` `drawProcPtr` to leave the old value unchanged).

Targetable List Controls

Extended list controls can now be target controls. If you set the `fCtlCanBeTarget` and `fCtlWantEvents` bits in your list control's `ctlMoreFlags`, the list becomes the target when you Tab to it or click in it or its scroll bar. While a list is the target, it has a bold outline (a "focus frame"), and the control automatically calls `ListKey` with any keystrokes it receives (except for Return and Tab).

If your list would be the only targetable control in the window, there is no need to make it targetable. Just set the `fCtlWantEvents` bit, leaving `fCtlCanBeTarget` clear.

`ctlFlag` Clarification

Toolbox Reference Volume 1, page 11-9, describes the `ctlFlag` of a List control record as "style of scroll bar." In fact, ctlFlag looks like this:

bit 7: control is invisible
bit 6: `testMemNever` (see above)
bits 5-2: reserved (should be zero)
bit 1: `fListSelect` from template's `listType` field (1 for single-select mode)
bit 0: `fListString` from template's`listType` field (1 for C strings)

`memNever` Note

To use the `memNever` bit (bit 5 in the member flags byte of each record), you should also set the `testMemNever` bit in the List Control's `ctlFlag` field.

If you don't set `testMemNever`, clicking on a member selects the member even if its `memNever` bit is set.

(Note: This has always been true. It is not a new feature of 6.0.)

CompareStrings $181C

CompareStrings compares two Pascal strings, using the same comparison criteria that SortList and SortList2 use when you pass $00000001 for compareProc.

That is, the comparison is case-insensitive, and it treats foreign characters and special typographical characters in a reasonable way. For example, accented characters are treated as similar unaccented characters, typographical quotation marks (“”) are treated as normal quotation marks (""). See the table below for a complete list of translations.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      space      |   Word—Space for result
|-----------------|
|      flags      |   Word—Flags (reserved, use zero)
|-----------------|
|     String1     |   Long—Pointer to first Pascal string
|-----------------|
|     String2     |   Long—Pointer to second Pascal string
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|      Order      |    Word—0 if equal, $FFFF if 1<2, 1 if 1>2
|-----------------|
                   <—SP
```

Errors none

```
C           extern pascal Word CompareStrings(flags, String1, String2);
            Word  flags;
            Ptr   String1, String2;
```

`flags` Reserved; must be zero.

`String1` Pointer to first Pascal string.

`String2` Pointer to second Pascal string.

`Order` Zero if the two strings are equal. $FFFF if `String1` comes before `String2`. $0001 if `String1` comes after `String2`.

List of Translations

The following translations occur internally during the comparision. The original strings are not modified.

•a..z become A..Z.
•Characters $80..$9F, $CB, $CC, $CD become unaccented capital letters.
•Character $A2 (cents) becomes “C”.
•Character $A7 (ß) becomes “B”.
•Character $AB (a left-slanting apostrophe) becomes an apostrophe.
•Character $AF becomes “0”.
•Character $B4 becomes “Y”.
•Character $BE becomes $AE.
•Character $BF becomes “0”.
•The “<<” and “>>” characters become quotation marks.
•Character $CA (nonbreaking space) becomes a space.
•Character $CF becomes $CE.
•Character $D0 becomes “-”
•Character $D1 (dash) becomes a hyphen.
•Typographical quotes become plain quotes.
•Typographical single-quotes become apostrophes.
•Character $D8 (y-umlat) becomes a Y.
•Characters $D9..$F5 are not in Shaston, but are translated appropriately for international purposes.
•Other characters remain unchanged.

ListKey $171C

ListKey accepts keystrokes and jumps the selection around in the specified list appropriately. Arrows are supported, and "prefix strings" of up to 32 characters are supported. For prefix strings to work in a reasonable way, the list must be sorted (as with SortList or SortList2 with a compareProc of $00000001, a case-insensitive sort).

Note: If you are using extended list controls, you do not normally need to use ListKey. Instead, set the fCtlWantEvents and fCtlCanBeTarget bits in your control's ctlMoreFlags field, and the list control calls ListKey for you automatically.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      flags      |   Word—flags
|-----------------|
|   theEventRec   |   Long—Pointer to a filled-in event record
|-----------------|
|   listCtlHndl   |   Long—Handle of the list control
|-----------------|
                  <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                  <—SP
```

Errors $1C02 listRejectEvent list did not handle the event

```
C           extern pascal Handle
ListKey(flags,theEventRec,listCtlHndl);
            Word        flags;
            EventRecPtr theEventRec;
            CtlRecHndl  listCtlHndl;
```

flags Bits 15-1 are reserved and should be 0.
Bit 0 is set if ListKey should ignore the first character of every string in the list. This is provided for lists of Volumes and Devices, like Standard File's volume list.

theEventRec Pointer to a valid event record. If the event is a keyDown or autoKey event, ListKey may select a different item in the specified list.

`listCtlHndl` Control Handle for the List Control the user sees as the active one. This can be either a standard list control or an extended list control (see Note above about extended list controls).

Notes

Before you call `ListKey`, the `QuickDraw` port should already be set to the window containing your list control.

For keyboard navigation in the list to work as expected, your list items must be sorted into case-insensitive alphabetical order, like `SortList2` sorts them when you pass $00000001 for the `compareProc`.

`ListKey` ignores events other than `keyDownEvt` and `autoKeyEvt`. You can pass other kinds of events, but it doesn't accomplish anything.

Since a Task Record or Extended Task Record begins with a regular Event Record, you can pass a pointer to any of these structures as the `theEventRec` parameter.

038 Media Controller

New for System 6.0. See separate documentation.

New Features of the Memory Manager

•Fixed a problem where a long hang and then a crash could result if an Out-of-Memory-Queue routine freed up the request number of bytes on the second pass, but the memory request still could still not be satisfied (because of fragmentation or special attributes of the handle being allocated or manipulated).

•Fixed a problem where, in rare cases, the "high hint handle" (usually the last-allocated non-fixed handle) and "low hint handle" (usually the last-allocated fixed handle) could cross and then become equal. After that happened, certain operations (like `DisposeHandle`) on the hint handle left the system in a delicate state: If the next handle allocation was for a non-fixed handle, the system would crash.

New Memory Manager Calls

`SetHandleID` $3002

`SetHandleID` provides a supported way to determine and optionally change the User ID associated with a Memory Manager handle.

To determine a handle's User ID without changing it, pass zero for the `newID` parameter. The previous ID is always returned, whether the ID is changed or not.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Word—Space for old ID
|-----------------|
|      newID      |   Word—new ID for handle (0 to leave unchanged)
|-----------------|
|    theHandle    |   Long—handle
|-----------------|
                  <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|      oldID      |   Word—handle's previous ID
|-----------------|
                   <—SP
```

Errors none

```
C           extern pascal word SetHandleID(newID,theHandle);
            Word  newID;
            Handle theHandle;
```

Note: `SetHandleID` is useful when a Control Panel needs to keep a chunk of code around while its window is not open:

1. Use `GetCodeResConverter` to get the address of the code resource converter
2. Use `ResourceConverter` to log the convert in for a particular resource type
3. Use `LoadResource` to load a code resource
4. Use `DetachResource` to prevent the resource from being disposed when the file is eventually closed
5. Use `GetNewID` to allocate a new memory ID for the chunk of code
6. Use `SetHandleID` to change the code's memory ID to the newly allocated one (so that when the system disposes of all memory using the Control Panel's memory ID, the code will not be disposed)

015 Menu Manager Update

New Features of the Menu Manager

•Pop-up menu controls now support `ctlMoreFlags` bits 7 ($0080), `fDrawPopDownIcon`, to draw a down-pointing triangle at the right edge when the menu is not popped up; and bit 5 ($0020), `fDrawIconInResult`, to draw the current menu item's icon when the menu is not popped up.

•The new call `InsertPathMItems` builds a menu, complete with icons, from a GS/OS pathname. Used by Standard File and the Finder already.

•When a menu item is blinking, the speed is now limited using `WaitUntil` in the Miscellaneous Tools. This way an accellerated machine does not blink the item too fast to see.

•Patched `EnableMItem` and `DisableMItem` on ROM 3 only to simulate a dispatcher error ($0001) when the Menu Manager has not been started up (for compatibility with a broken 3rd-party application).

•`MenuStartUp` now sets the menu item blink count from bits 4-3 of Battery RAM location $5E (the range is zero to three). Previously, the count was always three after `MenuStartUp`.

•When `MenuKey` receives a keypress with the Command key down but no menu item can be found with a matching key equivalent, `MenuKey` calls `SendRequest` with request code `systemSaysMenuKey` ($0F01) and `dataIn` equal to the Task Record pointer that was passed to `MenuKey`. This provides a way for desk accessories to have key equivalents without accidentally overriding an application's menu item key equivalents.

If the `systemSaysMenuKey` broadcast is accepted, `MenuKey` changes the what field of the event record to be a null event to prevent the application from taking any further action on the event.

(`MenuKey` does `systemSaysMenuKey` only if the Desk Manager was successfully started, the current menu bar is the System menu bar, and the system event mask allows posting of Desk Accessory events.)

•`InsertMenu` now returns error $0F04, `dupMenuID`, if a menu being inserted has the same menu ID as another menu already in the same menu bar. Previously , no error was returned, but the system would later hang inside `FixMenuBar`.

•`HideMenuBar` changes the SCBs only for the scanlines from 0 to `MenuHeight`-1. (It used to call `SetAllSCBs`.)

Icons in Menu Items

The menu manager now supports icons in menu items (including pop-up menu items). Several calls have been added: `SetMItemIcon`, `GetMItemIcon`, `SetMItemStruct`, `GetMItemStruct`, `RemoveStruct`, `SetMItemFlag2`, and `GetMItemFlag2`. A few old calls have been modified slightly and an additonal menu item structure has been defined.

The Menu Manager requires that QuickDraw II Auxiliary be available when icons are present in menu items.

Note: Do not create an icon with a width such that the width of the icon plus the width of the menu item's name are greater than the width of the screen.

Do not create an icon with a height greater than the height of the text in the menu item. No clipping is done when the icon is drawn.

Several new bits have been defined in the `itemFlag` field of the menu item record. (See page 37-15 of Toolbox Ref. Vol. 3 for more details on the structure of a menu item record/template.)

`itemFlag`	bit 10	Indicates whether or not there is an additional structure associated with this menu item. 0 = no structure associated with menu item 1 = there is an additional structure associated with item
	bits 9-8	If bit 10 is set, these bits describe how this structure will be referenced. 00 = Reference is by pointer 01 = Reference is by handle 10 = Reference is by resource ID 11 = Invalid value

When bit 10 is set the menu item record is defined as follows:

Menu Item Record

```
    |-----------------|
$00 |     version     |   Word—Version number for template, must be 0
    |-----------------|
$02 |     itemID      |   Word—Menu item ID
    |-----------------|
$04 |    itemChar     |   Byte—Primary keystroke equivalent character
    |-----------------|
$05 |   itemAltChar   |   Byte—Alternate keystroke equivalent character
    |-----------------|
$06 |    itemCheck    |   Word—Character code for checked items
    |-----------------|
$08 |    itemFlag     |   Word—Menu item flag word
    |-----------------|
$0A |  itemStructRef  |   Long—Reference to new structure (not
    |-----------------|       to item's name)
```

`itemStruct` Record:

```
    |-----------------|
$00 |    itemFlag2    |   Word—Bit flags that control attributes of
    |-----------------|       this structure
$02 |   itemTitleRef  |   Long—Reference to item name
    |-----------------|
$06 |   itemIconRef   |   Long—Reference to icon associated with item
    |-----------------|
```

Important: An `itemStruct` record is not just a template! Your menu item contains a reference to the `itemStruct`, so the ten-byte `itemStruct` structure must remain available. (For example, if your `itemStruct` is referenced by pointer, make it a global variable, not a stack-based local variable!)

This also means you can't share the same `itemStruct` among multiple menu items.

If your `itemStruct` records are referenced as `rItemStruct` resources, note that the Menu Manager makes their handles purgeable after each use. If you use `SetMItemIcon`, `SetMItemName`, or `SetMItemFlag2` and expect the results to "stick," you must mark your `rItemStruct` resources as Locked so they will remain in memory even after they are marked purgeable.

`itemFlag2` bit 15 — Indicates whether or not there is an icon associated with the menu item.
0 = No icon
1 = There is an icon

bits 14-2 Reserved. Must be set to 0. In the future these bits will define additonal fields that may be added to this record.

bits 1-0 Defines how the icon is referenced
00 = Reference is by pointer
01 = Reference is by handle
10 = Reference is by resource ID
11 = Invalid value

`ItemTitleRef` Since the reference to the `itemStruct` record is now stored in the `itemName` field of the item record, the reference to the item's name has been moved here. The bits that normally define how this field will be referenced are still controlled in the `itemFlag` field of the item record.

`ItemIconRef` This is the reference to the icon data structure. The structure itself is defined in Appendix E, page 48, of the Toolbox Reference Vol. 3.

The following existing calls have been modified to work with the new `itemStruct` record.

All these calls still perform as documented. Internally the call has changed to accommodate the possibility that the menu item may now have an `itemStruct` record associated with it.

`SetMItem`	$240F
`SetMItem2`	$410F
`GetMItem`	$250F
`SetMItemName`	$3A0F
`SetMItemName2`	$420F
`CalcMenuSize`	$1C0F

New Menu Manager Calls:

GetMItemBlink $4F0F

GetMItemBlink returns the current menu item blink setting, as set with SetMItemBlink.

The default menu item blink setting, from 0 to 3, is stored in bits 3-4 of Battery RAM location $5E.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Word—Space for result
|-----------------|
                  <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|      count      |   Word—number of times selected menu items blink
|-----------------|
                  <—SP
```

Errors none

```
C           extern pascal Word GetMItemBlink();
```

GetMItemFlag2 $4C0F

Returns the `itemfFag2` field for the `itemStruct` record associated with the menu item indicated. If bit 10 is not set then the value returned is not valid.

Note: To use this call on a menu item inside a pop-up menu, you must first set the current menu bar to be your pop-up control.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Word—Space for result
|-----------------|
|      itemID     |   Word—ID of menu item
|-----------------|
                    <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|    itemFlag2    |   Word—itemFlag2 field from itemStruct
|-----------------|
                    <—SP
```

Errors	$0F03 `menuNoStruct`	This error is returned if bit 10 of `itemFlag` is not set.

```
C           extern pascal Word GetMItemFlag2(itemID);
            Word  itemID;
```

GetMItemIcon $480F

Returns the reference to the icon associated with menu item indicated. Zero is returned if bit 10 of `itemFlag` is set to zero.

Note: To use this call on a menu item inside a pop-up menu, you must first set the current menu bar to be your pop-up control.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Long—Space for result
|-----------------|
|      itemID     |   Word—ID of menu item
|-----------------|
                    <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|     iconRef     |   Long—reference to icon
|-----------------|
                    <—SP
```

Errors	$0F03	`menuNoStruct`	This error is returned if bit 10 of `itemFlag` is not set.

```
C           extern pascal Ref GetMItemIcon(itemID);
            Word  itemID;
```

GetMItemStruct $4A0F

Returns the reference to the `itemStruct` record of the menu item specified. If there is no structure, i.e. bit 10 of `itemFlag` is set to zero then zero will be returned as the reference.

Note: To use this call on a menu item inside a pop-up menu, you must first set the current menu bar to be your pop-up control.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Long—Space for result
|-----------------|
|      itemID     |   Word—ID of menu item
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|  itemStructRef  |   Long—reference to itemStruct
|-----------------|
                   <—SP
```

Errors	$0F03	`menuNoStruct`	This error is returned if bit 10 of `itemFlag` is not set.

```
C           extern pascal Ref GetMItemStruct(itemID);
            Word  itemID;
```

InsertPathMItems $500F

`InsertPathMItems` takes a GS/OS pathname and inserts one menu item into the specified menu for each segment of the pathname. Each item has an appropriate icon next to it: either a Folder (open or closed) or a device icon (for example, a hard drive, a 3.5" disk, a 5.25" disk, an AppleShare server, a RAM Disk, or a CD-ROM).

The GS/OS pathname you pass to `InsertPathMItems` should refer to a volume or directory, not a file.

After `InsertPathMItems` inserts all the necessary items, it calls `CalcMenuSize` for you automatically. There is no need to call `CalcMenuSize` separately unless you add or remove more items.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      flags      |   Word—Flags (see description)
|-----------------|
|     pathPtr     |   Long—Pointer to class-one pathname
|-----------------|
|    deviceNum    |   Word—Device number the path is on, if known
|-----------------|
|     menuID      |   Word—Menu ID of menu to insert into
|-----------------|
|     afterID     |   Word—Menu Item ID of item to insert after
|-----------------|
|    startingID   |   Word—Menu Item ID to use for first item inserted
|-----------------|
|    resultPtr    |   Long—Pointer to result buffer (see below)
|-----------------|
                  <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                  <—SP
```

Errors errors from `InsertMenuItem2` returned unchanged
Memory Manager errors returned unchanged

```
C           extern pascal void InsertPathMItems(flags, pathPtr,
                  deviceNum, menuID, afterID, startingID, resultPtr);
            Word  flags;
            GSOSStr255Ptr pathPtr;
            Word  deviceNum, menuID, afterID, startingID;
            Ptr   resultPtr;
```

`flags` bit 0: 0 = insert items with device at bottom
1 = insert items with device at top
bit 1: reserved (use 0)
bit 2: 0 = use closed folder icons
1 = use open folder icons
bit 3: 0 = call `GetDevNumber` to find the device
1 = the `deviceNum` parameter is valid
bit 4: 0 = do not assume `pathPtr` points to a fully-expanded pathname
1 = `pathPtr` is already fully expanded (like `ExpandPath` result)
bits 15-5: reserved (use 0)

`pathPtr` Pointer to class-one GS/OS pathname

`deviceNum` The GS/OS device number of the device corresponding to the pathname in `pathPtr`, if known. You must set bit 3 of flags for `InsertPathMItems` to pay attention to `deviceNum`. By supplying this information, you can save `InsertPathMItems` the trouble of calling `GetDevNumber` (which can cause disk access and take a significant amount of time).

If you pass $FFFF for `deviceNum`, `InsertPathMItems` uses a grayed-out disk icon to indicate that the volume is offline.

`menuID` The `MenuID` of the menu to insert into (passed to `InsertMItem2`)

`afterID` The Menu Item ID to insert **after**, in the specified menu (zero to insert at the top of the menu).

`startingID` The Menu Item ID for the first item to be inserted (item numbers build up sequentially from there).

`resultPtr` Pointer to a 10-byte buffer with the following format:
+000 WORD highest Menu Item ID used
+002 LONG first handle to be disposed after menu items are removed
+006 LONG second handle to be disposed after menu items are removed

Note: The items are always inserted working from left to right in the pathname, regardless of the setting of flags bit 0. The first menu item inserted (the "device" item) gets `startingID`, the second gets startingID+1, etc.

RemoveMItemStruct $4B0F

This call removes the `itemStruct` record from the item record. Bit 10 of the `itemFlag` is set to zero, bits 8 and 9 are set to zero, and the `itemTitleRef` field is copied from the `itemStruct` record back to the item record. If bit 10 is not already set then this call does nothing. If removing the `itemStruct` record will change the appearance of the menu item then `CalcMenuSize` must be called after `RemoveItemStruct`.

Note: this call does not dispose of the memory used for the `itemStruct` record.

Note: To use this call on a menu item inside a pop-up menu, you must first set the current menu bar to be your pop-up control.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|     itemID      |   Word—ID of menu item
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                   <—SP
```

Errors	$0F03	`menuNoStruct`	This error is returned if bit 10 of `itemFlag` is not set.

```
C           extern pascal void RemoveMItemStruct(itemID);
            Word  itemID;
```

SetMItemFlag2 $4D0F

Sets the `itemFlag2` field for the `itemStruct` record of the indicated menu item to the value passed. If you wanted to keep the existing bit settings the same then you must first call `GetMItemFlag2`, "OR" in your bit settings and then pass this value.

If you set or reset any bits that might change the appearance of a menu item then you must call `CalcMenuSize` after the `SetMItemFlag2` call.

Note: To use this call on a menu item inside a pop-up menu, you must first set the current menu bar to be your pop-up control.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|     newValue    |   Word—New value for itemFlag2 field
|-----------------|
|      itemID     |   Word—ID of menu item
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                   <—SP
```

Errors	$0F03	`menuNoStruct`	This error is returned if bit 10 of `itemFlag` is not set.

```
C           extern pascal void SetMItemFlag2(newValue,itemID);
            Word  newValue;
            Word  itemID;
```

SetMItemIcon $470F

Sets the ItemIconRef field in the itemStruct record for the menu item indicated. CalcMenuSize must be called on the menu of the affected menu item after the SetMItemIcon call since the width of the menu may have changed. The parameter IconDesc is used by the call to set the itemFlag2 field correctly.

Note: To use this call on a menu item inside a pop-up menu, you must first set the current menu bar to be your pop-up control.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|    iconDesc     |   Word—describes how icon is to be referenced
|-----------------|
|     iconRef     |   Long—reference to icon
|-----------------|
|     itemID      |   Word—ID of menu item
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                   <—SP
```

Errors	$0F03 menuNoStruct	This error is returned if bit 10 of itemFlag is not set.

```
C           extern pascal void SetMItemIcon(iconDesc,iconRef,itemID);
            Word  iconDesc;
            Ref   iconRef;
            Word  itemID;
```

SetMItemStruct $490F

Sets the ItemTitleRef field of the item record to the reference for the itemStruct record passed. This call always sets bit 10 of itemFlag, and it also sets bits 8 and 9 of itemFlag to reflect the itemStructDesc parameter passed. The reference that was in the itemTitleRef field is then automatically copied over to the "new" itemTitleRef field in the itemStruct record. If the itemStruct record changes the appearance of the menu item then CalcMenuSize must be called after the SetMItemStruct call.

Note: To use this call on a menu item inside a pop-up menu, you must first set the current menu bar to be your pop-up control.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
| itemStructDesc  |   Word—describes how itemStruct is referenced
|-----------------|
|  itemStructRef  |   Long—reference to itemStruct
|-----------------|
|     itemID      |   Word—ID of menu item
|-----------------|
                  <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                  <—SP
```

itemStructDesc Bits 0-1: 00=pointer, 01=handle, 10=rItemStruct resource

Errors none

```
C           extern pascal void SetMItemStruct(itemStructDesc,
                        itemStructRef,itemID);
            Word  itemStructDesc;
            Ref   itemStructRef;
            Word  itemID;
```

032 MIDI Tools Update

(no change)

003 Miscellaneous Tools Update

New Features of the Miscellaneous Tools

•`SetVector` on ROM 1 machines now behaves just like on ROM 3 machines (there is no error checking on the vector reference number).

•Changed `UnPackBytes` to fix to a rare case where it would treat bytes past the end of your source buffer as valid packed data (part of Apple IIGS Technical Note #94 is now obsolete).

•Added the new functions `SysBeep2`, `VersionString`, `WaitUntil`, `StringToText`, `ShowBootInfo`, and `ScanDevices`.

•Whenever the Bell vector is called (for example, by `SysBeep` or by printing a Control-G through the 40- or 80-column firmware), the border blinks if either (1) the system volume is set to the lowest setting or (2) bit 0 of Battery RAM location $5E is zero (indicating that the user wants or needs visual indication of sounds). This bit can be changed with the checkbox in the Sound control panel.

New Miscellaneous Tools Calls

ConvSeconds $3703

ConvSeconds is present in System Software 5.0.3 and later, but verbs 8 and 9 were documented incorrectly.

Allows conversion to and from a long integer containing the number of seconds since January 1, 1904—the format used by the Macintosh operating system. ConvSeconds is provided to allow easier handling of dates in applications that work with several different date formats.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Long—Space for result
|-----------------|
|    convVerb     |   Word—Direction and type of conversion
|-----------------|
|     seconds     |   Long—Number of seconds since January 1, 1904
|-----------------|
|     datePtr     |   Long—Pointer to buffer for converted date
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|   secondsOut    |   Long—resulting number of seconds
|-----------------|
                   <—SP
```

Errors	$0390	badTimeVerb	Invalid convVerb value
	$0391	badTimeData	Invalid date or time to be converted

```
C           extern pascal unsigned long convSeconds(convVerb, seconds,
               datePtr);
            unsigned Word     convVerb;
            unsigned Long     seconds;
            Pointer           datePtr;
```

`convVerb` The type and direction for the conversion. Valid verbs are:

0	from seconds to the Miscellaneous Tools `ReadTimeHex` format
1	from the Miscellaneous Tools `ReadTimeHex` format to seconds
2	from seconds to ASCII text (`ReadAsciiTime` format)
3	not implemented
4	from seconds to ProDOS date/time format
5	from ProDOS date/time format to seconds
6	return the current time in seconds
7	set the current time from seconds
8	from ProDOS date/time format to the Miscellaneous Tools `ReadTimeHex` format
9	from the Miscellaneous Tools `ReadTimeHex` format to ProDOS date/time format
10	from seconds to HyperCard IIGS format
11	from HyperCard IIGS format to seconds

Note: In previous documentation (including the 5.0.3/4 release notes) the values of verbs 8 and 9 were interchanged. The values above are correct.

HyperCard IIGS format is the same as the Miscellaneous Tools ReadTimeHex format except that the bytes for the month and the day are one-based instead of zero-based.

`seconds` The input number of seconds since January 1, 1904 for all conversions that convert from a number of seconds to a different format, as well as for setting the current time. Conversions to a number of seconds since January 1, 1904 ignore this parameter, although it must be present.

`datePtr` Pointer to a buffer for all input and output values that are not a number of seconds since January 1, 1904. Conversions from a number of seconds will place the results in the buffer pointed to by datePtr; conversions to a number of seconds will get the source from a record pointed to by datePtr. When converting between two formats that are not seconds, the input pointed to by datePtr will be overwritten by the output.

Warning The buffer pointed to by datePtr must always be at least 8 bytes long and must be at least 40 bytes long when converting to ASCII format. ConvSeconds will overwrite the first eight bytes of the buffer pointed to by datePtr even if the input is less than eight bytes long.

`secondsOut` The output number of seconds since January 1, 1904 for all conversions that convert from any other format to a number of seconds. Conversions from a number of seconds since January 1, 1904 do not use this result space, although it must be present.

Note: In System 6.0, `ConvSeconds` treats ProDOS year numbers as documented in the ProDOS 8 Technical Notes. Year values 40..99 are 1940..1999, and years 0..39 are 2000..2039.

ScanDevices $3D03

ScanDevices provides easy access to a GS/OS system service vector which checks for disk insertions.

ScanDevices makes a device status call to each device with removable media except for AppleDisk 5.25 devices, which cannot be polled quickly. The devices are polled in ascending order. Every qualifying device is polled, and then ScanDevices returns the device number of the first device that reported an insertion.

If you want to insure that an insertion reported by ScanDevices is a recent insertion, first call ScanDevices and ignore the result. This forces the system to notice any old insertions on devices that have not been accessed recently.

As a side-effect, ScanDevices causes GS/OS to call Nofity Procs to inform them of any inserts or ejects that are noticed during the status calls.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Word—Space for result
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|  deviceNumber   |   Word—device number
|-----------------|
                   <—SP
```

Errors none

```
C           extern pascal Word ScanDevices();
```

deviceNumber is either zero (no disk inserted) or the GS/OS device number of the lowest-numbered device which reported an insertion.

ShowBootInfo $3C03

ShowBootInfo provides a way for special system extensions to make their presence known while the system is starting up. (For example, Control Panel 2.0 calls ShowBootInfo to display the icons of all Control Panels that receive control at boot time.)

You can provide ShowBootInfo with an icon, a text string, or both. The icon should be 20 pixels tall.

ShowBootInfo displays the icon along the bottom of the super-hires screen (each icon appears farther to the right), or it displays the text string on the text screen. (Normally only the super-hires screen is visible during boot; if the user presses a key at the beginning of the boot sequence, the text screen is visible instead.)

If the row of icons reaches the right edge of the screen, the whole row is erased to blue and the next icon appears at the bottom left.

ShowBootInfo takes no action if QuickDraw II is started. This way if setup file is installed at some time other than boot, its icon will not interfere with an application's use of the desktop.

Note: For users who don't like a cluttered boot screen, setting bit 1 of Battery-RAM location $5F prevents ShowBootInfo from displaying icons. (It still displays text strings.)

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|    cStringPtr   |   Long—pointer to zero-terminated string (or NIL)
|-----------------|
|     iconPtr     |   Long—pointer to DrawIcon-style icon (or NIL)
|-----------------|
                  <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                  <—SP
```

Errors none

```
C           extern pascal void ShowBootInfo(cStringPtr, iconPtr);
            Long  cStringPtr, iconPtr;
```

`cStringPtr` points to a C string typically giving the name and version number of a system extension. Pass NIL if you don't want a string displayed. `ShowBootInfo` automatically starts a new line after displaying your string, so the string should not have a return character on the end.

To make your string fit in with other strings, use twenty-two characters to describe the name of the product, followed by "vXX.XX" for the version. For example:

```
System Loader          v04.00
System Dispatch Table v04.00
```

`iconPtr` points to a 20-pixel-tall icon in the same format `DrawIcon` requires (see QuickDraw II Auxiliary). Pass NIL if you don't want an icon displayed.

If bit 31 of `iconPtr` is set, the icon overwrites the previously drawn icon.

StringToText $3B03

StringToText translates 8-bit-character text into similar text that can be displayed on the Apple IIGS text screen. You specify whether the resulting text can contain MouseText characters, or whether it must be plain ASCII.

You also specify whether the resulting text is allowed to be longer than the original text; this permits substitutions such as "(C)" for "©" and ">=" for "≥".

Eight slightly different display character sets are available; unless you specify otherwise, StringToText converts into the currently active character set.

Note: In the worst case, the output text is 4 times as long as the input text. This case is where the input text consists entirely of a series of "™" characters; the output is a series of "(TM)" sequences, four characters each.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Word—Space for result
|-----------------|
|      Space      |   Word—Space for result
|-----------------|
|      flags      |   Word—flags (see below)
|-----------------|
|     textPtr     |   Long—pointer to source text
|-----------------|
|     textLen     |   Word—length of source text
|-----------------|
|    resultPtr    |   Long—pointer to result buffer
|-----------------|
                  <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|   resultFlags   |   Word—result flags (see below)
|-----------------|
| printableLength |   Word—Number of printing characters in result
|-----------------|
                  <—SP
```

Errors $034F mtBufferTooSmall

```
C           extern pascal LongWord StringToText(flags, textPtr,
                  textLen, resultPtr);
            Word  flags, textLen;
```

```
            Ptr   textPtr, resultPtr;
```

`flags` bit 15 (`fAllowMouseText`): 1 = allow MouseText in result; 0 = no MouseText
bit 14 (`fAllowLongerSubs`): 1 = allow substituting several characters for one
bit 13 (`fForceLanguage`): 1 = force the result language in bits 2-0; 0=use current display language
bit 12 (`fPassThru`): 1 = pass untranslated high-ASCII characters straight through instead of omitting them
bits 11-3: reserved (use 0)
bits 2-0: result language. Specifies one of the eight character sets available in the Apple IIGS text mode.
0 = USA English
1 = U.K. English
2 = French
3 = Danish
4 = Spanish
5 = Italian
6 = German
7 = Swedish

`textPtr` Pointer to the input text. All 8 bits of the input characters are significant; the character set is the same as Shaston 8 (whch is also the same as the Macintosh standard character set).

`textLen` Number of characters in the input text.

`resultPtr` Pointer to result buffer. The buffer is a GS/OS result buffer. The first word is the buffer size, the second word is the data size, and the data begins at offset 4. You must provide the buffer size word; the value includes the 4 bytes used by the size words.

`printableLength` Number of printable characters in the result. If any MouseText is present in the result, `printableLength` will be smaller than the data size word in the result buffer.

`resultFlags` Bit 15 is set if the output text differs from the input text. Bits 14-0 are reserved and should be ignored.

Note: If you call `StringToText` from a high-level language, the result is a 32-bit integer. You can use `HiWord` to get the `resultFlags` and `LoWord` to get the `printableLength`.

List of Translations

$7F (DEL) becomes a MouseText checkerboard character if MouseText is allowed; it is removed otherwise.
Bullet (•) becomes an asterisk (*)

"®" becomes "(R)"
"©" becomes "(C)"
"™" becomes "(TM)"
Character $AB becomes an apostrophe.
"≠" becomes "<>"
"±" becomes "+-"
"≤" becomes "<="
"≥" becomes ">="
Character $B5 (micro) becomes "u"
"ƒ" becomes "f"
"«" becomes "<<"
"»" becomes ">>"
Character $CA (nonbreaking space) becomes a space ($20)
Character $D0 becomes a hyphen.
Character $D1 (dash, —) becomes two hyphens if allowed, one if not.
Typographical double quotes become boring double quotes.
Typographical single quotes become boring single quotes.
Characters $80 to $9F and $CB to $CD (accented letters) become similar unaccented letters.
Character $C0 (¿) is preserved if the display language is Spanish; otherwise it becomes a "?"
Character $C1 (¡) is preserved if the display language is Spanish; otherwise it becomes a "!"
Character $CE becomes "OE"
Character $CF becomes "oe"
Character $D8 becomes "y"

Character $C9 (…) a MouseText ellipsis character if MouseText is allowed; otherwise it becomes three periods
Character $11 (hollow apple) becomes a MouseText hollow apple or is removed
Character $12 (check) becomes a MouseText check mark or is removed
Character $13 (solid diamond) becomes a MouseText solid diamond or is removed
Character $14 (solid apple) becomes a MouseText solid apple or is removed
Character $D7 (hollow diamond) becomes a MouseText solid diamond or is removed (there is no MouseText hollow diamond available)

"#" is removed if the display language is U.K., French, Spanish, or Italian; otherwise it is unchanged.
"@" is removed if the display language is French, Spanish, Italian, or German. Otherwise it is unchanged.
"\" and "|" are removed if the display language is French, Danish, Spanish, Italian, German, or Swedish. For USA and U.K. they are unchanged.
"[", "]", "{", and "}" are changed into "(" and ")" if the display language is French, Danish, Spanish, Italian, German, or Swedish. For USA and U.K. they are unchanged.
"`" is changed to an apostrophe if the display language is Italian. Otherwise it is unchanged.
"~" is removed if the display language is French, Italian, or German; otherwise it is unchanged.
"£" is removed if the display language is USA, Danish, German, or Swedish. For U.K, French, Spanish, and Italian it is preserved.
"ç" is preserved if the display language is French, Spanish, or Italian. Otherwise it is changed into a "c"
"a" with a grave accent is preserved for French, Italian. Otherwise it becomes an "a".
"å" is preserved for Danish and Swedish. Otherwise it becomes an a.
"Å" is preserved for Danish and Swedish. Otherwise it becomes an A.
"°" is preserved for French, Spanish, and Italian. Otherwise it is removed.
"§" is preserved for French, Spanish, Italian, and German. Otherwise it is removed.
"ø" and "Ø" are preserved for Danish; otherwise they are changed to "0" (zero).
"Æ" and "æ" are preserved for Danish; otherwise they become "AE" and "ae"
"Ñ" and "ñ" are preserved for Spanish. Otherwise they become "N" and "n"
"é" is preserved for French, Italian; otherwise it becomes "e"
"grave u" is preserved for French and Italian; otherwise it becomes "u"
"grave e" is preserved for French and Italian; otherwise it becomes "e"
"¨" is preserved for French; otherwise it is removed
"grave o" is preserved for Italian; otherwise it becomes "o"

“grave i” is preserved for Italian; otherwise it becomes “i”
“Ä”, “Ö”, “ä”, and “ö” are preserved for German and Swedish; otherwise they become “A”, “O”, “a”, and “o”
“Ü” and “ü” are preserved for German; otherwise they become “U” and “u”
“ß” is preserved for German; otherwise it becomes “ss”

Character $A2 (cents) becomes “c”
Character $B4 becomes “Y”
Character $BB becomes “a”
Character $BC becomes “o”
Character $BD becomes “O”
Character $D9 becomes “Y”
Character $DA becomes “/”
Character $DB becomes “o”
Character $DC becomes “<“
Character $DD becomes “>“
Character $DE becomes “fi”
Character $DF becomes “fl”
Character $E1 becomes “.”
Character $E2 becomes “,”
Character $E3 becomes “„”
Character $E5 becomes “A”
Character $E6 becomes “E”
Character $E7 becomes “A”
Characters $E8 and $E9 become “E”
Characters $EA, $EB, $EC, and $ED become “I”
Characters $EE and $EF become “O”
Character $F0 becomes a MouseText solid apple, if allowed.
Character $F1 becomes “O”
Characters $F2, $F3, and $F4 become “U”
Character $F5 becomes “i”

SysBeep2 **$3803**

SysBeep2 takes an integer parameter indicating what sound to make. This is a clean hook for providing some useful Universal Access features, or just for sound addicts to have loads of fun with. SysBeep2 is built on top of SendRequest in the Tool Locator.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|    beepType     |   Word—beep type
|-----------------|
                  <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                  <—SP
```

```
C     extern pascal void SysBeep2(beepType);
      Word beepType;
```

beepType:

bit 15: 1=do nothing if no request procedure handles request (sbSilence)
0=Beep if no request procedure handles the request
bit 14: 1=make the sound at the next GetNextEvent call
0=do it now (sbDefer)
bits 13-0: identifies the specific reason for making a sound (see table)

Notes on deferred sounds

Bit 14 has no effect if the Event Manager is not started. If the Event Manager is started, the beep is deferred until the next GetNextEvent call that allows keyDown, autoKey, or mouseDown events.

When bit 14 is set and there is already a SysBeep2 call waiting to be dispatched at the next GetNextEvent call, the new call is ignored. It does not override the previous call, and it does not get queued up to play in sequence. It works this way on purpose. For example, if you are about to call AlertWindow but wish to override the SysBeep2 sound AlertWindow will automatically play, call SysBeep2 yourself first to schedule a deferred sound. The deferred SysBeep2 that AlertWindow executes has no effect.

SysBeep2 codes

All codes not listed are reserved for future definition by Apple.

Codes for which the Sound Control Panel "Give visual indication of sound" checkbox applies are marked with an asterisk (*).

*0000	sbAlertStage0	Alert stage 0 ($8000=default to silence)
*0001	sbAlertStage1	Alert stage 1
*0002	sbAlertStage2	Alert stage 2 (special: defaults to beeping twice)
*0003	sbAlertStage3	Alert stage 3 (special: defaults to beeping 3 times)
*0004	sbOutsideWindow	Can't click there (clicked outside a dialog/alert)
*0005	sbOperationComplete	Task Completed
*0006		reserved
*0007		reserved
*0008	sbBadKeypress	Bad keypress
*0009	sbBadInputValue	Bad input value
*000A	sbInputFieldFull	Input field full
*000B	sbOperationImpossible	Task impossible
*000C	sbOperationFailed	Task failed
*000D..000F		reserved
0010		reserved
0011	sbGSOStoP8	Switch from GS/OS to P8
0012	sbP8toGSOS	Switch from P8 back to GS/OS
0013	sbDiskInserted	Disk inserted
0014	sbDiskEjected	Disk ejected
0015	sbSystemShutdown	System shutdown
0016		reserved for Volume contents changed
0017..002F		reserved (for other NotifyProc events)
*0030	sbDiskRequest	Disk request (like AlertWindow with disk-swap icon)
0031	sbSystemStartup	System startup
0032	sSystemRestart	reserved for System restart (not used)
*0033	sbBadDisk	Bad disk
0034	sbKeyClick	reserved for Key click
0035	sbReturnKey	reserved for Return key
0036	sbSpaceKey	reserved for Space key
0040	sbWhooshOpen	"whoosh open" (called by WhooshRect)
0041	sbWhooshClosed	"whoosh closed" (called by WhooshRect)
0042	sbFillTrash	filling trash
0043	sbEmptyTrash	emptying trash
*0050	sbAlertWindow	Attention, need user response
*0051		reserved for AlertWindow
*0052	sbAlertStop	Stop (example: AlertWindow with Stop icon)

*0053	sbAlertNote	Note (example: AlertWindow with Note icon)
*0054	sbAlertCaution	Caution (example: AlertWindow with Caution icon)
*0055-59		reserved for AlertWindow
0060	sbScreenBlanking	screen is blanking
0061	sbScreenUnblanking	screen is unblanking
0100	sbYouHaveMail	You Have Mail
*0Exx	sbErrorWindowBase	called by ErrorWindow for errors 0..$FF
*0EFF	sbErrorWindowOther	called by ErrorWindow for errors $0100..$FFFF
*0Fxx		reserved for assignment where a visual indication of the sound is appropriate.

Note: The toolbox installs a GS/OS Notify Proc at system setup time. For certain events (Shutdown, Disk Eject, Disk Insert, Switch to P8, and Switch to GS/OS), this notify proc calls SysBeep2 with codes in the range $8010 to $802F.

Sound Request Procedures

SysBeep2 calls SendRequest in the Tool Locator with requestCode $0001 (systemSaysBeep). The low word of dataIn contains bits 0 to 13 of beepType.

SysBeep2 first calls SendRequest and directs the request **only** at the Sound Control Panel. If the request is rejected, SysBeep2 calls SendRequest again to broadcast the reqeust to everyone.

Bit 31 of dataIn means that a request procedure should accept or reject the request as usual, but no actual sound should be produced. This way, it's possible to send out a "feeler" to determine if a given sound request will be handled or not (especially whether it will be handled by a reqeust procedure earlier than your own).

VersionString $3903

VersionString converts a 32-bit Version number into a Pascal string up to nine characters long in the supplied ten-byte buffer.

See `rVersion` in the New Resource Types section for the definition of a long version number and for examples of the strings `VersionString` returns.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      flags      |   Word—reserved (pass 0)
|-----------------|
|   theVersion    |   Long—the version number to convert
|-----------------|
|    stringPtr    |   Long—pointer to 10-byte buffer
|-----------------|
                    <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                    <—SP
```

Errors none

```
C           extern pascal void VersionSTring(flags, theVersion,
            stringPtr);
            Word  flags;
            Long  theVersion;
            Ptr   stringPtr;
```

`flagWord` is reserved and should be zero.

`theVersion` is a standard version longword, as described in the New Resource Types section.

WaitUntil $3A03

WaitUntil provides an upper limit on the frequency of repeating actions so that improvements in system speed do not accidentally make things happen too fast for the user to see. On the other hand, calling WaitUntil does not significantly slow things down in cases where it is already slow enough.

WaitUntil deals with one-word time stamps in units of 1/960th of a second (one sixteenth of a tick).

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Word—Space for result
|-----------------|
|    delayFrom    |   Word—anchor point to delay from
|-----------------|
|   delayAmount   |   Word—specifies how long to wait from anchor
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|     newTime     |   Word—new anchor point for next call
|-----------------|
                   <—SP
```

Errors none

```
C           extern pascal Word WaitUntil(delayFrom, delayAmount);
            Word  delayFrom, delayAmount;
```

delayFrom specifies the time in the past from which to wait. For the first call in a series, use zero. For the other calls, use the newTime value returned by the previous call.

delayAmount specifies the minimum delay to enforce since the delayFrom time. If it has already been long enough, WaitUntil returns right away; otherwise it kills some time before returning. delayAmount is in units of one sixteenth of a tick. For example, pass $0040 for 4 ticks.

newTime This is a representation of the time the WaitUntil call completes. This is suitable for passing to a future WaitUntil call.

Notes

•If interrupts are disabled, `WaitUntil` may return immediately. For useful results, interrupts should be left enabled.

•The timing is only guaranteed to plus or minus one tick in System 6.0.

•Battery RAM location $60 can affect how long `WaitUntil` waits. If the value is $00 or $FF, there is no effect. For any other value, the `delayAmount` parameter is multiplied by one less than the value of the battery RAM location (up to a maximum of $F00 ticks). Note that a value of $01 multiplies the delay by zero, eliminating the delay.

•Scroll Bar controls automatically call `WaitUntil` when their value changes. `HiliteControl` also calls `WaitUntil`. See the Control Manager chapter.

026 Note Sequencer Update

No change for System 6.0.

025 Note Synthesizer Update

No change for System 6.0.

019 Print Manager Update

New Features of the Print Manager

When the user boots from an AppleShare file server, the Print Manager now puts the Printer.Setup file inside the user's network user folder, instead of trying to put it in `*:System:Drivers` (to which the user may not have access).

If the user has no network user folder, then the `*:System:Drivers` path is used just like before.

Several dialogs that were created using the Dialog Manager now use `AlertWindow`. (This saves disk space and RAM space.)

New Features of QuickDraw II

Added a `QDStartUp` `MasterSCB` bit (value $0100, bit 8) that causes the screen not to be cleared if it is already being displayed. `StartUpTools` uses this to avoid wiping the screen first to black and then to the desktop pattern, when the Window Manager is also being started up.

`QDStartUp` checks bit 2 ($0004) of Battery RAM location $5F. If set, it calls `SetIntUse(0)` so that mouse pointer tracking is not based on scanline interrupts. This gives better results with accelerators, or with the Video Overlay Card. (System 6.0 does not provide a user-visible way to change the setting of this bit.)

Fixed a problem where `QDStartUp` on ROM 1 with shadowing was not clearing the bank-one screen.

Fixed a problem where `QDStartUp` on ROM 3 was not returning the "QD already started" error when QuickDraw II was already started.

`QDShutDown` checks whether QuickDraw II Auxiliary is active and calls `QDAuxShutDown` if it is. This is needed because other tools (Window Manager and Standard File) now load and start QuickDraw II Auxiliary if it isn't started. `QDShutDown` also sets the color tables and SCBs to standard values.

`InflateTextBuffer` now returns errors properly if QuickDraw Auxiliary's text buffers could not be resized.

Animated Cursors with `SetCursor`

`SetCursor` now supports flicker-free cursors. For example, you can get a spinning beachball cursor effect, without any annoying flicker.

To switch from one cursor in a sequence to the next, call `SetCursor` as usual, except set bit 31 of the cursor pointer. This tells `SetCursor` not to undraw the old cursor before drawing the new one. All cursors in a sequence must have identical sizes, hot spots, and masks! If you set bit 31 while changing to a cursor with a different size, hot spot, or mask, you will get "cursor droppings" on the screen.

Warning: If you set bit 31, it is your responsibility to set the bit only when the current cursor has the same size, hot spot, and mask as the cursor you are setting. Be defensive. If there is any chance the cursor is not already set to the previous cursor in your animated sequence, use `GetCursorAdr` to check. If it doesn't match, do not set bit 31 on the next `SetCursor` call.

Get640Colors $DA04

Get640Colors returns a pointer to a 512-byte table of 32 $00s, 32 $11s, ..., 32 $FFs.

These tables can be used as "solid" pen patterns in either 320 or 640 mode. (In 640 mode they are actually dithered patterns, but they are the 16 different apparently-solid colors available.)

If you only wish to call SetPenPat on one of these patterns, use Set640Color.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Long—Space for result
|-----------------|
                    <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|     tablePtr    |   Long—pointer to pattern table
|-----------------|
                    <—SP
```

Errors none

```
C           extern pascal Ptr Get640Colors();
```

Set640Color $DB04

Set640Color sets the current grafport's pen pattern to a "solid" 640-mode dithered color (just like SetDithColor in some Pascal libraries). (In 640 mode they are actually dithered patterns, but they are the 16 different apparently-solid colors available.)

You can use Set640Color in 320 mode, too, but it would be just as easy to use SetSolidPenPat—so the call is named for the case where it's useful.

Note: The pen pattern affects all QuickDraw II drawing except for text. Use SetForeColor and SetBackColor to affect text drawing.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|    colorNum     |   Word—color number
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                   <—SP
```

Errors none

```
C           extern pascal void Set640Color(colorNum);
            Word  colorNum;
```

018 QuickDraw II Auxiliary Update

New Features of QuickDraw II Auxiliary

`DrawPicture`

FastPort features are now disabled during `DrawPicture` so that pen pattern changes (and other port parameters) work correctly when you start up QuickDraw with the `fastPort` bit set.

When `DrawPicture` encounters an invalid picture opcode, it now returns error $121F instead of crashing.

`DrawIcon` and `fastPort`

`DrawIcon` did not work well with `fastPort` mode on; now it does. (Certain calls, such as `InvertRect`, were accidentally restricted to drawing in the icon's rectangle if used immediately after a `DrawIcon` call.)

GetSysIcon $0F12

GetSysIcon returns small icons representing files, devices, and other miscellaneous icons. Some icons have separate 320- and 640-mode versions (GetSysIcon calls GetMasterSCB to decide which one to return).

The device icons are:

- 5.25" disk
- 3.5" disk
- Hard disk
- AppleShare server
- RAM disk
- CD-ROM disk
- Offline disk

The file icons are:

- Folder, open or closed (file type $000F)
- Application (file type $00B3 or $00FF)
- Stack (file type $0055)
- Document (any other file type)

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Long—space for result
|-----------------|
|      flags      |   Word—what kind of icon to get
|-----------------|
|      value      |   Word—file type, device ID, or index
|-----------------|
|    auxValue     |   Long—auxiliary type of file, or zero
|-----------------|
                  <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|     iconPtr     |   Long—pointer to resulting icon
|-----------------|
                  <—SP
```

Errors $1230 badGetSysIconInput

```
C           extern pascal IconPtr GetSysIcon(flags, value, auxValue);
            Word  flags, value;
            Long  auxValue;
```

`flags` bits 15-3: reserved (use 0)
bit 2: 0 = use closed folder icons
1 = use open folder icons
bits 1-0: type of icon to get
00 = file type icon (value = GS/OS file type)
01 = device icon (value = GS/OS device ID)
10 = miscellaneous icon (see table for values)
11 = illegal value

`value` File type, device ID, or other value, depending on flags bits 1-0.
Miscellaneous icons:
0=Desktop icon (used in Standard File)
1=padlock icon
2=up arrow icon
3=down arrow icon
4=boxed down arrow icon (used in Standard File)

`auxValue` When value is a GS/OS file type, `auxValue` is an auxiliary type.

`IBeamCursor` $1312

Sets the QuickDraw II cursor to an I-beam cursor. This is suitable when the cursor is positioned over an editable text field.

The cursor comes from a resource: `rCursor` ID $07FF0001 for 640-mode, $07FF0101 for 320-mode.

`DoModalWindow` uses `IBeamCursor` automatically.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors none

```
C           extern pascal void IBeamCursor();
```

PixelMap2Rgn $1012

Transforms a pixel map into a QuickDraw II region. The points to be included in the region are specified by color.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Long—Space for result
|-----------------|
|    srcLocInfo   |   Long—Pointer to a locInfo record for the pixel
map
|-----------------|
|   bitsPerPixel  |   Word—Number of bits per pixel (either 2 or 4)
|-----------------|
| colorsToInclude |   Word—Bit flags indiciating which colors to
include
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|     theRgn      |   Long—Handle to region created from the pixel map
|-----------------|
                   <—SP
```

Errors $0433 `rgnFull` region is larger than 64K
Memory Manager Errors returned unchanged

```
C           extern pascal RgnHandle PixelMap2Rgn(srcLocInfo,
                  bitsPerPixel, colorsToInclude);
            LocInfoPtr  srcLocInfo;
            Integer     bitsPerPixel;
            Word        colorsToInclude;
```

`srcLocInfo` A pointer to a QuickDraw II `locInfo` structure that contains the source pixel map. `PixelMap2Rgn` requires a `locInfo` structure to determine the size of the pixel map.

`bitsPerPixel` The number of bits per pixel. Normally 4 for 320 mode or 2 for 640 mode; you can also specify 4 in 640 mode so that `PixelMap2Rgn` will treat two dithered pixels as one 16-color pixel. Values other than 2 or 4 are not supported.

`colorsToInclude` A word of bits flags, each indicating whether or not a given color pixel should be included in the resulting region or not. Bit 0 is color 0, etc. Only bits 0..3 are valid if `bitsPerPixel` is 2.

`theRgn` The QuickDraw II region constructed from the pixel map.

Discussion

QuickDraw II supports extensive graphics operations for regions. While a pixel map is a collection of pixels of any color, a region is a collection of points. Any given point is simply in the region or not in the region. Regions, being collections of points, have no intrinsic color. However, regions are more interesting objects and can be manipulated in ways not possible for pixel maps.

`PixelMap2Rgn` lets you create regions from any pixel map.

One application of `PixelMap2Rgn` is for a lasso tool in a graphics application, where the user draws around a graphic object and the lasso shrinks to exactly grab the object it surrounds. The application can use `CalcMask` (in QuickDraw II Auxiliary) to transform the source pixel map into a mask, where the selected portion is white and the unselected area is black. `PixelMap2Rgn` can then transform the white part of the mask into a region containing the lassoed pixels. The application can then perform any operation on the region, including inversion, framing, and filling. It can also use a slightly inset copy of the region (`InsetRgn`) subtracted from the original region (`DiffRgn`) as a thin border for a "shimmer" effect indicating the region selected.

Notes

All the coordinates of the `srcLocInfo` `boundsRect` must be nonnegative. `PixelMap2Rgn` operates on an entire pixel map, and is intended to be used with offscreen pixel maps. It is not usually useful to pass a window pointer as the `srcLocInfo`.

`PixelMap2Rgn` is stored in a dynamic segment, so the boot disk may be needed on the first call.

WhooshRect **$1412**

WhooshRect animates a "zooming" effect from one rectangle to another, as the Finder does when you open an icon. Before the visual effect, WhooshRect calls SysBeep2 to allow for a corresponding audio effect.

Note: smallRect doesn't actually have to be smaller than bigRect.

For best results, all four smallRect coordinates should be different from the corresponding bigRect coordinates. WhooshRect draws using an exclusive-or pen mode, and at times more than one intermediate rectangle is on the screen. If edges of the intermediate rectangles overlap, they cancel each other out. It never leaves garbage on the screen, but the effect of the animation is lost.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      flags      |   Long—flags (see below)
|-----------------|
|    smallRect    |   Long—pointer to first rectangle
|-----------------|
|     bigRect     |   Long—pointer to second rectangle
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                   <—SP
```

Errors none

```
C           extern pascal void WhooshRect(flags, smallRect, bigRect);
            Long  flags;
            Rect  *smallRect, *bigRect;
```

`flags` bit 31: 1 = zoom out, from small rectangle to large rectangle
0 = zoom in, from large rectangle to small rectangle
bit 30: 1 = use local coordinates in the current port
0 = use global coordinates
bit 29: 1 = skip the normal `SysBeep2` call
0 = call `SysBeep2`($8040) for zoom-open,
$8041 for zoom-closed
bits 28-0: reserved (use 0)

`smallRect` Pointer to the first rectangle. If this is NIL, `WhooshRect` returns without doing anything.

`bigRect` Pointer to the second rectangle. If this is NIL, `WhooshRect` returns without doing anything.

Clipping

If flag bit 30 is clear (global coordinates), rectangles are drawn in a port owned by the system, and very little clipping is done.

If flag bit 30 is set (local coordinates), rectangles are drawn in whatever port the caller has set, so the `visRgn` and `clipRgn` of that port are used.

030 Resource Manager Update

New Features of the Resource Manager

•The resource manager now protects all open resource files from being accidentally closed by applications. (In System 5.0.4 and earlier, only the Sys.Resources file was protected.)

•`ResourceStartUp` now returns error $1E12, `resDupStartUp`, if the Resource Manager has already been started up for the specified memory ID.

•Added new calls to support named resources: `RMFindNamedResource`, `RMGetResourceName`, `RMLoadNamedResource`, `RMSetResourceName`.

•`AddResource`, `SetResourceID`, and `RMSetResourceName` return error $1E13, `resInvalidTypeOrID`, if the specified resource type or resource ID is zero.

•The new call `LoadResource2` loads a resource and provides information on the previous state of that resource.

•`LoadResource` and `LoadResource2` both re-lock the handle being returned if the resource attributes say the handle was originally locked.

•It is now possible to have preload resources in Sys.Resources and actually have them preload.

•If the resource map for Sys.Resources, on disk, has a nonzero value in the `mapNext` field, previous versions of the Resource Manager would crash. A nonzero value is now tolerated.

•Cancelling out of a `LoadResource` for a locked resource now works correctly. It used to return garbage the next time you loaded that resource.

•`ResourceShutDown` refuses to shut down user $401E, which is the Resource Manager itself. This search path must always remain available for the system to work properly, so now this is enforced. `ResourceShutDown` returns error $1E0F, `resInvalidShutDown`, if $401E is the current resource application.

•`OpenResourceFile` has a flag bit to override the automatic loading of preload resources—set bit 15 of the `openAccess` parameter. (For example, the Finder overrides preloading when it opens a file's resource fork to get its `rComment`(1) and `rVersion`(1) resources.)

•The Resource Manager no longer reports an error when operating on an empty resource. (It used to get error $0053 because it was reading or writing zero bytes to address zero; GS/OS allows zero-byte-long reads and writes, but it does not allow address zero.)

•GetOpenFileRefNum inputs $0000 and $FFFF now work as documented. They were not previously implemented.

•CreateResourceFile on a file that already exists now ignores the access, filetype, and auxiliary type parameters as documented. Sufficiently strange values used to cause an error.

•If CloseResourceFile returns an error (such as $002B, disk write protected), there was previously no way to close the file. Now you can set bit 15 of the CloseResourceFile parameter to tell the Resource Manager to close the file even if it can't write out any changed resources or the up-to-date resource map. Use this option carefully to avoid leaving a resource fork in an inconsistent state.

•When looking for a free location in the resource fork, the Resource Manager assumes the fork's free list is valid. If the large free area at the end is missing, fatal error $1E42 now occurs (previously a resource would be placed at offset zero in the fork).

•UniqueResourceID can no longer return out-of-range ID values for range $FFFF.

•MatchResourceHandle now optionally returns the resource file ID of the file the handle belongs to. (This is important because calling HomeResourceFile to locate the resource finds the first accessible resource of the specified type and ID, which may not be the one you wanted.) To ask for the value, set bit 31 of the foundRec pointer; the foundRec is then defined as follows:

```
      |-----------------|
+000  |  resourceType   |   Word—Type of resource
      |-----------------|
+002  |   resourceID    |   Long—ID of resource
      |-----------------|
+006  |     fileID      |   Word—ID of file where resource was found
      |-----------------|
```

Named Resources

Four new calls support named resources, using the `rResName` resource format defined in Toolbox Reference, Volume 3.

The calls are `RMFindNamedResource`, `RMGetResourceName`, `RMLoadNamedResource`, and `RMSetResourceName`. (The `RM` prefix distinguishes these Resource Manager calls from the similar HyperCard IIGS callbacks.)

Case Sensitivity

Resource names **ARE** case sensitive. "Splat" and "SPLAT" are two distinct names.

Names are not directly tied to resources

When working with named resources, keep in mind that a resource name is associated with a particular resource type and ID (within a resource file).

A resource name is not directly associated with the resource, so operations like `RemoveResource` and `SetResourceID` can easily leave a "dangling" name, or dissociate a resource from a name.

Resource names are a convenience, but a resource name is not a property of a particular resource.

LoadResource2 $291E

LoadResource2 is like LoadResource, except that it returns information about the previous state of the returned handle.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Long—Space for result
|-----------------|
|      flags      |   Word—flags
|-----------------|
|    bufferPtr    |   Long—Pointer to result buffer
|-----------------|
|  resourceType   |   Word—Type of resource to find
|-----------------|
|   resourceID    |   Long—ID of resource to find
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|  resourceHandle |   Long—Handle of resource in memory
|-----------------|
                   <—SP
```

Errors	$1E03	resNoConverter	No converter routine found for resource type.
	$1E06	resNotFound	Specified resource not found.
	GS/OS errors	Returned unchanged.	
	Memory Manager errors		Returned unchanged.

```
C           extern pascal Handle LoadResource2(flags, bufferPtr,
                 resourceType, resourceID);
            Word  flags;
            Ptr   bufferPtr;
            Word  resourceType;
            Long  resourceID;
```

flags — reserved, use 0

bufferPtr — Pointer to a one-word result buffer which receives the previous attributes word of the handle, or $FFFF if the handle did not previously exist.

`resourceType` Resource type of resource to be loaded.

`resourceID` Resource ID of resource to be loaded.

RMFindNamedResource $2A1E

RMFindNamedResource takes a resource type and a resource name and finds the resource ID of the corresponding resource. The current resource file and search depth are respected.

If you simply want to load the resource, use RMLoadNamedResource. Since the resource found may not be the topmost resource with the returned ID, it is simpler to let RMLoadNamedResource load the resource from the proper file than to manipulate the current resource file setting yourself.

Resource names are stored in rResName resoures, as described in Toolbox Reference 3, Appendix E.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Long—Space for result
|-----------------|
|      rType      |   Word—resource type of resource to find
|-----------------|
|     namePtr     |   Long—pointer to Pascal string name of resource
|-----------------|
|    fileNumPtr   |   Long—pointer to Word to receive resource file ID
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|   resourceID    |   Long—ID of resource found
|-----------------|
                   <—SP
```

Errors $1E10 resNameNotFound
$1E11 resBadNameVers

```
C           extern pascal Long RMFindNamedResource(rType, namePtr,
                  fileNumPtr);
            Word  rType;
            Ptr   namePtr;
            Word  *fileNumPtr;
```

RMGetResourceName $2B1E

Returns the Pascal string name of the specified resource.

Resource names are stored in `rResName` resoures, as described in Toolbox Reference 3, Appendix E.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      rType      |   Word—resource type of resource to find
|-----------------|
|       rID       |   Long—resource ID of resource to find
|-----------------|
|     namePtr     |   Long—pointer to buffer to receive name
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                   <—SP
```

Errors $1E10 `resNameNotFound`
$1E11 `resBadNameVers`

```
C           extern pascal Long RMGetResourceName(rType, rID, namePtr);
            Word  rType;
            Long  rID;
            Ptr   namePtr;
```

RMLoadNamedResource $2C1E

RMLoadNamedResource takes a resource type and a resource name and loads the corresponding resource. The current resource file and search depth are respected.

Resource names are stored in rResName resoures, as described in Toolbox Reference 3, Appendix E.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Long—Space for result
|-----------------|
|      rType      |   Word—resource type of resource to load
|-----------------|
|     namePtr     |   Long—pointer to Pascal string name of resource
|-----------------|
                    <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|    resHandle    |   Long—handle to the loaded resource
|-----------------|
                    <—SP
```

Errors $1E10 resNameNotFound
$1E11 resBadNameVers
Memory Manager errors returned unchanged
GS/OS errors returned unchanged

```
C           extern pascal Handle RMLoadNamedResource(rType, namePtr);
            Word  rType;
            Ptr   namePtr;
```

RMSetResourceName $2D1E

Sets the name of the specified resource, first removing any existing name. The current resource file and depth are respected.

Resource names are stored in `rResName` resoures, as described in Toolbox Reference 3, Appendix E.

If `namePtr` points to a zero-length string, the resource's name is removed. If an `rResName` resource becomes empty, `RMSetResourceName` removes the `rResName` resource.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      rType      |   Word—resource type of resource to name
|-----------------|
|       rID       |   Long—resource ID of resource to name
|-----------------|
|     namePtr     |   Long—pointer to name (Pascal string)
|-----------------|
                    <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                    <—SP
```

Errors $1E10 `resNameNotFound`
$1E11 `resBadNameVers`
$1E13 `resInvalidTypeOrID`

C
```
extern pascal void RMSetResourceName(rType, rID, namePtr);
Word  rType;
Long  rID;
Ptr   namePtr;
```

010 SANE Update

For ROM 1, `SANEVersion` now returns version 3.0 for consistency with ROM 3.
There are no other changes.

007 Scheduler Update

New Features of the Scheduler

The system now clears the scheduler's private "don't-dispatch" flag once at boot time, just like `SchBootInit` does on ROM 3.

This fixes the problem where crashing in a scheduled task on ROM 1 would cause the scheduler to stop dispatching tasks until a power-down or a self-test (just rebooting was not sufficient).

022 Scrap Manager Update

New Features of the Scrap Manager

The Scrap Manager now permits individual scraps to exceed 64K. Previously, you could create scraps this large, but the `UnloadScrap` would not write them to disk properly.

Changed `ScrapStartUp` to zero out the Scrap State, because on ROM 3 `ScrapBootInit` gets called before the GS/OS is present, and the Scrap Manager was deciding that the scrap had already been read into memory.

The Scrap Manager no longer accidentally does a Close on reference number zero (possibly closing files it did not open). Previously, this would happen when the Scrap Manager tried failed to load the scrap from disk (because the Clipboard file was not present).

The Clipboard file is now created with GS/OS file type $F9 (System file).

The new call `GetIndScrap` allows utilities to work with all existing scraps instead of assuming that only previously-known scrap types are present.

GetIndScrap $1416

This call is useful for utilities that want to read all scrap types (Scrapbook-type desk accessories need this, or they can only save scrap types they know about).

To get information on all scraps present, call `GetIndScrap` repeatedly with index equal to 1, 2, etc., until `GetIndScrap` returns an error.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      index      |   Word—Which scrap to get
|-----------------|
|    bufferPtr    |   Long—Pointer to buffer to receive information
|-----------------|
                    <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                    <—SP
```

Errors $1610 `badScrapType` Bad value for index

```
C           extern pascal void GetIndScrap(index, buffer);
            Word  index;
            Ptr   buffer;
```

`index` Specifies which scrap to get (1 for the first scrap, 2 for the second, and so on).

`buffer` Pointer to a 10-byte result buffer with the following format:

+000 `scrapType` (word)
+002 `scrapSize` (long)
+006 `scrapHandle` (long)

008 Sound Tools Update

`SoundVersion` now returns $0303 to match ROM 3.

023 Standard File Update

New Features of Standard File

•When you insert a disk that is unformatted or contains an unrecognized file system, or which has the same name as an online volume with files open, Standard File calls `HandleDiskInsert` (see the Window Manager chapter) to let you rename, initialize, erase, or eject the disk.

•Standard File now uses `ListKey` in the `SFGetFile` dialogs to handle jumping around in the file list (see the List Manager section for more information on `ListKey`). This means that you can select files by typing as many characters from the beginning of a filename as you wish. (Previously, typing a letter would always jump to the first filename starting with that letter.)

•In `PutFile` dialogs, the file list is hilited with a bold outline when it is receiving keystrokes. Command-Tab alternates between the List and the edit-line being the target. Clicking in either the List or the edit-line makes that item the target.

•If you click New Folder or Save and the name in the edit-line field is not valid for the target file system, an alert appears suggesting a valid alternative name. You have the choice of sticking with your old name or accepting the new one. In either case, you may edit the name further, or change to another disk or directory, before retrying the Save or New Folder operation.

•There's a pop-up path menu now. Only the last section of the pathname is displayed; the others are visible when the menu is popped up. The padlock icon comes after the pathname segment.

•Filter procedures now have access to all `GetDirEntry` parameters, including the option list.

•Standard File now allows 128 volumes to be online. Previously, it was limited to 20.

•Added new key equivalents for the Volumes button (Command-D and Command-ESC).

•Standard File's icons come from `GetSysIcon` (see QuickDraw II Auxiliary).

•Long filenames are drawn narrower to allow more of the characters to be seen (uses `SetCharExtra` to put one less pixel than usual between characters).

•You can no longer type colons (:) into the file name field in `PutFile` dialogs.

•`SFStartUp` returns error $17FF, `sfNotStarted`, if you pass zero for the work area pointer.

•All Standard File calls other than the housekeeping calls return error $17FF, `sfNotStarted`, if Standard File has not been started.

•Fixed a problem where `SFMultiGet2` would occasionally return error $1705 for no good reason.

•Fixed a problem that would occasionally cause a filename to be duplicated many times in the file list.

034 Text Edit Update

New Features of Text Edit

Note: In 5.0.3 & later versions of `TEGetText`, bit 5 ($20) of the `bufferDesc` parameter is the `onlyGetSelection` bit. This works for all data formats except `LETextBox2` format (because that would require special handling of the style information).

•Corrected a `TEScroll` anomaly where scrolling to a specified character posistion would sometimes scroll to the very end of the text. The problem only appeared when the text was longer than 64K.

•Printable control characters (like the Apple symbols in Shaston) cause fewer problems with word wrapping than they used to.

•Text Edit controls no longer eat Command- key presses they do not use.

012 Text Tools Update

The Text Tools version number is now 3.1 ($0301).

The calls `GetInputDevice`, `GetOutputDevice`, and `GetErrorDevice` no longer leave garbage in the high byte of the `deviceType` result parameter.

001 Tool Locator Update

New Features of the Tool Locator

Inter-process communication

Two new calls support communication among any piece of code that wants to participate, including applications, desk accessories, and the system.

`StartUpTools`/`ShutDownTools` enhancements

`StartUpTools` now knows how to load and start Media Control and MIDI Synth tool sets, which are part of System 6.0, as well as Tool037 (which is not part of System 6.0). You can include these tool sets in your `StartStopToolsRec` now.

`StartUpTools` goes to some trouble for your application to make tool startup visibly smooth. If you're starting QuickDraw and the Window Manager, it asks QuickDraw II not to clear the screen if it's already on (`RefreshDesktop` eventually wipes over the old screen).

Instead of building the path "1/"+GET_NAME, `StartUpTools` now uses `LGetPathname2` on the caller's memory ID to locate the correct resource fork. (This makes `StartUpTools` work for applications with "/" in their name.)

`StartUpTools` `startStopRefDesc` bit 3 ($0008) means "open my resource fork as-allowed instead of read-only."

`StartUptools` or `ShutDownTools` `startStopRefDesc` bit 4 ($0010) means skip starting up the Resource Manager. If you have already started the Resource Manager, you must set this bit! This way `StartUpTools` does not attempt a duplicate `ResourceStartUp` call, and `ShutDownTools` does not attempt a premature `ResourceShutDown`. `StartUpTools` also doesn't try to pre-allocate the Super Hi-res screen memory if you set this bit.

`ShutDownTools` always asks QuickDraw to leave the Super Hi-res screen turned on if `ShutDownTools` determines that a Quit call will be handled by GS/OS rather than by a shell program (the test is whether GS/OS's idea of the current application memory ID matches the QuickDraw memory ID). GS/OS handles making the text screen visible if necessary.

`ShutDownTools` `startStopRefDesc` bit 2 ($0004) means "leave the Super Hi-res screen turned on, and don't mess with it." If you leave this bit clear, `ShutDownTools` erases the menu bar to a white rectangle before shutting down, so that the previous application's menus are not visible while the next application is starting up.

Tool Set Versions

Changed the way StartUpTools, LoadTools, and LoadOneTool examine bits 14 through 12 of version words. If one of these bits is off in the requested version, it is ignored in the actual version.

SaveTextState and RestoreTextState

SaveTextState now preserves and enables text page one shadowing, and RestoreTextState restores the status of text page one shadowing.

RestoreTextState(NIL) takes no action (in case SaveTextState returned NIL because it could not allocate memory).

TLMountVolume

TLMountVolume has always required QuickDraw and Event Manager to be started.

UnloadOneTool

Changed UnloadOneTool to return no error when unloading a tool that already wasn't there, but which had an entry in the default TPT. It was usually returning error $00FE before.

Message Type $0011, pathnames to open or print

There is a new message type, $0011, for passing the GS/OS-string pathnames of files to open or print (any number of pathnames is allowed, including zero).

Finder 6.0 uses this in addition to message type $0001 (Pascal string pathnames of files to open or print). The format of message $0011 is as follows:

```
+000    LONG used by the system
+004    WORD message type ($0011)
+006    WORD printFlag ($0000 = Open, $0001 = Print)
+008    C1String   pathname of first file (class-one GS/OS input string)
+xxx    C1String   pathname of second file
       ...
+yyy    WORD $0000 terminates list (a null-string pathname)
```

Note: When MessageCenter deletes message $0001, it automatically deletes message $0011, too.

New Tool Locator Calls

AcceptRequests $1B01

Provide straightforward inter-process communication. The system keeps a list of procedure pointers associated with all the processes that can receive requests.

To notify the system that you can receive requests, call `AcceptRequests`. When you can no longer accept requests, call `AcceptRequests`, with `requestProc`=NIL and the same `nameString` and `userID` you passed the first time.

Another way to remove a request proc is to pass `nameString`=NIL, `requestProc`=NIL, specifying only a `userID`. This removes all request the procedures with the specified `userID`.

Warning: If you neglect to remove your request procedure before your application quits, or before your code is otherwise left dangling, the system will crash.

Only one `AcceptRequests` call will succeed for each `nameString`. If it is useful for more than one copy of your program to exist in the system, generate a unique NameString at run-time by concatenating an ASCII representation of your user ID to the end of your string. (You can also have multiple request procedures in your product, as long as they have separate names.)

Since requests can be sent to a select group of request procs based on prefix strings of the name strings, choose your strings carefully. The recommended format for nameString is "`YourCompany~YourProduct~`". This is to support application-specific request types (in the range $8000..FFFF) that apply to request procedures that match a given prefix string (see `SendRequest`).

You may wish to include your product's version number at the end of your request procedure's name ("`YourCompany~YourProduct~v1.2~`"). You will normally not want to include the version number when sending requests to your named procedures.

In specialized cases, like supporting fourth-party modules for your applications, you will need a different convention, like "YourProductName~FourthPartyName~ TheirProductName~". The important point is that request codes $8000 and up are defined within prefix-string domains.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|   nameString    |   Long—pointer to Pascal name string
|-----------------|
|     userID      |   Word—user ID associated with this request proc
|-----------------|
|   requestProc   |   Long—address of the request procedure to install
|-----------------|
                  <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                  <—SP
```

`nameString`: `AcceptRequests` makes its own copy of the string; your copy does not have to stay around after the call completes.

Errors:

$0113	`srqNameTooLong`	name must be 62 characters or less
$0121	`srqDuplicateName`	

```
C           extern pascal void AcceptRequests(nameString, userID,
                        requestProc)
            Pointer nameString;
            Word userID;
            WordProcPtr requestProc;
```

For debugging purposes only: The list of current request procedures is currently kept in the Message Center, one named message per request proc. Each message has a $8000+ Type and has this form:

STRING	$FF “>“ Company~Product~ (The string has a length byte)
WORD	UserID
LONG	RequestProcPtr

Parameters to a Request Procedure

Parameters

Stack when request procedure gets control

```
|Previous contents|
|-----------------|
|      space      |   Word—space for result
|-----------------|
|     reqCode     |   Word—request code
|-----------------|
|     dataIn      |   Long—input data or pointer to input data
|-----------------|
|     dataOut     |   Long—pointer to output buffer
|-----------------|
|       RTL       |   3 bytes—return address
|-----------------|
                   <—SP
```

Stack just before the request procedure executes an RTL

```
|Previous contents|
|-----------------|
|     result      |   Word—bit 15 set if the request was accepted
|-----------------|
|      RTL        |
|-----------------|
                   <-SP
```

`result`:
Bit 15 = handled the request
Bits 14-0 = reserved (use 0)

The result space is pre-initialized to zero, so the request procedure doesn't have to store anything there to reject the request.

Bank and Direct Page registers

The Bank and Direct Page registers are undefined on entry to a request procedure. If you normally use the Bank register to access your global variables, you must save, set, and restore it, or access your globals with long addressing.

To help track down erroneous request procedures, the Bank and Direct Page registers are worse than undefined. Bank is $FE (ROM), and Direct Page is $CCCC (in peripheral-card I/O space). A request procedure accidentally using these values will almost certainly fail dramatically.

`SendRequest` Request codes

Codes $0000..7FFF have global meaning and are defined by Apple.
Codes $8000..FFFF are defined separately for specific prefix strings. Invent an appropriate prefix string for your product or family of products and you can assign your own codes in this range.

$0001 = `systemSaysBeep` (used by `SysBeep2` in the Miscellaneous Tools)
$0002 = `systemSaysUnknownDisk` (see Window Manager, under `HandleDiskInsert`)
$0003 = `srqGoAway` (see below)
$0004 = `srqGetrSoundSample` (see Sound Control Panel documentation)
$0005 = `srqSynchronize` (wait for asynchronous operations to complete, such as sounds played with `srqPlayrSoundSample`; `dataIn` and `dataOut` are reserved and should be zero)
$0006 = `srqPlayrSoundSample` (see Sound Control Panel documentation; `dataIn` is a `rSoundSample` handle or a special value)
$0008 = `systemSaysNewDeskMsg` (see the Window Manager chapter)
$000E = `systemSaysEjectingDev` (sent by `HandleDiskInsert`; low word of `dataIn` = device number)
$0502 = `systemSaysDeskStartUp`—at `DeskStartUp` time (`dataIn` and `dataOut` are reserved)
$0503 = `systemSaysDeskShutDown`—at `DeskShutDown` time (`dataIn` and `dataOut` are reserved)
$051E = `systemSaysFixedAppleMenu`—after `FixAppleMenu` adds items to the Apple menu
$0F01 = `systemSaysMenuKey`—`MenuKey` got a key it didn't find a match for (see Menu Manager)
$01xx = Reserved for `finderSaysXXX`; see Finder 6.0 documentation.

The `srqGoAway` request

If you receive the `srqGoAway` code, someone is asking for permission to call `UserShutDown` on you, to remove you from memory. You are not required to accept this request.

If you do accept an `srqGoAway` request, you must fill in the `resultID` field of `dataOut` either with zero (indicating it is not okay to remove you from memory) or with your `userID`. `dataIn` is reserved (should be passed zero and ignored by accepting procedure until some bits are defined). The `dataOut` buffer has the following format:

+000: `recvCount`	word filled in by `SendRequest`
+002: `resultID`	word filled in by the accepting procedure (memory ID to be used with `AcceptRequests` and `UserShutDown`, or zero if the accepting procedure refuses to go away)
+004: `resultFlags`	word filled in by accepting procedure

bit 15: 1=OK to shut down as Restartable, 0=not OK
bits 14-0: reserved, return 0

Sample request procedure skeleton in assembly

```
OldDPage        equ 1
RTL1            equ 3
DataOut         equ 6
DataIn          equ 10
Request         equ 14
Result          equ 16

SampleReqProc   phd
                tsc
                tcd

                lda <Request
                cmp #myRequestType
                bne @exit

                ...
                lda #$8000
                sta <Result

@exit           pld
                lda 2,s
                sta 12,s
                lda 1,s
                sta 11,s
                ply
                ply
                ply
                ply
                ply
                rtl
```

Installing a request procedure from assembly:

```
                pea ^myNameString
                pea myNameString
                pha
                _MMStartUp      ;get my memory ID
                pea ^mySampleReqProc
                pea mySampleReqProc
                _AcceptRequests
                ...

myNameString    dc.b 24,'CompanyName~ProductName~'
```

Removing a request procedure from assembly:

```
                pea ^myNameString
                pea myNameString
                pha
                _MMStartUp      ;get my memory ID
                pea 0
                pea 0
                _AcceptRequests
                ...
```

Sample request procedure in C

```
pascal unsigned myRequestProc(request, dataIn, dataOut)
       unsigned request;
       long dataIn;
       long dataOut;
{
```

```
        unsigned oldB = SaveDB();  /* may be needed for global var access
    */
        unsigned result = 0;

        switch(request)
          {
               case finderSaysHello:
                     result = handleHello();
                     break;

               case finderSaysGoodbye:
                     result = handleGoodbye();
                     break;

               case finderSaysExtrasChosen:
                     result = handleExtrasChosen( (unsigned) dataIn );
                     break;
          }

        RestoreDB(oldB);
        return( result ? 0x8000:0 );
    }
```

Installing a request procedure from C:

```
    AcceptRequests("\pCompanyName~ProductName~", MMStartUp(), myRequestProc);
```

Removing a request procedure from C:

```
    AcceptRequests("\pCompanyName~ProductName~", MMStartUp(), 0L);
```

Sample request procedure in Pascal

```
{ use compiler option to force long global addressing, if appropriate }
function myRequestProc(request: integer; dataIn: univ longint;
                       dataOut: univ longint): integer;
var
    result: integer;
begin
    result := 0;
    case request of
        finderSaysHello:
            result := handleHello;
        finderSaysGoodbye:
            result := handleGoodbye;
        finderSaysExtrasChosen:
            result := handleExtrasChosen(LoWord(dataIn));
    end; { case }
    if result <> 0 then result := $8000;
    myRequestProc := result;
end;
```

Installing a request procedure from Pascal:

```
    AcceptRequests('CompanyName~ProductName~', MMStartUp, @myRequestProc);
```

Removing a request procedure from Pascal:

```
    AcceptRequests('CompanyName~ProductName~', MMStartUp, NIL);
```

GetMsgHandle $1A01

Returns the handle to a messsage in the Message Center. The returned handle is NIL if there's an error. Otherwise it's a handle to the system's copy of the message.

An application should not modify the message handle and should not assume that it's locked (it isn't), but it can look in there to find the data and the message type and contents (the Type word is at offset +4 in the block). If the type is $8000 or above, the message begins with a Pascal String name.

Note: To delete a message if you don't know the message type, use `GetMsgHandle`, look at offset 4 in the block to get the message type, and feed that to `MessageCenter`(3,type,NIL).

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Long—Space for result handle
|-----------------|
|      flags      |   Word—flags
|-----------------|
|   messageRef    |   Long—specifies which message to fetch
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|    theHandle    |   Long—handle of the specified message
|-----------------|
                   <—SP
```

Errors $0111 messNotFound

```
C           extern pascal Handle GetMsgHandle(flags,messageRef);
            Word  flags;
            Long  messageRef;
```

`flags`:

Bits 15-2 are reserved.

Bits 1 and 0 specify the type of reference in messageRef:

00 = get the `messageRef`-th message, counting from 1 = first message
01 = get the message with Type matching the low word of `messageRef`
10 = get the message with name `messageRef` points to (Pascal string)
11 = reserved

SendRequest **$1C01**

Sends the specified request to zero or more qualifying request procedures that have been registered with `AcceptRequests`. (See discussion under `AcceptRequests`.)

This is a synchronous operation; `SendRequest` returns after calling zero or more request procedures.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|     reqCode     |   Word—request code
|-----------------|
|     sendHow     |   Word—specifies how to send the request
|-----------------|
|     target      |   Long—specifies recipient of request
|-----------------|
|     dataIn      |   Long—input data or pointer to input data
|-----------------|
|     dataOut     |   Long—pointer to output buffer
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                   <—SP
```

Errors	$0120 `reqNotAccepted`	nobody accepted the request
	$0122 `invalidSendRequest`	bad combination of `reqCode` & `target`

```
C           extern pascal void SendRequest(reqCode, sendHow, target,
              dataIn, dataOut);
            Word  reqCode, sendHow;
            Long  target, dataIn;
            Ptr dataOut;
```

`reqCode`: See table under `AcceptRequests`. The request code is passed to all the selected request procedures so they can decide what, if anything, to do with the input and output data buffers.

The `sendHow` parameter:

Bit 15: 1 = Stop after first acceptance (`stopAfterOne`).
0 = Send to any number of request procedures
Bits 14-2: reserved (use 0).
Bits 1-0:
00 = send to all ReqProcs, `sendToAll` (target is reserved and should be zero)

01 = select ReqProcs by prefix of name, `sendToName` (target points to a Pascal string)
10 = select a ReqProc by User ID, `sendToUserID` (the User ID is the low word of target)
11 = reserved

The `dataIn` parameter is defined separately for each request code. It can be a pointer or handle to some other data when that's convenient (the lifetime and ownership of that data is defined by the request code).

The `dataOut` buffer has this form:

+000 Word	`recvCount`—number of times the message was accepted
+002 ...	Depends on the request code

If `dataOut` is NIL, `SendRequest` does not attempt to fill in the `recvCount` field. So if the recipient does not require an output buffer and you don't care how many times the request is accepted, you may pass NIL for `dataOut`.

Requests are offered to request procedures starting with the most-recently-installed procedure and working backwards. In practice, this usually means that the system software can provide a standard behavior, that third-party desk accessories and initialization files can override the standard behavior, and that the current application can override everything.

By the way, the `recvCount` word is only filled in when the `SendRequest` call is finishing up. A request procedure cannot examine the word to determine how many procedures have handled the request so far.

033 Video Overlay Update

New Features of the Video Overlay tools

•`VDGGStatus` did not previously work with selector value $11 (`LineInterrupts`); it always crashed when given that selector value.

Now, in tool set version $0103, it works properly.

014 Window Manager Update

New Features of the Window Manager

•`WindStatus` now returns with the carry set and A=0 when the Window Manager is started up but a window update is in progress (`BeginUpdate` has been called more times than `EndUpdate`). This causes GS/OS to put disk-request alerts on the text screen instead of calling `AlertWindow`, when a window update is in progress.

•Fixed window title clipping so that it doesn't draw onto the desktop if the window is extremely narrow (this is still a cosmetic problem for ROM 3).

•`GetSysWFlag` and `GetWKind` are now guaranteed to return FALSE when passed a window pointer of NIL. (Previously, the result was unpredictable.)

`ErrorWindow` enhancements

•Most `ErrorWindow` dialogs look nicer. For example, icons are included, and the button typically says "Continue" rather than "OK" (after the user gets an error, things are generally not OK!).

•When `ErrorWindow` calls `AlertWindow`, it sets `alertFlags` bit 5 to put the button or buttons on the right.

•When `ErrorWindow` is called with an error in the range $0000 to $00FF, it calls `SysBeep2` with a `beepType` of $CE00 to $CEFF, so that system extensions can provide audio feedback after the dialog draws. For errors not in the $0000 to $00FF range, `ErrorWindow` calls `SysBeep2`($CEFF).

•`ErrorWindow` now correctly returns with the carry flag clear if no error occured.

`AlertWindow` enhancements

•`AlertWindow` no longer hangs when there is a caret (^) in the message string.

•`AlertWindow` now allows the separator character to appear inside substitution strings with no side effects. Such characters are never treated as separators, so there is no need to do special processing.

•`AlertWindow` is now able to refresh the contents of its window (for example, if the window is temporarily obscured by the Video Keyboard window).

•The standard `AlertWindow` icons are now colorful (they come from Sys.Resources).

•When the Disk-swap icon (icon "6") is used, `AlertWindow` automatically cooperates with GS/OS to watch for the user inserting a disk. When it notices an insertion, it

automatically blinks the default button and returns to the caller. (There is a flag bit to enable this behavior without using a disk-swap icon.)

•Nearly all buttons appearing in `AlertWindow` have key equivalents. Return is equivalent to the bold-outlined default button, as always. Esc and Command-period are equivalent to a button with the name "Cancel". All other buttons get their title's first letter as their key equivalent (in both upper- and lower-case if it's a letter).

(A button other than "Cancel" or the default button receives no key equivalent if its first letter is the same as the first letter of any other button. Leading blanks are ignored.)

•Bit 3 ($0008, `awTextFullWidth`) in the `alertFlags` parameter makes `AlertWindow` ignore the width of the icon when computing the rectangle for the text (this provides more control when centering text).

•`AlertWindow` sometimes calls `SysBeep2` with a `beepType` computed from the icon number in the alert string. In some cases, the call happens only if bit 4 ($0010, `awForceBeep`) in the `alertFlags` parameter is set.

Icon#	Meaning	SysBeep2 call
0	none	$C050 if flag bit 4 set
1	custom	none
2	Stop	$C052
3	Note	$C053 if flag bit 4 set
4	Caution	$C054
5	Disk	none
6	Disk-swap	$C030 (always)

•Bit 5 ($0020, `awButtonLayout`) in `alertFlags` makes `AlertWindow` position the buttons in the latest cool way. If there's one button, it goes in the lower right. If there are two, they are clustered in the lower right. If there are three, the first one is way on the left, and the last two are clustered on the right. (See Inside Macintosh VI chapter 2 and Human Interface Note #10.)

•Bit 6 ($0040, `awNoDevScan`) in `alertFlags` makes `AlertWindow` skip the initial call to `ScanDevices`, where it ignores any as-yet-unnoticed disk inserts. GS/OS will set this bit when asking for a disk to be inserted, so that if you inserted one in a device it just finished polling, `AlertWindow` notices it with no further user action.

•Bit 7 ($0080, `awNoDisposeRes`) in `alertFlags` is defined when the alert string is passed by resource ID. This bit makes `AlertWindow` release the resource to purge level 3 instead of disposing of it completely. If your resource is locked and you set this flag bit, your resource will remain in memory.

•Setting bit 8 ($0100, `awWatchForDisk`) in `alertFlags` makes `AlertWindow` watch for disk insertions, just like when you use the disk-swap icon.

•Setting bit 9 ($0200, `awIconIsResource`) in `alertFlags` indicates that the four imbedded icon-pointer bytes are the resource ID of an `rIcon` resource, rather than a pointer to an `PaintPixels`-style icon. Only set this bit when imbedding icon information in the alert string. Note: `AlertWindow` assumes that the `LoadResource` call for the icon will succeed. You can make sure it will by pre-flighting it (do the `LoadResource` yourself first).

•Setting bit 10 ($0400, `awFullColor`) in `alertFlags` sets the alert window's font flags to $0004 to allow 16-color text in 640 mode.

`Desktop(checkForNewDeskMsg)`

Selector 8, `checkForNewDeskMsg`, to the `Desktop` call causes the system to re-check the `MessageCenter` for a new desk message. (This is not a new feature.)

In System 6, `Desktop(checkForNewDeskMsg)` calls `SendRequest` with request code $0008, `systemSaysNewDeskMessage` so that applications with custom desktop-drawing routines can easily discover that there may be a new desktop pattern or picture.

For fakeModalDialog Users

The features of the Developer Technical Support `fakeModalDialog` tool set (version 1.0) are now present in the Window Manager, Control Manager, and QuickDraw II Auxiliary.

The following table summarizes where the various calls went. The new calls are in the Window Manager except as noted. Several new calls begin with "MW" for "Modal Window."

`fakeModalDialog` call	System 6.0 call
`fakeModalDialog`	`DoModalWindow`
`fmdEditMenu`	`MWSetUpEditMenu`
`fmdFindCursorCtl`	`FindCursorCtl`
`fmdGetCtlPart`	`MWGetCtlPart`
`fmdGetError`	no equivalent (check toolbox error codes directly)
`fmdGetIBeamAdr`	no equivalent (use `IBeamCursor`, `GetCursorAdr`)
`fmdGetMenuProc`	`MWSetMenuProc` (returns previous value)
`fmdIBeamCursor`	`IBeamCursor` (QuickDraw II Auxiliary)
`fmdInitIBeam`	no equivalent
`fmdLEGetText`	`GetLETextByID` (Control Manager)
`fmdLESetText`	`SetLETextByID` (Control Manager)
`fmdSetIBeam`	no equivalent
`fmdSetMenuProc`	`MWSetMenuProc`
`fmdStdDrawProc`	`MWStdDrawProc`
`fmdWhichRadio`	`FindRadioButton` (Control Manager)

New Window Manager Calls

DoModalWindow $640E

DoModalWindow handles user interaction in a window containing extended controls. Doing modal dialogs with DoModalWindow is much more flexible than using the Dialog Manager.

Modal dialogs handles by DoModalWindow can optionally be movable (they can be dragged by their title bar, and they also have an "alert frame" inside). You can also let the user bring desk accessories in front or the modal dialog. The dialog is still modal in that no other application windows can be selected until the dialog is dismissed.

Note: DoModalWindow is very similar to the fakeModalDialog call in the DTS Libraries and Tools. See the section For fakeModalDialog Users earlier in this chapter.

Here is a typical sequence involving DoModalWindow:

1. Create a window using NewWindow or NewWindow2. Create extended controls along with it, or use NewControl2 to create them separately.

2. Call DoModalWindow repeatedly. It returns even if nothing interesting happened. If the user did something, the result from DoModalWindow tells you what.

3. When the user finally does something to dismiss your window (like clicking OK or Cancel), call retrieve any information needed from the controls—radio button states (FindRadioButton), check box states (GetCtlValue), text field contents (GetLETextByID, TEGetText), etc.—and then use CloseWindow to close the window.

4. If there were any Edit Line or Text Edit fields in your winodw, DoModalWindow may have left the cursor set to an I-Beam. Call InitCursor or SetCursor to change it to something known.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Long—Space for result
|-----------------|
|    eventPtr     |   Long—Pointer to an extended task record
|-----------------|
|   updateProc    |   Long—Pointer to update procedure, NIL for
standard
|-----------------|
|    eventHook    |   Long—Pointer to event hook routine, NIL for none
|-----------------|
|    beepProc     |   Long—Pointer to “beep” procedure (see
description)
|-----------------|
|      flags      |   Word—bit flags describing how this dialog behaves
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|        ID       |   Long—ID of control or menu item was selected
|-----------------|
                   <—SP
```

Errors none

```
C           extern pascal Long DoModalWindow(eventPtr, updateProc,
                  eventHook, beepProc, flags);
            EventRecordPtr eventPtr;
            VoidProcPtr updateProc, eventHook, beepProc;
            Word flags;
```

`eventPtr` The address of an extended task record to be used for calls to `GetNextEvent`.

In 6.0, `DoModalWindow` does not call `TaskMaster`, so don’t expect all the Task Record fields to be filled in. In particular, `DoModalWindow` does not count multiple clicks for you (`wmClickCount` does not get set).

`updateProc` Pointer to a routine to be called when the modal dialog window needs to be updated. This address is stored in the window’s `wContDraw` field. If this address is NIL, `DoModalWindow` stores the address of the standard draw procedure (`MWStdDrawProc`), which calls the Control Manager routine DrawControls and draws an interior alert frame if appropriate.

`eventHook` The address of a routine to be called with the results of `GetNextEvent` (or NIL if none). Your event hook procedure can look in the event record

pointer to by `eventPtr` and change fields as necessary. The event hook routine receives a single pointer on the stack, which is must remove before returning with an RTL.

If bit 31 of `eventHook` is set, `DoModalWindow` translates Command-period keypresses into Escape keypresses before calling any `eventHook` routine. (It's okay to pass $80000000 to translate Command-periods but not provide an event hook routine.)

`beepProc` Pointer to a routine to be called when the user clicks outside the modal dialog box. If this is NIL, `DoModalWindow` calls the Miscellaneous Tools routine `SysBeep2` with `beepType` $0004.
If `beepProc` is -1 ($FFFFFFFF), `DoModalWindow` does nothing. You can use this routine to alert the user in different ways, like playing custom sounds or flashing the menu bar.

`flags` Bit flags indicating how the modal dialog box should behave.

`mwMovable`	bit 15	Indicates whether or not the modal dialog box should be a movable modal dialog box. Regardless of this bit's setting, the window must have a drag region or it cannot move. 0 = Dialog box is not movable 1 = Dialog box is movable if it has a drag region (title bar)
`mwUpdateAll`	bit 14	Indicates whether `DoModalWindow` should handle update events for all windows or just the frontmost application window, which is the dialog box. `DoModalWindow` calls the routine in the `wContDraw` field of the window record to update other windows; all windows must have either NIL or a valid content-draw procedure in their `wContDraw` fields. 0 = Update only the dialog window 1 = Update all application windows as needed Note: System windows (NDAs, for example) automatically update during `GetNextEvent`, regardless of the `mwUpdateAll` setting.
reserved	bits 13-6	Reserved for future expansion. Must be set to zero.
`mwWantActivate`	bit 5	Indicates whether `DoModalWindow` should return activate events to the caller. 0 = handle activates internally and then process another event without returning to the caller (this is faster) 1 = return to the caller after handling an activate event

`mwDeskAcc`	bit 4	Indicates whether `DoModalWindow` should automatically handle desk accessories. 0 = Don't handle desk accessories 1 = Automatically handle desk accessories
`mwIBeam`	bit 3	Indicates whether `DoModalWindow` should automatically use the I-beam cursor when the hot spot is over an Edit Line or Text Edit control. 0 = Use an arrow cursor everywhere 1 = Use an I-beam cursor over editable text controls
`mwMenuKey`	bit 2	Indicates whether menu key events should be treated as menu events first. `DoModalWindow` will handle standard editing menu key events even if the menu key events are diabled. All key events will be treated as regular key-down events if `MenuKey` returns FALSE. 0 = Treat menu key events as key down events 1 = Treat menu key events as menu events
`mwMenuSelect`	bit 1	Indicates whether the user can pull down menus. `DoModalWindow` handles standard editing menu selections for Edit Line and Text Edit controls automatically. 0 = Don't allow pull-down menu selections 1 = Allow pull-down menu selections
`mwNoScrapForLE`	bit 0	Indicates whether cut, copy, and paste operations on Edit Line controls should use the Scrap Manager. Not using the desk scrap means that text cut or copied from Edit Line controls can't be pasted into Text Edit controls; however, this may be useful when trying to protect a large desk scrap from being accidentally overwritten. 0 = Use the desk scrap for Edit Line editing 1 = Don't use the desk scrap for Edit Line editing

`ID` An indication of what action the user took. If the user selected a control, this field is the control ID. If the user selected a menu item, the high word of ID is the menu ID and the low word of ID is the menu item ID. To distinguish between control IDs and menu selections, bit 31 is set if a menu was selected. Therefore, all control IDs used in the modal dialog window must have bit 31 clear and all menu IDs used with the modal dialog window must have bit 15 clear, or you will be unable to distinguish control selections from menu selections. (If both pull-down menus and menu key events are disabled in flags, your control ID values may have bit 31 set with no ill effects.)

If ID is zero, you can examine the event record at `eventPtr` to see what happened (for example, a keypress or mouse click not claimed by any control, or an update or activate event).

Note that when a hit on a control is returned, `DoModalWindow` has put the Control Handle into the `TaskData2` field of your extended task record.

Differences between `fakeModalDialog` and `DoModalWindow`

If you set bit 31 of the `eventHook` procedure pointer, `DoModalWindow` automatically converts Command-period keypresses into Escape keypresses. After that, it still calls your event hook routine if the rest of the pointer is non-NIL.

`DoModalWindow` calls `SysBeep2`($0004) if you click outside the dialog window inappropriately.

`DoModalWindow` uses an event mask of $0FFF when it calls `GetNextEvent`. It used to use $FFFF, so it was claming app1..app4 events, which was generally not appropriate.

FindCursorCtl $690E

Note: FindCursorCtl is very similar to the fmdFindCursorCtl call in the DTS Libraries and Tools. See the section For fakeModalDialog Users earlier in this chapter.

Returns the handle for the control beneath a given point. DoModalWindow uses this routine in I-beam cursor handling; if the control under the cursor is an Edit Line or Text Edit control, DoModalWindow changes to an I-beam cursor.

Note that the xLoc and yLoc values are in local coordinates of the specified window.

FindCursorControl does not care about the hilite state of a control. It treats inactive controls (hilite $FF) just like any other controls.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Word—Space for result
|-----------------|
|  ctlHandlePtr   |   Long—Pointer to space for control handle
|-----------------|
|      xLoc       |   Word—local X coordinate of point
|-----------------|
|      yLoc       |   Word—local Y coordinte of point
|-----------------|
|     windPtr     |   Long—Pointer to window to check (NIL=front)
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|    partCode     |   Word—Part code for control beneath the given
point
|-----------------|
                   <—SP
```

Errors none

```
C           extern pascal unsigned int FindCursorCtl(ctlHandlePtr,
                  xLoc, yLoc, windPtr);
            CtlRecHndlPtr     ctlHandlePtr;
            Integer           xLoc;
            Integer           yLoc;
            WindowPtr         windPtr;
```

GetAuxWindInfo $630E

GetAuxWindInfo returns a pointer to a block of auxiliary data for a specified window. If the window doesn't have an auxiliary info record yet, one is created and filled with zeroes.

The auxiliary window record is for the system's convenience, but some fields may be set by applications and utilties. (See the Desk Manager Update.)

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Long—Space for resulting pointer
|-----------------|
|     windPtr     |   Long—window pointer
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|   auxInfoPtr    |   Long—pointer to auxiliary window record
|-----------------|
                   <—SP
```

Errors $0201 could not allocate memory

```
C           extern pascal Ptr GetAuxWindInfo(theWindow);
            Ptr   theWindow;
```

Here is Auxiliary Window structure:

+000	Word	size in bytes (currently 28; may grow in the future)
+002	Word	*bank register value (low byte of word)
+004	Word	*direct-page register value
+006	Word	*resource app value
+008	Long	*old update region handle
+012	Long	*old port (for EndUpdate)
+016	Long	*window layer (for Windoid support)
+020	Word	min vertical size for System window
+022	Word	min horizontal size for System window
+024	Long	NDA Structure Pointer (see the Desk Manager update)

Fields marked with an asterisk (*) are reserved for future use.

HandleDiskInsert $6B0E

HandleDiskInsert lets an application know about disks the user has inserted or ejected. When an inserted disk cannot be read or identified, the user gets a chance to initialize the disk.

When a disk is not claimed by any installed file system, HandleDiskInsert calls SendRequest to see if the file system can be identified. If it can, the procedure accepting the request gives relevant information to the user and then tells HandleDiskInsert whether to eject the disk, leave it online, initialize it, or erase it.

The standard system routine for identifying unrecognized disks is discussed in detail below. Applications and utilities may with to install AcceptRequest procedures to identify additional disk formats.

When a duplicate disk is inserted, the user gets a chance to rename it.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Word—space for result
|-----------------|
|      Space      |   Word—space for result
|-----------------|
|      flags      |   Word—input flags (see below)
|-----------------|
|      devNum     |   Word—GS/OS device number
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|   resultFlags   |   Word—resulting flags
|-----------------|
|   resultDevNum  |   Word—resulting device number
|-----------------|
                   <—SP
```

Errors GS/OS errors returned unchanged

```
C           extern pascal long HandleDiskInsert(flags, devNum);
            word  flags, devNum;
```

flags Tells HandleDiskInsert how to proceed.
bit 15 (hdiScan): 1 = scan devices looking for inserts and ejects

bit 14 (`hdiHandle`): 1 = identify a disk and handle user interaction if the disk is not usable
bit 13 (`hdiUpdate`):
1 = update the status of a particular device or all devices (most applications have no need for this)
bit 12 (`hdiReportEjects`):
1 = report any detected disk-ejects as well as inserts
0 = report only inserts, not ejects
bit 11 (`hdiNoDelay`):
1 = bypass the normal 1-second scanning delay; scan immediately
0 = scan only if at least 60 ticks have elapsed since the previous scan
bit 10 (`hdiDupDisk`):
1 = prompt user to rename or eject disk, as if the `Volume` call returned a duplicate-disk error (if you set this bit, you must also set bit 14, and devNum must be valid)
0 = don't simulate a duplicate-disk error
bit 9 (`hdiCheckTapeDrives`):
1 = scan Apple SCSI tape drives
0 = do not scan Apple SCSI tape drives
(This bit applies to scanning and to updating the status of a particular tape drive device.)
bit 8 (`hdiUnreadable`): 1 = the disk being processed is known to be unformatted (simulate an I/O error to save time)
bits 7-1: Reserved, use 0.
bit 0 (`hdiMarkOffline`): 1 = Mark all devices as offline, so that disk inserts are reported for all already-online volumes

`devNum` Normally zero. Pass a nonzero device number if you already detected an insert & want the system to check it out & possibly ask the user to format it (flag bit 15 clear, 14 set); or if you are informing the system that you have already taken care of the new status of a particular device (flag bit 13 set).

`resultDevNum` the device number of a device that was inserted or ejected. Zero if nothing happened, or if the user chose to Eject a disk that was discovered to be inserted.

`resultFlags` Bits 15-2: Reserved (ignore).
Bit 1 = set if `resultDevNum` represents a disk that the user elected to format.
Bit 0 = set if `resultDevNum` is an ejection.

Discussion

Scanning for Inserts and Ejects

Scanning occurs only if flag bit 15 is set. Once an insertion is detected, it is either handled as described in the next section, or the device number is returned directly to the application (depending on flag bit 14).

• When scanning, Apple 5.25 devices are ignored, as are character devices.
• Block devices with non-removable media are still scanned, since it's important for some applications to get a "first time" insert for those devices.
• If 60 ticks have not elapsed since the last time `HandleDiskInsert` scanned devices, no scanning is performed. You can bypass this check by setting flag bit 11.
• `HandleDiskInsert` keeps an internal table recording its idea of the online/offline status of each device. When a device's status differs from the value recorded in this table, an insert or eject has occurred. This table is owned by the current application, not by desk accessories or other system components. At `WindStartUp` time, the table is initialized to show the current status of every device.

Handling an Insertion

Handling of an insertion occurs only if flag bit 14 is set. The device to handle comes from a scan as described above or is passed in as the `devNum` parameter, depending on the setting of bit 15.

• When an insert is detected, it does a `Volume` call on the device. If there's no error, keep looking for additional insertions if bit 15 is set. If there's nowhere else to look, return with no error.
• If Volume returns an I/O error, call `AlertWindow` asking the user to Eject or Initialize the disk.
    • If the user elects to Eject, make the `DControl` Eject call to get rid of it.
    • If the user elects to Initialize, make the GS/OS Format call to format it (letting the user name the disk and specify the file system & format options within the `Format` call).
    • If they cancel out of the format dialog, eject the disk and return no error.
    • If the format is successful, return the device number as the `resultDevNum`.
•If `Volume` returns an unrecognized-volume error, proceed as described under "Identifying Unknown Disks."
•If `Volume` returns a Duplicate Volume Name error and it is possible to rename the volume, give the user a chance to rename the disk, if possible.
• If the `Volume` call returns a strange error, leave it online, return the device number to the caller, and return the `Volume` error to the caller.

Identifying Unknown Disks

When the `Volume` call returns error $52 (unknown file system), `HandleDiskInsert` calls `SendRequest` with request code $0002, `systemSaysUnknownDisk`, to give utilities a chance to identify the disk and put up a special dialog. The low word of `dataIn` contains the GS/OS device number of the device in question; the high word is reserved and should be ignored. `dataOut` points to a buffer with the following format:

+000	`recvCount`	Word	used by `SendRequest`
+002	`reserved`	Word	reserved
+004	`disposition`	Word	result

`disposition` tells `HandleDiskInsert` what to do with the disk. Legal values are:

- $FFFF leave the volume online even though its was unrecognized. Report no error to the caller.
- $0000 eject the device
- $0001 make a GS/OS `Format` call on the device, letting the user choose the name, file system, and formatting options
- $0002 make a GS/OS `Erase` call on the device, letting the user choose the name and file system

If the `systemSaysUnknownDisk` request is not accepted, `HandleDiskInsert` puts up an `AlertWindow` reading:

> Using the installed File System Translators, GS/OS does not recognize this disk (in device .BLAHDEBLAH). Do you want to initialize it?

There are two buttons: a default Eject button, and an Initialize button (the action button, in the lower right). Option-initialize means Erase, as above.

(To simplify the user's choice, Erase is not presented as a separate button. If it were an explicit choice, the system would have to explain the risks of an Erase over an Initialize; users may not realize they have no guarantee that all the blocks on their disk are usable if they choose Erase.)

When call completes, the cursor is set to the same thing it was before. During the call, Arrow and Watch cursors are used.

The System UnknownDisk Procedure

The system installs a `AcceptRequests` procedure at boot time. When this procedure receives a request to identify a disk, it checks for the following file systems. If the file system can be identified and the corresponding file system translator is not already installed, the procedure displays a special message to the user, gets the user's response, and accepts the request.

Here are the messages that can be displayed:

> The disk in device .BLAHDEBLAH appears to be in Apple II Pascal format. Installing "File System: Pascal FST" (using the Installer) allows GS/OS to read this disk.

The disk in device .BLAHDEBLAH appears to be in Macintosh MFS format. System 6.0 cannot read MFS disks, but it can use the newer HFS format.

The disk in device .BLAHDEBLAH appears to be in HFS format. Installing "File System: HFS FST" (using the Installer) allows GS/OS to use this disk. HFS is used widely on the Macintosh.

The disk in device .BLAHDEBLAH appears to be in MS-DOS format. An MS-DOS File System Translator is required to use this disk.

The disk in device .BLAHDEBLAH appears to be in Apple II DOS 3.3 format. Installing "File System: DOS 3.3 FST" (using the Installer) allows GS/OS to read this disk.

The disk in device .BLAHDEBLAH appears to be in Apple II DOS 3.3 format. The DOS 3.3 File System Translator in System 6.0 only works with 5.25" disks.

The disk in device .BLAHDEBLAH appears to be in High Sierra format. Installing "Drive: CD-ROM" (using the Installer) allows GS/OS to use this disk.

The disk in device .BLAHDEBLAH appears to be in ISO 9660 format. Installing "Drive: CD-ROM" (using the Installer) allows GS/OS to read this disk.

The disk in device .BLAHDEBLAH appears to be in ProDOS format. Installing the ProDOS File System Translator allows GS/OS to read this disk.

Example

Many applications will simply call `HandleDiskInsert` each time through the main event loop, passing `flags` = $C000 (do scanning and handle inserts), `devNum` = 0, and ignoring `resultDevNum` and `resultFlags`.

MWGetCtlPart $650E

Returns the part code from any TrackControl call made by the most recent DoModalWindow call. (If DoModalWindow did not call TrackControl on the most recent call, then MWGetCtlPart returns zero.)

Note: MWGetCtlPart is very similar to the fmdGetCtlPart call in the DTS Libraries and Tools. See the section For fakeModalDialog Users earlier in this chapter.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Word—Space for control part
|-----------------|
                   <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|     ctlPart     |   Word—part code from DoModalWindow's
|-----------------|        last TrackControl call
                   <—SP
```

Errors none

```
C           extern pascal Word MWGetCtlPart();
```

MWSetMenuProc $660E

Note: MWSetMenuProc is a combination of the fmdSetMenuProc and fmdGetMenuProc calls in the DTS Libraries and Tools. See the section For fakeModalDialog Users earlier in this chapter.

MWSetMenuProc informs DoModalWindow of the address of a routine to be called when the frontmost window changes inside a DoModalWindow call. Since DoModalWindow can allow you to have both menus and desk accessory windows enabled, there may be menu items which should be enabled for your modal window and disabled for desk accessory windows. If this address is not NIL, DoModalWindow calls the procedure at this address whenever the frontmost window changes, giving your application the chance to change the state of menu items to reflect the new frontmost window.

Note: MWSetMenuProc is a combination of the fmdSetMenuProc and fmdGetMenuProc calls from the DTS Libraries and Tools. See the section For fakeModalDialog Users earlier in this chapter.

If you pass $FFFFFFFF (-1) for newMenuProc, then the menu proc is left unchanged, and MWSetMenuProc simply returns the address of the current menu proc.

NDA Note: If an NDA sets the menu proc, it must restore the old value before returning control to the application.

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      Space      |   Long—Space for result
|-----------------|
|   newMenuProc   |   Long—pointer to new menu procedure (or -1)
|-----------------|
                    <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
|   oldMenuProc   |   Long—pointer to previous menu procedure
|-----------------|
                    <—SP
```

Errors none

```
C           extern pascal VoidProcPtr MWSetMenuProc(newMenuProc);
            VoidProcPtr newMenuProc;
```

MWSetUpEditMenu $680E

Note: `MWSetUpEditMenu` is very similar to the `fmdEditMenu` call in the DTS Libraries and Tools. See the section For fakeModalDialog Users earlier in this chapter.

Sets the sate of the standard ememu items (Undo, Cut, Copy, Paste, Clear, and Close) based on the frontmost window.

If the frontmost window is a desk accessory window, `MWSetUpEditMenu` enables Undo, Cut, Copy, Paste, Clear, and Close. If the frontmost window is not a desk accessory window, `MWSetUpEditMenu` enables and disables items based on the target control in the window.

If the current target control is an Edit Line control, `MWSetUpEditMenu` enables Cut, Copy, and Clear if any text is selected. Paste is also enabled if a text scrap longer than zero bytes exists.

If the current target control is an editable Text Edit control, `MWSetUpEditMenu` enables cut, copy, and clear. Paste is also enabled if a text scrap longer than zero bytes exists.

If the current target control is a read-only Text Edit control, `MWSetUpEditMenu` enables Copy but disables Cut, Paste, and Clear.

In all other cases, `MWSetUpEditMenu` disables cut, copy, paste, and clear.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors none

```
C           extern pascal void MWSetUpEditMenu();
```

MWStdDrawProc $670E

Note: MWStdDrawProc is very similar to the fmdStdDrawProc call in the DTS Libraries and Tools. See the section For fakeModalDialog Users earlier in this chapter.

MWStdDrawProc is what DoModalWindow calls to update a modal window if you do not supply your own update procedure.

If you do provide your own update procedure, it may be convenient to call MWStdDrawProc from there, in addition to whatever else your procedure does.

MWStdDrawProc calls DrawControls on the window which is the current port, and it also draws an "alert frame" inside the window if necessary (if the window frame's fAlert bit is clear and its fFlex bit is set).

MWStdDrawProc draws in the current port, which must be a window.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors none

```
C           extern pascal void MWStdDrawProc();
```

ResizeInfoBar $6A0E

Sets the vertical size of a standard window's information bar. (Do not use this call with custom windows.)

Parameters

Stack before call

```
|Previous contents|
|-----------------|
|      flags      |   Word—flags (reserved, use 0)
|-----------------|
|     newSize     |   Word—New info bar height
|-----------------|
|     windPtr     |   Long—Pointer to window
|-----------------|
                  <—SP
```

Stack after call

```
|Previous contents|
|-----------------|
                  <—SP
```

Errors none

```
C           extern pascal void ResizeInfoBar(flags, newHeight, windPtr)
            Integer           flags;
            Integer           newHeight;
            WindowPtr         windPtr;
```

Appendix A—ToStrip and ToBusyStrip vectors

(These aren't particularly new, but they weren't documented before.)

These two vectors are for tool sets to jump to when a System Tool or User Tool function exits.

```
ToBusyStrip     $E10180
ToStrip         $E10184
```

Inputs: X = error code (0 if no error)
Y = number of bytes of input parameters to strip

Set up the registers and jump to `ToStrip`. It shifts the 6 bytes of `RTL` addresses up by Y bytes, sets up A and the carry appropriately, and returns to whoever called the tool.

If the scheduler `BUSY` flag needs to be decremented, jump to `ToBusyStrip` instead of `ToStrip`.

Appendix B—Battery RAM Use

New Battery RAM locations are documented here. Most applications have no need for this information.

Remember that all of Battery RAM belongs to the system, and that not all system uses of Battery RAM are documented.

$5A Key Translation setting

> Use `GetKeyTranslation` and `SetKeyTranslation` in the Event Manager to examine and modify this setting.

$5B CloseView settings:

```
cvPowerMask equ %00001111 ; bits 0-3 (magnification)
cvUseKeys   equ %00010000 ; bit 4
cvMagnify   equ %00100000 ; bit 5
cvInvert    equ %01000000 ; bit 6
cvEnabled   equ %10000000 ; bit 7
```

$5E Applications and Utilities Group settings

bit 0 set means no closed captioning ("visual indication of sounds")
bit 1 set means standard time, clear if daylight savings time
bit 2 set means not to have auto daylight savings; clear means to have auto daylight savings
bit 4-3 = amount of menu item blink (0..3)

$5F Miscalleneous Toolbox settings

bits 7-6: byte validity check (%10 if the byte has been initialized)
bits 5-3: reserved, should be zero
bit 2: 1 = no QD scanline interrupts (`QDStartUp` does `SetIntUse`(0))
bit 1: 1 = no `ShowBootInfo` icons
bit 0: 1 = alphabetize desk accessory lists

$60 Toolbox: `WaitUntil` scaling (see Miscellaneous Tools)

$61 Reserved for network medium selection

$62 Specifies OS for network boot (1=GS/OS, 2=ProDOS 8)