Programmer's Reference for System 6.0

Edited by Mike Westerfield

Contents

Acknowledgements

This book was developed from Apple's Engineering Requirements Specification (ERS) documents for System 6.0. The source material included:

Apple IIGS Toolbox Changes for System Software 6.0 (ERS version 2.5), David A. Lyons, February 26, 1992.

Apple IIGS Media Control Tool Set ERS (Version 1.0a3), Dan Hitchens.

GS/OS Enhancements in System 6, Greg Branche, dated April 3, 1992.

Apple IIGS Finder 6.0, System 6 Delta ERS v3.1CD, Andy Nicholas and Dave Lyons, March 5, 1992.

Information about MIDI Synth extracted from various sources by Tim Swihart.

Apple II Video Overlay Card, Developer Notes, May 31 1989.

Pascal File System Translator External ERS, v1.00 a03.

GS/OS AppleShare File System Translator External ERS, Version 0.26CD, Mark Day.

GS/OS DOS 3.3 FST ERS, version 1.2, October 31, 1990.

GS/OS HFS File System Translator External ERS, Version 0.09.

Addendum to SCSI Driver ERS, Version 2.0, Matt Gulick.

GS/OS AppleDisk 5.25 Driver ERS, Version 2.05.

The source material is quoted heavily. All source material is Copyright 1989, 1991, 1992, Apple Computer, Inc. It is used here with permission.

In addition, some of the material is derived from personal notes sent by Dave Lyons and Matt Deatherage. In particular, the material describing the modal dialog window calls at the start of the Window Manager chapter was written for this book by Matt Deatherage.

In addition to allowing me to develop this book from Apple's ERSs, some of the original System 6.0 engineers from Apple reviewed this book. The reviewers included Greg Branche, Matt Deatherage, and Dave Lyons.

I would like to thank them for taking the time to review the documentation so carefully. Matt even held marathon review sessions through Star Trek to finish the review – the ultimate sacrifice! (Dave says he really did watch it. Still, it's the though that counts.) While there are always errors in a document of this size and complexity, it would have been far, far worse if they hadn't taken the time to review this book. Naturally blame any remaining mistakes on the source – me!

Finally, folks tend to forget that it's the managers who help products happen, even if they aren't actually doing the development themselves. Tim Swihart made this book happen. Without Tim's help and managerial intercession at Apple, this book would not have been published. Thanks, Tim!

4700 Irving Blvd N.W. Suite 207
Albuquerque, N.M. 87114
(505) 898-8183

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

ALL IMPLIED WARRANTIES ON THIS MATERIAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, **NEITHER APPLE OR THE BYTE WORKS MAKES ANY WARRANTY OR REPRESENTATION, EITHER EXPRESSED OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

IN NO EVENT WILL APPLE OR THE BYTE WORKS BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESSED OR IMPLIED. No Apple or Byte Works dealer, agent, or employee is authorized to make any modification, extension, or addition to this warrantee.

Some states do not allow the exclusion or limitation of implied warrantees or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Chapter 1 **Apple Desktop Bus Tool Set Update**

This chapter contains new information about the Apple Desktop Bus Tool Set. The original reference to this tool set is in Volume 1, Chapter 3 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 26 of the *Apple IIGS Toolbox Reference*.

New Features

- `ADBVersion` now returns the same version number on ROM 1 and ROM 3 machines.

Chapter 2 **Audio Compression and Expansion Tool Set Update**

This chapter contains new information about the Audio Compression and Expansion Tool Set (ACE). The original reference to this tool set is in Volume 1, Chapter 27 of the *Apple IIGS Toolbox Reference*.

New Features

- There are two new tool calls used to deal with pieces of sound.

GetACEExpState $0D1D

`GetACEExpState` returns a table of internal values that record the current expansion state. By passing these values back to `SetACEExpState`, it is possible to restart the sound expansion process at the same point where `GetACEExpState` was called.

The method ACE uses to compress a run of bytes is dependent on all of the previous bytes it has compressed. To start expanding sound from an arbitrary point, there must be some way to record the compression state, then restore this state to start expanding at the arbitrary point. `GetACEExpState` is the first half of this process, returning the state of the sound compression engine at an arbitrary position in the sound sequence. `SetACEExpState` restores the state of the sound compression engine, so the program can begin expanding the sound from the recorded position.

Saving and restoring the sound state is a feature of the Audio IFF-C file format, and is discussed at length in File Type Note for File Type $D8, auxiliary type $0001.

Parameters

Stack before call

Previous contents	
— *theBuffer* —	**Long**—Pointer to buffer
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $1D03 `aceNotActive` The ACE tool set has not been started.

C

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

`theBuffer` Pointer to a 16-byte buffer that `GetACEExpState` fills with the current expansion state.

`SetACEExpState` $0E1D

`SetACEExpState` resets the expansion process so that subsequent calls to `ACEExpand` will expand sound starting from a previously recorded spot. Use `GetACEExpState` to record the parameters needed for this call.

The method ACE uses to compress a run of bytes is dependent on all of the previous bytes it has compressed. To start expanding sound from an arbitrary point, there must be some way to record the compression state, then restore this state to start expanding at the arbitrary point. `GetACEExpState` is the first half of this process, returning the state of the sound compression engine at an arbitrary position in the sound sequence. `SetACEExpState` restores the state of the sound compression engine, so the program can begin expanding the sound from the recorded position.

Saving and restoring the sound state is a feature of the Audio IFF-C file format, and is discussed at length in File Type Note for File Type $D8, auxiliary type $0001.

Parameters

Stack before call

Previous contents	
— *theBuffer* —	**Long**—Pointer to buffer
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $1D03 `aceNotActive` The ACE tool set has not been started.

C

```
extern pascal void SetAceExpState(theBuffer);
Ptr theBuffer;
```

`theBuffer` Pointer to a 16 byte buffer that has been filled in by a previous call to `GetACEExpState`.

Chapter 3 **Control Manager Update**

This chapter contains new information about the Control Manager. The original reference to this tool set is in Volume 1, Chapter 4 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 28 of the *Apple IIGS Toolbox Reference*.

Clarifications

- In a List control, to get the full functionality of the `memNever` bit (bit 5 of the member flags) in an item in the list, you must also set the `testMemNever` bit (bit 6 of the control's `ctlFlag` field). This has been true since System Software 5.0. See Chapter 10 of this book (which contains the update to the List Manager) for more information about the `memNever` and `testMemNever` bits.

- `MakeNextCtlTarget` always acts on the front window.

New Features

- There are two new controls, thermometers and rectangles. These are described later in this chapter.

- Pop-up menu controls have been enhanced. See the Menu Manager chapter for details.

- The parameter count for a LineEdit control template can now be either 8 or 9. (In the past, 8 was the only allowed value.) A parameter count of 9 is used for a password-style LineEdit item, where the characters are shown as some password character rather than the actual character typed. The new, ninth parameter is a word parameter containing the character to use.

- Icon buttons can now be disabled by setting bit 3 ($0008) of the Icon Button `flags` parameter. When this bit is set, the button does not track or return hits when you click on it.

- Scroll bars are now drawn with a gray background instead of a checkerboard pattern of black and white pixels.

  Make any changes to the color table before creating scroll bar controls. If the color table is changed while the scroll bar is visible, the background could look very odd. If the color table is changed before the control is created, the Control Manager scans the color table to find an appropriate gray. If none is found, the old checkerboard pattern is used.

  If you change color tables when a scroll bar already exists, you should call `CtlNewRes` so the Control Manager has a chance to notice and use an appropriate gray or checkerboard pattern.

- Scroll bar controls use the new Miscellaneous Tool Set routine `WaitUntil` to limit the scroll speed to 15 control value changes per second. This gives a consistent maximum speed, regardless of whether the computer is using an accelerator card.

- `HiliteControl` now uses the new Miscellaneous Tool Set routine `WaitUntil` to limit the blink speed for controls, so the blink speed is consistent whether or not the computer uses an accelerator card. The minimum time between blinks is now 4 ticks.

- Static Text controls can be drawn more quickly. To get the faster drawing speed, set bit 2 ($0004, `fBlastText`) in the `ctlFlag` field of the Static Text control. While this will speed up the process of drawing the text, there are some restrictions:

    - String substitutions are not performed.
    - Imbedded format characters are not allowed.
    - Word wrapping is not performed.
    - The control is not clipped to its bounding rectangle, so you need to be sure the text fits in the rectangle.
    - The Control Manager does not erase the unused part of the control rectangle for you, as it does if you do not set the `fBlastText` bit.

  Setting bit 3 ($0008, `fTextCanDim`) in the `ctlFlag` field of a Static Text control makes the text gray when the control is inactive. (The control can be inactive either because its highlight value is $00FF or because the window is inactive.) Setting `fTextCanDim` is not recommended for large amounts of text, since the system draws text normally before applying a pattern to make it gray.

  You can use `fTextCanDim` in conjunction with `fBlastText`.

- `LoadResource` now relocks handles, so custom control definition procedures no longer have to worry about getting called while their code is purgeable. This makes one section of the pre-May 1992 Apple IIGS Technical Note #81 obsolete.

- `HiliteControl` now uses `WaitUntil` in the Miscellaneous Tool Set to limit how fast a control can blink. When `HiliteControl` sets the highlight state of the most recently highlighted control to zero, it enforces a minimum wait of 4 ticks since the first highlight.

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

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

  If you have already written a targetable custom control, you should call `MakeNextCtlTarget` for Command-Tab as well as Tab.

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

- The new `FindRadioButton` tool call quickly determines which radio button in a family of radio buttons is selected.

- The new `SetLETextByID` and `GetLETextByID` tool calls make it easy to read or set the text in a LineEdit control.

New Controls

Rectangle

Rectangle controls are used to draw rectangles or lines (very thin rectangles) in a window.

It's often useful to include lines or rectangles in a window or dialog box to visually separate related groups of items. One example of this is the Finder's "Preferences" dialog box in Finder 6.0 – the checkboxes for controlling list view preferences are grouped together with a rectangle. Since all of the other items in this dialog are controls, the Rectangle control lets the Finder use this element with a content draw routine that does little more than call the Control Manager routine `DrawControls`.

When using Rectangle controls like this, make sure the Rectangle control is defined before the controls to be drawn within the rectangle. (With resource-based control lists, this means the Rectangle control appears *later* in the control list, since the controls are drawn starting with the last control in the list and working towards the start of the list.) This insures the Rectangle control is drawn first and other controls are drawn "on top" of the rectangle.

If you don't want to receive mouse clicks on a Rectangle control, set the control's highlight value to $FF.

The Rectangle control definition procedure has resource ID $07FF0003 in the system resource file.

Control Template:

$00	*pCount*	**Word**—parameter count (6, 8, 9 or 10)
$02	*ID*	**Long**—Application assigned control ID
$06	*rect*	**8 bytes**—Boundary rectangle for control
$0E	*procRef*	**Long**—Rectangle control = 87FF0003
$12	*flags*	**Word**—Control flags
$14	*moreFlags*	**Word**—Additional control flags
$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

`flags` Bits 15-8 are reserved and should be set to 0.
Set bit 7 for an invisible control, and clear this bit for a visible control.
Bits 6-2 are reserved and should be set to 0.
Bits 1-0 define the control's appearance:

- 00 Make the control transparent. The control isn't drawn, but hits are still reported.
- 01 Use a gray pattern.
- 10 Use a black pattern.
- 11 Reserved.

`moreFlags` This value should always be set to $1000. The actual bit definitions are:

Bit 15 (`fCtlTarget`) must be 0.
Bit 14 (`fCtlCanBeTarget`) must be 0.
Bit 13 (`fCtlWantEvent`) must be 0.
Bit 12 (`fCtlProcRefNotPtr`) must be 1.
Bit 11 (`fCtlTellAboutSize`) must be 0.
Bit 10 (`fCtlIsMultiPart`) must be 0.
Bits 9-0 are reserved and should be set to 0.

`penHeight` The pen height is used when drawing the border of the rectangle. It defaults to 1. If you specify a pen height, you must also specify a pen width.

`penWidth` The pen width is used when drawing the border of the rectangle. It defaults to 2. In 320 mode, the value for the pen width is divided by 2.

`penMask` Pen mask used when drawing the rectangle.

`penPattern` Pen pattern used when drawing the rectangle. Using this parameter overrides the pattern specified in `flags`, but the visible bits are still used, and should be set to 01.

Thermometer

A Thermometer control is a rectangle that gradually fills as an operation completes. At convenient intervals, your application calls `SetCtlValue`, 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.)

The Thermometer control defproc has resource ID $07FF0002 in the system resource file.

Control Template:

$00	*pCount*	**Word**—parameter count (8 or 9)
$02	— *ID* —	**Long**—Application assigned control ID
$06	*rect*	**8 bytes**—Boundary rectangle for control
$0E	— *procRef* —	**Long**—Thermometer control = $87FF0002
$12	*flags*	**Word**—Control flags
$14	*moreFlags*	**Word**—Additional control flags
$16	— *refCon* —	**Long**—Application assigned constant
$1A	*value*	**Word**—Determines position of mercury
$1C	*data*	**Word**—Determines scale
$1E	— *colorTableRef* —	***Long**—Color table reference

`flags` Set bit 0 ($0001) for a horizontal thermometer; leave bit 0 clear for a vertical thermometer. All other bits are reserved, and must be set to 0.

`moreFlags` Bit 15 (`fCtlTarget`) must be 0.
Bit 14 (`fCtlCanBeTarget`) must be 0.
Bit 13 (`fCtlWantEvent`) must be 0.
Bit 12 (`fCtlProcRefNotPtr`) must be 1.
Bit 11 (`fCtlTellAboutSize`) must be 0.
Bit 10 (`fCtlIsMultiPart`) must be 0.
Bits 9-2 are reserved, and should be set to 0.
Bits 1-0 define the type of the color table reference:

- 00 `colorTableRef` is a pointer.
- 01 `colorTableRef` is a handle.
- 10 `colorTableRef` is an `rCtlColorTbl` resource ID.

`colorTableRef` The format for the color table is:

$00	*outlineColor*	**Word**—$000w
$02	*interiorColor*	**Word**—$000x
$04	*foreColor*	**Word**—$000y
$06	*fillColor*	**Word**—$p00z

`w` is the outline color.
`x` is the interior color.
`y` is the foreground thermometer fill color for a dotted pattern.
`z` is the thermometer fill color.
`p` is 0 for a solid pattern, or 8 for a dotted pattern. (That is, set bit 15 for a dotted thermometer fill pattern.)

The default color table is:

$0000	Black outline.
$000F	White interior.
$0000	Not used, since the pattern is solid.
$0004	Solid red mercury.

Tip `GetCtlTitle` and `SetCtlTitle` deal with the `ctlData` field of the control record, so you can use these calls to read or 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

`FindRadioButton` 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).

Note `FindRadioButton` is very similar to the `fmdWhichRadio` call in the DTS Tools and Libraries.

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 (-1).

GetLETextByID $3B10

Note `GetLETextByID` is very similar to the `fmdLEGetText` call in the DTS Tools and Libraries.

`GetLETextByID` returns the text of a LineEdit control in a buffer supplied by the caller.

`GetLETextByID` saves you the trouble of calling `GetCtlHandleFromID` to get the LineEdit control handle, retrieving the LineEdit record from the control (using `GetCtlTitle`) and then making LineEdit 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 string starting at the buffer's beginning or as a C string starting at the buffer's second byte. Strings with a length over 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 LineEdit control. Be sure the buffer is long enough to hold the maximum number of characters possible in the LineEdit 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 a LineEdit control. Specifying the control ID of anything other than a LineEdit control is a bad thing.

Note This call is in the Control Manager instead of the LineEdit Tool Set because it works with LineEdit *controls* and **not** LineEdit *records*.

Parameters

Stack before call

Previous contents	
windPtr	**Long**—Pointer to the window containing the control
LECtlID	**Long**—Control ID for the LineEdit control
textPtr	**Long**—Pointer to the result buffer
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$1004	`noCtlError`	No controls in window.
$1005	`noExtendedCtlError`	No extended controls in window.
$1009	`noSuchIDError`	The specified ID cannot be found.
$100C	`noFrontWindowError`	There is no front window.

C

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

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

`LECtlID` The control ID of the LineEdit control from which to retrieve 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. If the length of the string exceeds 255 bytes, the length byte is set to 0.

Since the string is preceded by a length byte and followed by a null terminator, you must be sure the buffer is at least two bytes longer than the longest possible string the LineEdit control will allow.

SetLETextByID $3A10

Note `SetLETextByID` is very similar to the `fmdLESetText` call in the DTS Tools and Libraries.

`SetLETextByID` sets the text of an LineEdit control to a string supplied by the caller, selects all of the text, and invalidates the `viewRect` of the LineEdit 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 LineEdit control handle, retrieving the LineEdit record from the control (using `GetCtlTitle`) and then making LineEdit tool calls to actually set the text and the selection.

△ **Important** `SetLETextByID` does no checking to insure that the control ID specified belongs to a LineEdit control. Specifying the control ID of anything other than a LineEdit control is a bad thing. △

Note This call is in the Control Manager instead of the LineEdit Tool Set because it works with LineEdit *controls* and **not** LineEdit *records*.

Parameters

Stack before call

Previous contents	
windPtr	**Long**—Pointer to the window containing the control
LECtlID	**Long**—Control ID for the LineEdit control
textPtr	**Long**—Pointer to Pascal string
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$1004	`noCtlError`	No controls in window.
$1005	`noExtendedCtlError`	No extended controls in window.
$1009	`noSuchIDError`	The specified ID cannot be found.
$100C	`noFrontWindowError`	There is no front window.

Errors from `LESetText` are returned unchanged.

C

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

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

`LECtlID` The control ID of the LineEdit control to receive the text.

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

Chapter 4 **Desk Manager Update**

This chapter contains new information about the Desk Manager. The original reference to this tool set is in Volume 1, Chapter 5 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 29 of the *Apple IIGS Toolbox Reference*.

New Features

Classic Desk Accessory Changes

- If bit 0 of Battery RAM location $5F is set, the Desk Manager sorts the Classic Desk Accessory (CDA) menu alphabetically. However, `Control Panel` always remains at the top, and `Quit` always remains at the bottom. (There is a checkbox in the General control panel to enable and disable the sorting 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 (Escape) moves to the `Quit` item at the bottom.

- CDA names may now contain Control-N. Any part of the name following the Control-N is displayed as normal text even when the CDA is highlighted 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 New Desk Accessory (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. (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 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 the frontmost window, 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 $18 for a pointer to an structure with the following format:

$00	*status*	**Word**—Use $0000 (reserved for the Desk Manager)
$02	— *openProc* —	**Long**—Reserved (use 0)
$06	— *closeProc* —	**Long**—Pointer to the NDA style Close routine
$0A	— *actionProc* —	**Long**—Pointer to the NDA style Action routine
$0E	— *initProc* —	**Long**—Reserved (use 0)
$12	*period*	**Word**—Reserved (use 0)
$14	*eventMask*	**Word**—Event mask, just like for an NDA
$16	— *lastServiced* —	**Long**—Reserved (use 0)
$1A	— *windowPtr* —	**Long**—Reserved (use 0)
$1E	— *ndaHandle* —	**Long**—Reserved (use 0)
$22	*memoryID*	**Word**—Your memory ID (important for resource application switching)

This allows NDAs to have more than one (modeless) window. It also allows Finder extensions or code 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, `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:

$00	*result*	**Word**—Space for result
$02	*fwValue*	**Word**—Value returned from `FindWindow`
$04	— *windowPtr* —	**Long**—Window pointer
$08	— *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). If you do, `SystemClick` exits, taking no further action.

CallDeskAcc $2405

`CallDeskAcc` calls an NDA's Action or Initialization 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.

Tip `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`.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
flags	**Word**—Flags
— *daReference* —	**Long**—Specifies which desk accessory to call
action	**Word**—Action code to pass to desk accessory
— *data* —	**Long**—Data value to pass to desk accessory
	<—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: 0 = Call an NDA; a value of 1 is reserved for calling a CDA.
- bits 14-2: Reserved; set to 0.
- bit 1: 1 = Call the NDA's Initialization 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 to `GetNumNDAs`.

`daReference` 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` The value to pass in the A register when the desk accessory is called. This will be the action code passed to the action routine if bit 1 of the flags bit is clear.

Action codes greater than 9 are only sent to the desk accessory if its event mask is $Axxx. If a value greater than 9 is passed when the event mask is not of the form $Axxx, an error $0520 is returned. A desk accessory with an event mask of $Axxx is supposed to ignore any action codes it does not recognize.

`data` Value to pass to the desk accessory in the X and Y registers. When the desk accessory is called, the most significant word of this value is passed in the X register, while the least significant word is passed in the Y register.

GetDeskAccInfo $2305

`GetDeskAccInfo` gives safe access to certain information about the desk accessories currently installed in the system.

Parameters

Stack before call

Previous contents	
flags	**Word**—Flags
— *daReference* —	**Long**—Specifies which desk accessory to get information about
buffSize	**Word**—Size of the result buffer
— *bufferPtr* —	**Long**—Pointer to the 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 about a CDA; 0 = get information about an NDA.
- bits 14-1: Reserved; set to 0.
- bit 0: 1 = `daReference` is a window pointer; 0 = `daReference` is an index number.

`daReference` 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 a result buffer. For a CDA, the result buffer has this format:

$00	*size*	**Word**—Returned data size
$02	— *theHandle* —	**Long**—Handle to the CDA

The buffer format for an NDA is:

$00	*dataSize*	**Word**—Returned data size
$02	*status*	**Word**—NDA status; 0 if closed, non-zero if open
$04	— *openPtr* —	**Long**—Pointer to the NDA's Open routine
$08	— *closePtr* —	**Long**—Pointer to the NDA's Close routine
$0C	— *actionPtr* —	**Long**—Pointer to the NDA's Action routine
$10	— *initPtr* —	**Long**—Pointer to the NDA's Initialization routine
$14	*period*	**Word**—NDA's period
$16	*eventMask*	**Word**—NDA's event mask
$18	— *tickCount* —	**Long**—Tick count for the last Run event sent to the NDA
$1C	— *windowPtr* —	**Long**—NDA's main window pointer, `NIL` if none
$20	— *theHandle* —	**Long**—Handle to the NDA
$24	*userID*	**Word**—NDA's user ID

GetDeskGlobal $2505

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

`GetDeskGlobal($0000)` returns 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 which 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;
```

`selector` Indicates the kind of information to return. The only value currently defined is $0000, which tells the Desk Manager to return the last window examined.

Chapter 5 **Dialog Manager Update**

This chapter contains new information about the Dialog Manager. The original reference to this tool set is in Volume 1, Chapter 6 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 30 of the *Apple IIGS Toolbox Reference*.

New Features

- The standard icons for `NoteAlert`, `StopAlert` and `CautionAlert` now use colors.
- 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 a LineEdit 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 in ROM 3 machines.
- The default `ErrorSound` procedure now calls `SysBeep2` 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)`

Chapter 6 **Event Manager Update**

This chapter contains new information about the Event Manager. The original reference to this tool set is in Volume 1, Chapter 7 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 31 of the *Apple IIGS Toolbox Reference*.

New Features

- `GetNextEvent` is patched to fix a ROM 3 `ModalDialog` bug. This fix prevents `ModalDialog` from stealing `app1` through `app4` events.

- `GetNextEvent` now dispatches any deferred `SysBeep2` request.

- `EMStartUp` and `EMShutDown` new preserve the cursor location in the event of a smooth application launch, when the Super Hi-Res 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 Hi-Res screen is not already turned on.

Chapter 7 **Font Manager Update**

This chapter contains new information about the Font Manager. The original reference to this tool set is in Volume 1, Chapter 8 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 32 of the *Apple IIGS Toolbox Reference*.

New Features

- `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 the Tab key moves you between these and the "other size" LineEdit 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 checkboxes. Esc and Command-period (Command-.)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 offset/width table value (a combination of the `owTOffset` and `highOWTLoc` fields) 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.

Chapter 8 **Integer Math Tool Set Update**

This chapter contains new information about the Integer Math Tool Set. The original reference to this tool set is in Volume 1, Chapter 9 of the *Apple IIGS Toolbox Reference*.

New Features

- `IMVersion` now returns the same version number on ROM 1 and ROM 3 machines.

Chapter 9 **LineEdit Tool Set Update**

This chapter contains new information about the LineEdit Tool Set. The original reference to this tool set is in Volume 1, Chapter 10 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 34 of the *Apple IIGS Toolbox Reference*.

New Features

- LineEdit controls now support Shift-clicking to extend a selection. (This always worked in LineEdit records, but it didn't work for LineEdit controls.)
- In older versions of LineEdit, `LETextBox` would strip a random amount of stuff from the stack if called with a `textLength` parameter of zero. This has been fixed.
- The default password character is now a hollow diamond instead of an asterisk.
- LineEdit 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 (at address $xx0000).

Note See the Control Manager Update for information about changes to LineEdit controls.

Chapter 10 **List Manager Update**

This chapter contains new information about the List Manager. The original reference to this tool set is in Volume 1, Chapter 11 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 35 of the *Apple IIGS Toolbox Reference*.

Clarifications

- 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.

- *Apple IIGS 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` bit.
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).

New Features

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.

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, and typographical quotation marks (“”) are treated as normal quotation marks ("). See the table at the end of this section for a complete list of translations.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
flags	**Word**—Flags (reserved; use 0)
— *string1* —	**Long**—Pointer to the first Pascal string
— *string2* —	**Long**—Pointer to the second Pascal string
	<—SP

Stack after call

Previous contents	
order	**Word**—0 if equal; -1 if string1<string2; 1 if string1>string2
	<—SP

Errors None.

C
```
extern pascal Word CompareStrings(flags, string1, string2);
Word flags;
Ptr string1, string2;
```

`flags` Reserved; must be 0.

`string1` Pointer to the first Pascal string to compare.

`string2` Pointer to the second Pascal string to compare.

`order` $0000 if the two strings are equal. $FFFF if `String1` comes before `String2`. $0001 if `String1` comes after `String2`.

Translations

The strings are compared as if certain characters were changed into other characters before the compare is performed. For example, using strict ASCII comparisons, ‘a’ comes after ‘C’, since the ASCII character code for ‘a’ ($61) is larger than the ASCII character code for ‘C’ ($43). The character translations would convert the ‘a’ to an ‘A’, though, sorting the characters in standard case insensitive alphabetical order.

The original strings are not modified.

- ‘a’..‘z’ become ‘A’..‘Z’.
- Characters $80..$9F, $CB, $CC and $CD become unaccented capital letters.
- ‘¢’ ($A2) becomes ‘C’.
- ß ($A7) becomes ‘B’.
- ‘`’ ($AB) becomes an apostrophe (').
- ‘Ø’ ($AF) becomes ‘0’.
- ‘¥’ ($B4) becomes ‘Y’.
- ‘æ’ ($BE) becomes ‘Æ’ ($AE).
- ‘ø’ ($BF) becomes ‘0’.
- The ‘«’ and ‘»’ characters become quotation marks.
- Character $CA (the non-breaking space) becomes a space.
- ‘œ’ ($CF) becomes ‘Œ’ ($CE).
- ‘–’ ($D0, the long hyphen) becomes ‘-’ (the minus sign).
- ‘—’ ($D1, the dash) becomes ‘-’ (the minus sign).
- Typographical quotes (‘“’ and ‘”’) become plain quotes.
- Typographical single-quotes (‘‘’ and ‘’’) become apostrophes.
- ‘ÿ’ ($D8) becomes ‘Y’.
- Characters $D9..$F5 are not in Shaston (which contains all of the “standard” Apple characters), but are translated appropriately for international purposes.
- All other characters are 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; that is, you can type the first N characters of any item to jump to it. For prefix strings to work in a reasonable way, the list must be sorted. The list should be sorted using a case insensitive sort, like the one performed by `SortList` or `SortList2` with a `compareProc` of $00000001.

Tip 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.

Note Before you call `ListKey`, set the current QuickDraw port to the window containing your list control.

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` The list control 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 must 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.

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

Tip 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.

`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 the tip, at the start of this tool call description, about extended List controls.)

Chapter 11 **Media Control Tool Set**

This chapter documents the Media Control Tool Set. This is new material, not published in Volumes 1 to 3 of the *Apple IIGS Toolbox Reference*.

About the Media Control Tool Set

The Media Control Tool Set is a collection of routines that provide a consistent interface for controlling multimedia devices.

History

The initial motivation for designing this tool set was to port two multimedia toolkits (the HyperCard Videodisc Toolkit and HyperCard Audio-CD Toolkit) that were designed for the Macintosh. Since they were both written in high-level languages and provided control of multimedia devices for HyperCard, porting them over without deviating from the initial commands and implementation technique (that is, XCMDS) seemed in order.

Unfortunately, that approach covers only part of the total needs of developers and users. XCMDs only work with HyperCard applications; desk accessories and applications are left to reinvent the wheel (so to speak) if they want to provide the same capability.

Another problem that occurs is that the Videodisc Toolkit and the Audio-CD Toolkit are totally independent pieces of code with different calling sequences, yet, they provide many of the same capabilities. A better approach would be to combine the two toolkits into one.

This commonality of capabilities between the Audio-CD and Videodisc toolkits suggested that all multimedia devices provide a useful common set of similar controlling features (that is play, stop, pause, scan forward, scan backwards, etc.). Using only this small common set you can effectively command all kinds of media devices, such as a laserdisc, VCR, camcorder, CD, audio tape recorder, slide projector, digitized sounds, MIDI sequences, and so forth. The fact that all of these devices need roughly the same controlling features is the basis for this tool set.

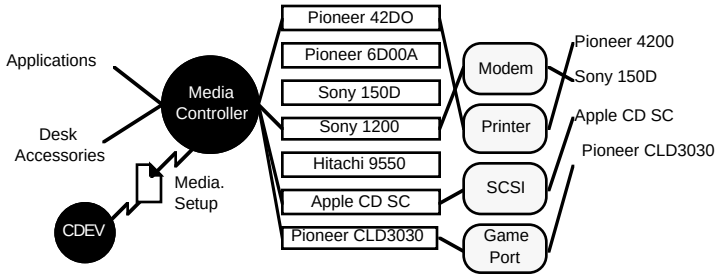
The primary goal for this tool set is to provide a standard, consistent interface for multimedia devices. This consistent interface not only allows developers to produce HyperCard IIGS stacks with multimedia capability, but also makes it easier to develop application programs and desk accessories.

Another goal of this tool set is to provide HyperCard IIGS with the same capability and commands as the Laserdisc and Audio-CD toolkits that were designed for the Macintosh. This tool set is designed so HyperCard IIGS and HyperCard for the Macintosh can use identical HyperTalk XCMDs to control the various multimedia devices.

System Overview

The Media Control software is made up of these parts:

CDEV	A Media Control control panel.
Media.Setup	A file created by the Media Control control panel that describes current configuration parameters (user selections made via the Media Control control panel).
Tool Set	The actual tool set code.
Device Drivers	Drivers tailored for specific media devices.



Media Control Control Panel and Media Channels

The Media Control control panel allows the user to specify the configuration he wants, then writes the selected configuration to a file named "Media.Setup." The user picks a device type (that is Pioneer 6000A, Sony 1500, Apple CD SC, etc.) and the port over which the device is connected (that is modem port, printer port, SCSI, etc.). This connection or pathway is called a **`media_channel`**. The tool set has the capability to open more than one `media_channel` at one time, so the user must also choose a `media_channel` number for the media path. The number of media channels that can be open at one time is currently limited to 8.

An example of the media channels in the above diagram would be:

1. Pioneer 4200 driver connected to printer port.
2. Sony 1200 driver connected to modem port.
3. Apple CD SC driver connected to a SCSI port.

Media.Setup File

This file contains the current media configuration as specified by the user by means of the Media Control control panel. The Media.Setup file resides in *:System:Drivers:Media.Control:Media.Setup. If the Media Control control panel or tool set doesn't find a Media.Setup file there, it looks for one on a local volume; this supports network booting.

On network booted systems, If a diskless user boots strictly from the file server, the Media Control control panel and tool set looks in the server volume for the Media.Setup file. If the user has a disk drive and has the minimal system disk to boot from, the Media Control control panel and tool set looks on the minimal system disk volume for the Media.Setup file.

The structure of the Media.Setup file is as follows:

$0000	*signature*	**Word**—File signature bytes
$0002	*device1*	**33 bytes**—Device connected to media channel 1
$0023	*device2*	**33 bytes**—Device connected to media channel 2
$0044	*device3*	**33 bytes**—Device connected to media channel 3
$0065	*device4*	**33 bytes**—Device connected to media channel 4
$0086	*device5*	**33 bytes**—Device connected to media channel 5
$00A7	*device6*	**33 bytes**—Device connected to media channel 6
$00C8	*device7*	**33 bytes**—Device connected to media channel 7
$00E9	*device8*	**33 bytes**—Device connected to media channel 8
$010A	*port1*	**33 bytes**—Port connected to media channel 1
$012B	*port2*	**33 bytes**—Port connected to media channel 2
$014C	*port3*	**33 bytes**—Port connected to media channel 3
$016D	*port4*	**33 bytes**—Port connected to media channel 4
$018E	*port5*	**33 bytes**—Port connected to media channel 5
$01AF	*port6*	**33 bytes**—Port connected to media channel 6
$01D0	*port7*	**33 bytes**—Port connected to media channel 7
$01F1	*port8*	**33 bytes**—Port connected to media channel 8

The device and port connections are Pascal strings. If a connection has been setup by the Media Control control panel, the 33 byte space will hold the device or port's ASCII Pascal string name. If no connection was made, the 33 byte reserved space contains a null Pascal string.

Tool Set

Applications make calls through the tool set to control the media devices. The tool set actually handles few of the commands, since most calls are handed off to the selected device driver, where the real work is performed. The application calls the tool set, which in turn makes appropriate calls to the device driver, which in turn makes appropriate GS/OS calls to the device. This technique is analogous to the way printing is handled. The tool set provides the consistent interface to the user while each of the device drivers worry about the specific characteristics of the device and how to perform the tool set commands.

Device Drivers

The device drivers are where most of the actual work is performed. Each device driver is tailored specifically for the device that it is to control. The device driver knows and understands the

particular command sequences that control the device and communicates them through the port driver.

The following device drivers are supplied with System 6:

Pioneer 4200	Through Modem or Printer port.
Apple CDSC	SCSI device.
Pioneer 2000	Through game port.

Port Drivers

Port drivers are GS/OS device drivers, and calls to the device are made through standard GS/OS calls.

File Locations

The media control device drivers, Media.Setup file, and the CD Remote resource files are located on the boot-up volume in “*:System:Drivers:Media.Control:”.

CD Remote Resource File Format

The CD Remote resource files are a database that is designed for compatibility with the CD Remote desk accessory data base on the Macintosh. The information is stored differently on the Apple IIGS; however, the data content is the same. Calls in the tool set are provided to retrieve and store information and can be accessed by Apple IIGS desk accessories, applications, and HyperCard IIGS XCMDs.

The following sample shows how the file is currently set up, but it is provided for your information only. Tool calls should always be used to access the file.

```
resource rPstring (1) {"1.0d1,Media Ctrl,1990"};
resource rPstring (2) {"Fleetwood Mac: Tango In The Night"};
resource rPstring (101) {"Big Love"};
resource rPstring (102) {"Seven Wonders"};
resource rPstring (103) {"Everywhere"};
resource rPstring (104) {"Caroline"};
resource rPstring (105) {"Tango In The Night"};
resource rPstring (106) {"Mystefied"};
resource rPstring (107) {"Little Lies"};
resource rPstring (108) {"Family Man"};
resource rPstring (109) {"Welcome To The Room...Sara"};
resource rPstring (110) {"Isn't It Midnight"};
resource rPstring (111) {"When I See You Again"};
resource rPstring (112) {"You and I, Part II"};

resource rCstring (1)

{"01,01,01,02,01,03,01,04,01,05,01,06,01,07,01,08,01,09,01,10,01,11,01,12"}
;
```

Legend:

- `rPstring (1)` is a comma separated Pascal string with the following entries:
  Version number.
  Media Control signature “Media Ctrl”.
  Year.
- `rPstring (2)` is the disc’s title.
- `rPstring` (100+track no.) is the track title (for example, `rPstring` (105) is track title string for track 5.)
- `rCstring (1)` is the disc’s program string.

CD Remote File Names

Many of the tool calls read or write CD Remote files. Instead of passing a specific file name, programs using these tool calls pass a unique 32 bit ID, and the Media Control Tool Set creates a file name from the ID value. The calls that use this mechanism for converting an ID into a file name include `MCSetDiscTitle`, `MCGetDiscTitle`, `MCSetTrackTitle`, `MCGetTrackTitle`, `MCSetProgram`, and `MCGetProgram`. This section describes how the ID is converted into a file name.

The 32 bit ID itself is formed by `MCGetDiscID`. See the description of `MCGetDiscID` for details on how the ID is formed and what limitations exist.

The ID returned by `MCGetDiscID` consists of two parts. The first four bits are a disk type identifier, while the remaining bits identify a particular disk of the given type.

Unique-ID = $XYYYYYYY

X (bits 31-28)= Disk type identifier bits.
YYYYYYY (bits 27-0)=Unique number.

The file name is formed by converting the disk type to a two letter code. This is followed by a period and seven characters formed by converting the last 28 bits of the ID to hexadecimal digits.

The two letter codes are:

value	letters	use
$0	CD	Compact Discs.
$1	LD	Laser Discs.
$2	VD	Video Device (VCR, Camcorder, etc.).
$3-$F	M3-MF	Media Device (generic, applications can use as they wish).

Examples:

Unique ID	File name
$00123456	CD.0123456
$29876543	VD.9876543
$789ABCDEF	M7.89ABCDEF

Tool Set Dependencies

The Media Control Tool Set requires the following tools to be loaded and started:

tool number	tool name
$01	Tool Locator
$02	Memory Manager
$03	Miscellaneous Tool Set
$0B	Integer Math Tool Set
$1E	Resource Manager

Typical Applications

A True Multimedia Control NDA

The Media Control NDA is one existing example that shows the Media Control Tool Set in action. It has a simple user interface incorporating a basic set of buttons (play, stop, scan, step, etc.) and the ability to switch between media channels.

Multimedia Sequence Editor, Scheduler

The capabilities that the Media Control Tool Set provides, and the consistent interface for controlling multimedia devices, allows for designing a multimedia sequencing editor. The multimedia sequencing editor would finally allow the user the capability to control a whole host of media devices from one application. This multimedia sequencing editor could be icon based and would allow the user to manipulate sequences of media on a time line.

Media Control Tool Set Housekeeping Calls

MCBootInit $0126

`MCBootInit` initializes the Media Control Tool Set. This function is called by the Tool Locator when the tool is loaded. An application should never make this call.

Parameters

The stack is not affected by this call.

Errors None.

C

```
extern pascal void MCBootInit()
```

MCStartUp $0226

`MCStartUp` is called by the application to start the Media Control Tool Set. This call must be made by the application before any other calls are made to the Media Control Tool Set.

Parameters

Stack before call

Previous contents	
userID	**Word**—user ID
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $2610 `mcWasStarted` The tool was already started.

GS/OS errors, Memory Manager errors, Miscellaneous Tool Set errors, and Integer Math Tool Set errors are returned unchanged.

C

```
extern pascal void MCStartUp(userID);
Word userID;
```

`userID` Memory Manager User ID for the application starting the Media Control Tool Set.

MCShutDown $0326

`MCShutDown` shuts down the Media Control Tool Set. Any application that starts the Media Control Tool Set with a call to `MCStartUp` must make this call to shut the tool set down. The call should be made after all other calls to the Media Control Tool Set.

Parameters

The stack is not affected by this call.

Errors $260F `mcWasShutDown` The tool set was already shut down.

GS/OS errors, Memory Manager errors, Miscellaneous Tool Set errors, and Integer Math Tool Set errors are returned unchanged.

C

```
extern pascal void MCShutDown();
```

MCVersion $0426

`MCVersion` returns the version number for the Media Control Tool Set.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
	<—SP

Stack after call

Previous contents	
versionInfo	**Word**—Version number
	<—SP

Errors Memory Manager errors are returned unchanged.

C

```
extern pascal Word MCVersion();
```

`versionInfo` Media Control Tool Set version number.

MCReset $0526

`MCReset` resets the Media Control Tool Set. An application should never make this call.

Parameters

The stack is not affected by this call.

Errors None.

C

```
extern pascal void MCReset();
```

MCStatus $0626

`MCStatus` indicates whether the Media Control Tool Set is active.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
	<—SP

Stack after call

Previous contents	
activeFlag	**Word**—Tool status
	<—SP

Errors None.

C

```
extern pascal Word MCStatus();
```

`activeFlag` The value returned is `TRUE` (non-zero) if the Media Control Tool Set is active, and `FALSE` ($0000) if it is not active.

MCBinToTime $0D26

`MCBinToTime` converts a binary value to its equivalent BCD time value.

Parameters

Stack before call

Previous contents	
— *Space* —	**Long**—Space for result
— *mcBinVal* —	**Long**—Binary value to convert
	<—SP

Stack after call

Previous contents	
— *result* —	**Long**—BCD time
	<—SP

Errors Integer Math Tool Set errors are returned unchanged.

C

```
extern pascal Long MCBinToTime(mcBinVal);
Long mcBinVal;
```

`mcBinVal` Binary value to convert to BCD format.

`result` Time in BCD format:

Bits 31-24	$00 to $99	BCD hours.
Bits 23-16	$00 to $59	BCD Minutes.
Bits 15-8	$00 to $59	BCD Seconds.
Bits 7-0	$00 to $74	BCD Partial Seconds.

A partial second is 1/75th of a second; this is the time unit used on CDs.

MCControl $1B26

`MCControl` is used to send a command to a device driver.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Channel number
ctlCommand	**Word**—Control command
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2607	`mcDevRtnError`	Device returned error (cannot perform the command).
$2608	`mcUnRecStatus`	Unrecognized status from the device.
$2609	`mcBadSelector`	Invalid selector value specified.
$260A	`mcFunnyData`	Funny data received (try again).
$260B	`mcInvalidPort`	Invalid port specified.
$2611	`mcBadChannel`	An invalid media channel was specified.

GS/OS errors, Memory Manager errors, Miscellaneous Tool Set errors, and Integer Math Tool Set errors are returned unchanged.

C

```
extern pascal void MCChannel(mcChannelNo, ctlCommand);
Word mcChannelNo, ctlCommand;
```

`mcChannelNo` Channel number for the device to send the command to.

`ctlCommand` Control command. See “Media Control Tool Set Constants” at the end of this chapter for a complete list of the commands.

MCDShutDown $1526

`MCDShutDown` shuts down a device driver. This call is normally made by the Media Control Tool Set, not by an application.

One unusual reason to make this call from an application is to prepare for an `MCUnloadDriver` call. See the description of `MCUnloadDriver` for details.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Passed channel number
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Memory Manager errors are returned unchanged.

C

```
extern pascal void MCDShutDown(mcChannelNo);
Word mcChannelNo;
```

`mcChannelNo` Channel number for the driver to shut down.

MCDStartUp $1426

`MCDStartUp` starts up a driver. This call is normally made by the Media Control Tool Set, not by the application.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Passed channel number
— *portNamePtr* —	**Long**—String pointer to connected port's name
userID	**Word**—Driver's user ID
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$260B	`mcInvalidPort`	An invalid port was specified.
$2611	`mcBadChannel`	An invalid media channel was specified.

GS/OS errors, Memory Manager errors, Miscellaneous Tool Set errors, Integer Math Tool Set errors, and Resource Manager errors are returned unchanged.

C

```
extern pascal void MCDStartUp(mcChannelNo, portNamePtr, userID);
Word mcChannelNo;
Ptr portNamePtr;
Word userID;
```

`mcChannelNo` Channel number for the driver to start.

`portNamePtr` Pointer to the name of the port to connect to. The name is in Pascal string format. An example of a name is "`.APPLESCSI.CDROM01.00`". In general, the format of the name depends on the driver being used for that channel.

`userID` Device driver's user ID.

MCGetDiscID $2826

`MCGetDiscID` returns a unique ID for the currently running disc. For a CD, this value is the serial number if one is available, and the total number of blocks if a serial number is not available. While it is possible for this ID to be the same for two CDs with no serial numbers, it is unlikely that two CDs will be the same length to 1/75th of a second.

This ID is normally used for calls to `MCGetDiscTitle`, `MCSetDiscTitle`, `MCGetTrackTitle`, `MCSetTrackTitle`, `MCGetProgram` and `MCSetProgram`.

Note This call can take a significant amount of time on some devices or volumes.

Parameters

Stack before call

Previous contents	
— *Space* —	**Long**—Space for result
mcChannelNo	**Word**—Passed channel number
	<—SP

Stack after call

Previous contents	
— *result* —	**Long**—Unique disc ID
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal Long MCGetDiscID(mcChannelNo);
Word mcChannelNo;
```

`mcChannelNo` Channel number.

`result` Unique ID; returned as a BCD time value with the same format as the time returned by `MCBinToTime`.

MCGetDiscTitle $1226

`MCGetDiscTitle` returns the title of the disc, as recorded in the CD Remote file. See the introductory information at the start of this chapter for more information about the CD Remote file.

Tip A file not found error is returned if there is no file for the given disc ID. You can use this fact to check to see if there are any entries for the disc.

Parameters

Stack before call

Previous contents	
mcDiscID	**Long**—Passed disc ID
pStrPtr	**Long**—Passed result Pascal string pointer
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$260E	`mcItemNotThere`	Item not found in CD Remote database.
$0046	`fileNotFound`	File not found.

C

```
extern pascal void MCGetDiscTitle(mcDiscID, pStrPtr);
Long mcDiskID;
Ptr pStrPtr;
```

`mcDiscID` Unique disc or media identifier; see `MCGetDiscID`.

`pStrPtr` Pointer to a 256 byte buffer where the title of the disc will be returned as a Pascal string.

MCGetDiscTOC $2726

`MCGetDiscTOC` returns track information that can be used to create a table of contents. The application passes a track number, and `MCGetDiscTOC` returns its starting time address.

Parameters

Stack before call

Previous contents	
— *Space* —	**Long**—Space for result
mcChannelNo	**Word**—Channel number
mcTrackNo	**Word**—Track number
	<—SP

Stack after call

Previous contents	
— *result* —	**Long**—BCD time
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal Long MCGetDiscTOC(mcChannelNo, mcTrackNo);
Word mcChannelNo, mcTrackNo;
```

`mcChannelNo` Channel number.

`mcTrackNo` Track number.

`result` Time address for the track, given in the BCD time format, as documented for the `MCBinToTime` call.

MCGetErrorMsg $0926

`MCGetErrorMsg` returns a text description of any Media Control Tool Set error code.

Parameters

Stack before call

Previous contents	
mcErrorNo	**Word**—Passed Media Control Tool Set error
— *pStrPtr* —	**Long**—Passed Pascal string pointer
	<—SP

Stack after call

Previous contents	
	<—SP

Errors If no error message exists for the error passed, the error number is returned as the error for this call.

C

```
extern pascal void (mcErrorNo, pStrPtr);
Word mcErrorNo;
Ptr pStrPtr;
```

`mcErrorNo` Any Media Control Tool Set error code.

`pStrPtr` Pointer to a 256 byte buffer. A text error message is placed in this buffer in Pascal string format.

MCGetFeatures $1626

`MCGetFeatures` returns information about the features supported by a given device. The application passes a feature number, and `MCGetFeatures` returns information about that feature.

Parameters

Stack before call

Previous contents	
— *Space* —	**Long**—Space for result
mcChannelNo	**Word**—Channel number
mcFeatSel	**Word**—Feature selector value
	<—SP

Stack after call

Previous contents	
— *result* —	**Long**—Result
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal Long MCGetFeatures(mcChannelNo, mcFeatSel);
Word mcChannelNo, mcFeatSel;
```

`mcChannelNo` Channel number.

`mcFeatSel` An integer identifying which feature to select. Information about the selected feature is returned by `MCGetFeatures`.

The table below shows the values returned for any particular value for `mcFeatSel`. The numeric values for `mcFeatSel` are given at the end of this chapter in the section "Media Control Tool Set Constants."

`mcFeatSet`	returned
`mcFTypes`	This value is a bit map. If bit 0 ($0001) is set, the device does `InChapters`. If bit 1 ($0002) is set, the device does `InFrames`. If bit 2 ($0004) is set, the device does `InTimes`.
`mcFStep`	Maximum frames per second value; normally 255. A value of 0 indicates that `step` is not supported by this device.
`mcFRecord`	Returns 1 if the device supports `MCRecord` and 0 if it does not.
`mcFVideo`	Returns 1 if the device supports toggling of video, and 0 if it does not.

`mcFEject`	Returns 1 if the device supports ejecting the media, and 0 if it does not.
`mcFLock`	Returns 1 if the device supports user lock, and 0 if not. User lock prevents the user from physically operating the device, as in ejecting a disc.
`mcFVDisplay`	Returns 1 if the device supports video display of the location, and 0 if not.
`mcFVOverlay`	Returns the number of overlay characters supported by the device, or 0 if the device does not support overlay of characters.
`mcFVOChars`	Number of characters per line supported by overlay; 0 for devices that do not support overlay.
`mcFVolume`	Returns 1 if the device supports volume control, and 0 if not.

MCGetName $2D26

`MCGetName` returns the name and version for a device driver.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Channel number
— *pStrPtr* —	**Long**—Pointer to Pascal string buffer
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal void MCGetName(mcChannelNo, pStrPtr);
Word mcChannelNo;
Ptr pStrPtr;
```

`mcChannelNo` `MCGetName` returns information about the device connected to this channel.

`PStrPtr` Pointer to a 256 byte character buffer. `MCGetName` returns a Pascal string in this buffer; the string has this format:

- The characters “MCToolkit”.
- The device driver’s short name, followed by its version number. If no device is connected to the channel, this part of the string will be “NoPlayer”.
- Connected port name.

If an error occurs, no output string is returned.

Examples:

“MCToolkit Pioneer4200 1.1 MODEM”
“MCToolkit Pioneer4200 1.2d3 GAME PORT”
“MCToolkit NoPlayer”

MCGetNoTracks $2926

`MCGetNoTracks` returns the number of tracks (CD disc) or chapters (laser disc) for the currently running media.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
mcChannelNo	**Word**—Channel number
	<—SP

Stack after call

Previous contents	
result	**Word**—Number of tracks or chapters
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal Word MCGetNoTracks(mcChannelNo);
Word mcChannelNo;
```

`mcChannelNo` Channel number.

`result` Number of tracks or chapters.

MCGetPosition $2426

`MCGetPosition` returns the current position for a device.

Parameters

Stack before call

Previous contents	
— *Space* —	**Long**—Space for result
mcChannelNo	**Word**—Channel number
mcUnitType	**Word**—unit type requested for results
	<—SP

Stack after call

Previous contents	
— *result* —	**Long**—Position
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal Long MCGetPosition(mcChannelNo, mcUnitType);
Word mcChannelNo, mcUnitType;
```

`mcChannelNo` Channel number.

`mcUnitType` Type of units to use for the result value. This can be any of:

`mcInChapters`	1	Return the position in terms of chapters (LaserDisc) or tracks (CD disc). The value returned is a long integer.
`mcInFrames`	2	Return the position in terms of frames of video. The value returned is a long integer.
`mcInTime`	3	Return the position in terms of elapsed time. For this return value, the format is a BCD time long word, as documented for the `MCBinToTime` call.

`result` Current position in the units requested via the `mcUnitType` parameter.

MCGetProgram $1026

`MCGetProgram` returns the program for a particular disc. The program determines which tracks, if any, will be played from a particular disc.

Parameters

Stack before call

Previous contents	
mcDiscID	**Long**—Disc ID
resultPtr	**Long**—GS/OS string result pointer
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcItemNotThere`	Item not found in CD Remote database.
$004F	`buffTooSmall`	Buffer too small.

C

```
extern pascal void MCGetProgram(mcDiscID, resultPtr);
Long mcDiscID;
Ptr resultPtr;
```

`mcDiscID` Unique disc or media identifier; see `MCGetDiscID`.

`resultPtr` Pointer to a GS/OS string result buffer.

GS/OS strings consist of a buffer length word for the entire string buffer (including the two words at the start of the buffer), a length word that is set to the current string length, and ASCII characters. The buffer length word is set by the caller before the call to `MCGetProgram`.

The program is returned as a comma delimited string. There are two entries for each track or chapter on a disc; the number of tracks or chapters can be determined using `MCGetTimes`. For each track or chapter, the first entry is the track or chapter number in the sequence specified in the CD Remote file, while the second entry is a 1 if the track or chapter should be played, and a zero if not.

A typical returned string would be:

"01,01,02,01,03,01,04,01"

MCGetSpeeds $1D26

`MCGetSpeeds` returns a list of the available speeds supported by a player.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Channel number
— *pStrPtr* —	**Long**—Pointer to Pascal string buffer
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal void MCGetSpeeds(mcChannelNo, pStrPtr);
Word mcChannelNo;
Ptr pStrPtr;
```

`mcChannelNo` Channel number.

`pStrPtr` Pointer to a 256 byte Pascal string buffer. The list of speeds is returned as a comma delimited ASCII string, with the speeds given in frames per second.

MCGetStatus $1A26

`MCGetStatus` returns the status of a device. Several different kinds of status information are available; the kind of information desired is specified via the `mcStatusSel` parameter.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
mcChannelNo	**Word**—Channel number
mcStatusSel	**Word**—Status selector value
	<—SP

Stack after call

Previous contents	
result	**Word**—Status; see description
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal Word MCGetStatus(mcChannelNo, mcStatusSel);
Word mcChannelNo, mcStatusSel;
```

`mcChannelNo` Channel number.

`mcStatusSel` Status selector. This value specifies which kind of status information to display. The table below shows the possible return values for each status selector. The numeric values for the constants used in the table are shown at the end of this chapter, in the section "Media Control Tool Set Constants."

`mcStatusSel`	returned	notes
`mcSDeviceType`	`mcSLaserDisc`	The device is a laserdisc player.
	`mcSCDAudio`	The device is an audio CD player.
	`mcSLaserCD`	The device is a combined laserdisc and CD player.
	`mcSVCR`	The device is a VCR.
	`mcSCamCorder`	The device is a camcorder.
	`mcSVMonitor`	The device is a video monitor.
`mcSPlayStatus`	`mcSPlaying`	The device is currently playing.
	`mcSStill`	The device is still (not doing anything).
	`mcSParked`	The device is parked or ejected.
	`mcSUnknown`	The play status can't be determined.
`mcSDoorStatus`	`mcSDoorOpen`	The door for the device is open. For example, the CD tray that holds the CD is out.
	`mcSDoorClosed`	The door for the device is closed.

	`mcSUnknown`	The position of the door can't be determined.
`mcSDiscType`	`mcS_CLV`	A constant linear velocity (CLV) disc is in the device.
	`mcS_CAV`	A constant angular velocity (CAV) disc is in the device.
	`mcS_CDV`	A compact disc video (CDV) disc is in the device.
	`mcS_CD`	A compact disc (CD) disc is in the device.
	`mcSUnknown`	The type of disc can't be determined.
`mcSDiscSize`	`mcSDisc3inch`	A 3 inch disc is in the device.
	`mcSDisc5inch`	A 5 inch disc is in the device.
	`mcSDisc8inch`	A 8 inch disc is in the device.
	`mcSDisc12inch`	A 12 inch disc is in the device.
	`mcSUnknown`	The size of the disc can't be determined.
`mcSDiscSide`	`mcSSideOne`	Playing side one.
	`mcSSideTwo`	Playing side two.
	`mcSUnknown`	The disc side can't be determined.

`result` This is the returned value. The possible return values are listed in the table above.

MCGetTimes $2626

`MCGetTimes` returns information about a disc. As the name of the call implies, the information is time related, although not all of the information returned is a time in the traditional sense. The meaning of the returned value is determined by the `mcTimesSel` parameter.

Parameters

Stack before call

Previous contents	
— *Space* —	**Long**—Space for result
mcChannelNo	**Word**—Channel number
mcTimesSel	**Word**—Info selector value
	<—SP

Stack after call

Previous contents	
— *result* —	**Long**—BCD time
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal Long MCGetTimes(mcChannelNo, mcTimesSel);
Word mcChannelNo, mcStatusSel;
```

`mcChannelNo` Channel number.

`mcTimesSel` This parameter selects one of the various times that the call can return. The table below shows what time is returned for each possible time selector. The numeric values for the constants used in the table are shown at the end of this chapter, in the section "Media Control Tool Set Constants."

`mcTimesSel`	Meaning of `result`
`mcElapsedTrack`	Elapsed time for the currently playing track (CD) or chapter (video disc).
`mcRemainTrack`	Remaining time for the currently playing track (CD) or chapter (video disc).
`mcElapsedDisc`	Elapsed time for the currently playing disc.
`mcRemainDisc`	Remaining time for the currently playing disc.
`mcTotalDisc`	The total run time for the disc.
`mcTotalFrames`	The total number of frames on the disc. The number of frames is returned as a BCD value.
`mcTracks`	The start and end track numbers for the disc. The most significant word of the long word result contains the ending track number, while the least significant word holds the starting track number.

`result` The result is a long word. The meaning is determined by the `mcTimesSel` parameter, as described in the table above. Unless otherwise noted, the results are times, returned in BCD format, as described for the `MCBinToTime` call.

MCGetTrackTitle $0E26

`MCGetTrackTitle` reads the CD Remote database file, returning a track title from that file.

Parameters

Stack before call

Previous contents	
— *mcDiscId* —	**Long**—Disc ID
mcTrackNo	**Word**—Track number
— *pStrPtr* —	**Long**—Pointer Pascal string buffer
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$260E	`mcItemNotThere`	Item not found in CD Remote database.
$0046	`fileNotFound`	File not found.

C

```
extern pascal Word MCGetTrackTitle(mcDiscID, mcTrackNo, pStrPtr);
Long mcDiscID;
Word mcTrackNo, pStrPtr;
```

`mcDiscID` Unique disc or media identifier. See `MCGetDiscID`.

`mcTrackNo` `MCGetTrackTitle` returns the title of the track with this track number.

`pStrPtr` This pointer points to a Pascal string buffer where the track name is stored. If there is an error or if the disc isn't in the CD Remote file database, this string is set to the null string. The buffer should be 256 characters long to allow for the longest possible Pascal string.

MCJog $2026

`MCJog` moves either forward or backward on a disc.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Channel number
mcUnitType	**Word**—Unit type
— *mcNJog* —	**Long**—Number of units to jog
mcJogRepeat	**Word**—Times to repeat
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal void MCJog(mcChannelNo, mcUnitType, mcNJog,
   mcJogRepeat);
Word mcChannelNo, mcUnitType;
Long mcNJog;
Word mcJogRepeat;
```

`mcChannelNo` Channel number.

`mcUnitType` This is the type of units to use for the move. For example, you can move forward a specific number of frames, or, if you prefer, move forward a specific amount of time.

The numeric values for the constants used in the table are shown at the end of this chapter, in the section "Media Control Tool Set Constants."

mcUnitType	Use
`mcInChapters`	The units are given in chapters (laserdisc) or tracks (CD). The value itself is a long integer.
`mcInFrames`	The units are given in frames. This unit is used for video sources, such as VHS tape, video discs, and so forth. The value itself is a long integer.
`mcInTime`	The units are given as a time in BCD format. See `MCBinToTime` for a breakdown of the format for the time.

`mcNJog` This is the number of units to jog forward or backward.

`mcJogRepeat` This is a repeat count. It can be positive or negative. The media will skip `mcNJog*mcJogRepeat` times, skipping `mcUnitType` units each time.

MCLoadDriver $0A26

The Media Control Tool Set makes this call to load a driver into memory. This call is normally not used by an application.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Passed channel number
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $2611 `badChannel` An invalid media control channel was passed.

GS/OS errors, Memory Manager errors, Miscellaneous Tool Set errors, and Integer Math Tool Set errors are returned unchanged.

C

```
extern pascal void MCLoadDriver(mcChannelNo);
Word mcChannelNo;
```

`mcChannelNo` Channel number for the driver to load.

MCPause $1826

If you are playing a device, `MCPause` puts it in pause mode. `MCPlay` will start playing from the device, again.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Passed channel number
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal Word MCPause(mcChannelNo);
Word mcChannelNo;
```

`mcChannelNo` Channel number.

`MCPlay` $1726

`MCPlay` starts the device at the normal playing speed.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Passed channel number
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal void MCPlay(mcChannelNo);
Word mcChannelNo;
```

`mcChannelNo` Channel number.

MCRecord $2A26

If the device supports recording, `MCRecord` will tell the device to start recording. An error is returned if the device does not support recording.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Passed channel number
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal void MCRecord(mcChannelNo);
Word mcChannelNo;
```

`mcChannelNo` Channel number.

MCScan $1C26

`MCScan` gives a device independent way to scan forward or backward. The scan will continue until it can't go any farther (for example, when the end of a tape is reached) or until some other command is used, like `MCPlay`, `MCPause`, `MCRecord`, or `MCStop`.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Channel number
mcDirection	**Word**—Scan direction
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal void MCScan(mcChannelNo, mcDirection);
Word mcChannelNo, mcDirection;
```

`mcChannelNo` Channel number.

`mcDirection` This is the scan direction. Any positive value scans forward, while any negative value scans backward.

MCSearchDone $2226

MCSearchDone returns a status value indicating whether a previous MCSearchTo command has completed.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
mcChannelNo	**Word**—Channel number
	<—SP

Stack after call

Previous contents	
result	**Word**—TRUE if search complete, else FALSE
	<—SP

Errors

$2605	mcNotLoaded	No driver is currently loaded.
$2611	mcBadChannel	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal Word MCSearchDone(mcChannelNo);
Word mcChannelNo;
```

mcChannelNo Channel number.

result This is a boolean value. It will be TRUE (1) if the search point has been reached, and FALSE (0) if the search point has not been reached.

Note MCSearchDone only returns TRUE once per search.

MCSearchTo $2126

`MCSearchTo` starts a search to the location specified by `searchloc`.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Channel number
mcUnitType	**Word**—Unit type
— *searchLoc* —	**Long**—Location to search to
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal void MCSearchDone(mcChannelNo, mcUnitType,
   searchLoc);
Word mcChannelNo, mcUnitType;
Long searchLoc;
```

`mcChannelNo` Channel number.

`mcUnitType` This is the type of units to use for the search. For example, you can move forward to a specific frame, or, if you prefer, move forward to a specific time. The position is relative to the start of the media.

The numeric values for the constants used in the table are shown at the end of this chapter, in the section "Media Control Tool Set Constants."

mcUnitType	Use
`mcInChapters`	The units are given in chapters (laserdisc) or tracks (CD). The value itself is a long integer.
`mcInFrames`	The units are given in frames. This unit is used for video. The value itself is a long integer.
`mcInTime`	The units are given as a time in BCD time format. See `MCBinToTime` for a breakdown of the format for the time.

`searchLoc` This is the location you want to search to, given in `mcUnitType` units.

Tip After performing an `MCSearchTo` command, use an `MCSearchWait` call or a series of `MCSearchDone` calls to insure the search has completed before issuing another command.

MCSearchWait $2326

`MCSearchWait` is used after an `MCSearchTo` call. It waits until the search position has been reached, then returns.

Note This call should not be used unless an `MCSearchTo` call has just been issued. If no `MCSearchTo` call has been made, the `MCSearchWait` will not return until it decides a time-out error has occurred.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Passed channel number
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal void MCSearchWait(mcChannelNo);
Word mcChannelNo;
```

`mcChannelNo` Channel number.

MCSendRawData $1926

`MCSendRawData` sends a block of bytes to a device. This allows a program to send blocks of raw data to the device to make use of features of the device that are not supported directly by this tool set.

See also `MCWaitRawData`, which will read raw data from a device.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Channel number
— *mcNativePtr* —	**Long**—GS/OS string pointer to data to send to the device
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal void MCSendRawData(mcChannelNo, mcNativePtr);
Word mcChannelNo;
Long mcNativePtr;
```

`mcChannelNo` Channel number.

`mcNativePtr` Pointer to a GS/OS input string containing the raw data to send to the device.

MCSetAudio $2526

`MCSetAudio` controls the audio output for a device.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Channel number
mcAudioCtl	**Word**—Audio control value
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal void MCSetAudio(mcChannelNo, mcAudioCtl);
Word mcChannelNo, mcAudioCtl;
```

`mcChannelNo` Channel number.

`mcAudioCtl` This parameter tells the call what to do. The table below shows the various values you can pass and tells what effect the call will have on the device. In the notes, "channel of sound" refers to the sound data from a disc or other audio data source, and "speaker" refers to the output from the player, which would normally go to a speaker.

The numeric values for the constants used in the table are shown at the end of this chapter, in the section "Media Control Tool Set Constants."

mcAudioCtl	Use
`AudioOff`	Turn audio off.
`AudioStereo`	Play the left audio channel through the left speaker and the right audio channel through the right speaker. This is a normal, stereo mode.
`AudioMonaural`	Mix the left and right audio channels and play the mix through both channels. This is a normal monaural (non-stereo) playing mode.
`AudioReverse`	Play the left channel of sound through the right speaker, and the right channel of sound through the left speaker.
`AudioRight`	Play the right audio channel through the right speaker. The left audio channel is not played.
`AudioLeft`	Play the left audio channel through the left speaker. The right audio channel is not played.
`AudioLinR`	Play the left channel of sound only, but play it through the right speaker.

`AudioRinL`	Play the right channel of sound only, but play it through the left speaker.
`AudioMinR`	Mix the left and right channel of sound, playing them through the right speaker.
`AudioMinL`	Mix the left and right channel of sound, playing them through the left speaker.
`AudioRinLR`	Play the right channel of sound through both the left and right speaker.
`AudioLinLR`	Play the left channel of sound through both the left and right speaker.
`AudioRinLMR`	Play the right audio channel through the left speaker. Mix the left and right audio channel, playing the mix through the right speaker.
`AudioLinLMR`	Play the left audio channel through the left speaker. Mix the left and right audio channel, playing the mix through the right speaker.
`AudioLRinR`	Play the right channel of sound through the right speaker. Mix the left and right audio channels, playing them through the left speaker.
`AudioLLinR`	Play the left channel of sound through the right speaker. Mix the left and right audio channels, playing them through the left speaker.

MCSetDiscTitle $1326

`MCSetDiscTitle` sets the disc title in the CD Remote data file.

See the information at the start of this chapter for more information about the CD Remote file. See `MCGetDiscID` for information about unique disc IDs.

Note If a CD Remote data file doesn't already exist for the unique disc ID `mcDiscID`, `MCSetDiscTitle` will create a new data file.

Parameters

Stack before call

Previous contents	
— *mcDiscID* —	**Long**—Disc ID
— *titlePtr* —	**Long**—Pointer to Pascal string title
	<—SP

Stack after call

Previous contents	
	<—SP

Errors GS/OS and Resource Manager errors are returned unchanged.

C

```
extern pascal void MCSetDiscTitle(mcDiscID, titlePtr);
Long mcDiscID, titlePtr;
```

`mcDiscID` Unique disc or media identifier.

`titlePtr` Pointer to the title string. The title is given as a Pascal string.

MCSetProgram $1126

`MCSetProgram` sets the program string in a CD Remote database file.

The program string is a comma separated list. There are two entries for each track or chapter. The first item is the track or chapter number in the sequence specified in the CD Remote database file. The second item is 1 if the track or chapter should be played, and 0 if the track or chapter should not be played. A typical program string would be “01,01,02,01,03,01,04,01”.

See the introductory information at the start of this chapter for more information about the CD Remote file. See `MCGetDiscID` for information about unique disc IDs.

Note If a CD Remote data file doesn’t already exist for the unique disc ID `mcDiscID`, `MCSetProgram` will create a new data file.

Parameters

Stack before call

Previous contents	
mcDiscID	**Long**—Unique disc ID
mcProgPtr	**Long**—GS/OS string pointer
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $2612 `mcInvalidParam` An invalid parameter was specified.

GS/OS and Resource Manager errors are returned unchanged.

C

```
extern pascal void MCSetProgram(mcDiscID, mcProgPtr);
Long mcDiscID, mcProgPtr;
```

`mcDiscID` Unique disc or media identifier.

`mcProgPtr` Pointer to a GS/OS input string containing the program string.

MCSetTrackTitle $0F26

`MCSetTrackTitle` sets the title for a track in the CD Remote data file.

See the introductory information at the start of this chapter for more information about the CD Remote file. See `MCGetDiscID` for information about unique disc IDs.

Note If a CD Remote data file doesn't already exist for the unique disc ID `mcDiscID`, `MCSetTrackTitle` will create a new data file.

Parameters

Stack before call

Previous contents	
mcDiscID	**Long**—Disc ID
trackNum	**Word**—Track number
titlePtr	**Long**—Pointer to Pascal string title
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $2612 `mcInvalidParam` An invalid parameter was specified.

GS/OS and Resource Manager errors are returned unchanged.

C

```
extern pascal void MCSetTrackTitle(mcDiscID, trackNum, titlePtr);
Long mcDiscID;
Word trackNum;
Long titlePtr;
```

`mcDiscID` Unique disc or media identifier.

`trackNum` Set the title for this track number.

`titlePtr` Pointer to the track title string. The title is given as a Pascal string.

MCSetVolume $2E26

`MCSetVolume` sets the volume level for a device.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Channel number
mcLeftVol	**Word**—Left volume level
mcRightVol	**Word**—Right volume level
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal void MCSetAudio(mcChannelNo, mcLeftVol, mcRightVol);
Word mcChannelNo, mcLeftVol, mcRightVol;
```

`mcChannelNo` Channel number.

`mcLeftVol` This is the volume for the left channel. The values can range from 0 for no sound to $FFFF for full volume.

`mcRightVol` This is the volume for the right channel. The values can range from 0 for no sound to $FFFF for full volume.

MCSpeed $1E26

MCSpeed is used to play a device at a specific speed. The speed is passed in frames per second; 30 frames per second is the normal playback speed.

Setting a speed with MCSpeed affects the next call to MCPlay. A subsequent call to MCPlay, with no other call to MCSpeed, switches the playing speed back to the default value of 30 frames per second.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Channel number
mcFPS	**Word**—Frames per second
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	mcNotLoaded	No driver is currently loaded.
$2611	mcBadChannel	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal void MCSpeed(mcChannelNo, mcFPS);
Word mcChannelNo, mcFPS;
```

mcChannelNo Channel number.

mcFPS Speed in frames per second.

MCStop $2B26

`MCStop` stops the device.

After an `MCStop` command some devices resume play at the current location and some resume at the start of the media.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Channel number
mcFPS	**Word**—Frames per second
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal void MCStop(mcChannelNo);
Word mcChannelNo;
```

`mcChannelNo` Channel number.

MCStopAt $1F26

`MCStopAt` sets a stop location. When the device reaches the location given, it stops playing. This call is normally used before the call to `MCPlay` to insure that the device stops correctly.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Channel number
mcUnitType	**Word**—Unit type
— *mcStopLoc* —	**Long**—Stop location
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal void MCStopAt(mcChannelNo, mcUnitType, mcStopLoc);
Word mcChannelNo, mcUnitType;
Long mcStopLoc;
```

`mcChannelNo` Channel number.

`mcUnitType` This is the type of units to use. For example, you can stop at a specific frame, or, if you prefer, stop at a specific time. The location and time values are relative to the start of the disc.

The numeric values for the constants used in the table are shown at the end of this chapter, in the section "Media Control Tool Set Constants."

mcUnitType	Use
`mcInChapters`	The units are given in chapters (laserdisc) or tracks (CD). The value itself is a long integer.
`mcInFrames`	The units are given in frames. This unit is used for video. The value itself is a long integer.
`mcInTime`	The units are given as a time in BCD time format. See `MCBinToTime` for a breakdown of the format for the time.

`mcStopLoc` This is the location the device will stop at, given in `mcUnitType` units.

MCTimeToBin $0C26

`MCTimeToBin` converts a BCD time value from hours, minutes, seconds, and frames to its binary equivalent.

Parameters

Stack before call

Previous contents	
— *Space* —	**Long**—Space for result
— *mcTimeValue* —	**Long**—Time value to convert
	<—SP

Stack after call

Previous contents	
— *result* —	**Long**—Binary time
	<—SP

Errors Integer Math Tool Set errors are returned unchanged.

C

```
extern pascal Long MCTimeToBin(mcTimeValue);
Long mcTimeValue;
```

`mcTimeValue` Time in BCD format:

Bits 31-24	$00 to $99	BCD hours.
Bits 23-16	$00 to $59	BCD Minutes.
Bits 15-8	$00 to $59	BCD Seconds.
Bits 7-0	$00 to $74	BCD Partial Seconds.

A partial second is 1/75th of a second; this is the time unit used on CDs.

`result` Time as a number of partial seconds. For example, `MCTimeToBin(0x01234567)` would return 0x12D687 (1234567 in decimal).

MCUnLoadDriver $0B26

`MCUnLoadDriver` unloads a driver from memory.

This call is normally made by the Media Control Tool Set, and not by the application. One unusual reason for an application to make this call is when the application has manually changed the Media.Setup file. After changing the file, `MCUnLoadDriver` can be called to force the driver to unload. The driver will then be loaded and reinitialized with the new parameters the next time it is needed.

Note Be sure to call `MCDShutDown` to force the device driver to shut down before making this call.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Passed channel number
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	`mcNotLoaded`	No driver is currently loaded.
$2611	`mcBadChannel`	An invalid media control channel was passed.

GS/OS errors, Memory Manager errors, Miscellaneous Tool Set errors, and Integer Math Tool Set errors are returned unchanged.

C

```
extern pascal void MCUnLoadDriver(mcChannelNo);
Word mcChannelNo;
```

`mcChannelNo` Channel number for the driver to unload.

MCWaitRawData $2C26

MCWaitRawData reads raw bytes from a device. It is used in conjunction with MCSendRawData to send and receive commands or information that are not supported by the other tool calls in this tool set.

Parameters

Stack before call

Previous contents	
mcChannelNo	**Word**—Channel number
— *resultPtr* —	**Long**—GS/OS string result pointer
tickWait	**Word**—Number of ticks before time-out error
termMask	**Word**—Terminal character and mask
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2605	mcNotLoaded	No driver is currently loaded.
$2611	mcBadChannel	An invalid media channel was specified.

Errors from the driver are returned unchanged.

C

```
extern pascal void MCWaitRawData(mcChannelNo, resultPtr, tickWait,
   termMask);
Word mcChannelNo;
Long resultPtr;
Word tickWait, termMask;
```

mcChannelNo Channel number for the driver to unload.

resultPtr Pointer to a GS/OS output string buffer. The raw data will be placed in this buffer.

tickWait The number of system ticks to occur before MCWaitRawData terminates with a time-out error. This is used to prevent system hangs waiting for characters from the device that may never occur. One system click is 1/60th of a second.

termMask This parameter is split into two one-byte values. The least significant byte (bits 7-0) are a mask character, while the most significant byte (bits 15-8) is a terminal character.

MCWaitRawData determines when to stop by masking the received characters with the passed mask and comparing it against the terminal character. When the two values are equal, MCWaitRawData stops. The raw data, including the stop byte, will be placed in the GS/OS output string pointed to by resultPtr.

If the passed mask character is zero and the terminal character is non-zero, this routine reads until the GS/OS output string buffer is filled. This allows for block transfers without looking for a terminal character.

If mask character and the terminal character are zero, `MCWaitRawData` will wait up to `tickwait` ticks for the device to make a character available. If a character is available, the single character is returned in a GS/OS output string. If no characters are available within the specified time, a null string (zero length) is returned. This capability can be used to poll the device for single bytes; you would generally pass 1 for `tickwait`.

Media Control Tool Set Summary

Media Control Tool Set Constants

Name	Value	Description
`InChapters`	1	Selector value for chapters
`InFrames`	2	Selector value for frames
`InTimes`	3	Selector value for times

Control values for `MCControl`

`mcCInit`	1	initialize player
`mcCEject`	2	eject disc
`mcCVideoOn`	3	turn video on
`mcCVideoOff`	4	turn video off
`mcCDisplayOn`	5	turn video position display off
`mcCDisplayOff`	6	turn video position display on
`mcCBlankVideo`	7	blank video for next `MCSearchTo`
`mcCDefaultCom`	8	set default communications
`mcCLockDev`	9	lock the device
`mcCUnLockDev`	10	unlock the device
`mcC8Data1Stop`	40	set 8-data 1-stop bit
`mcC7Data1Stop`	41	set 7-data 1-stop bit
`mcC6Data1Stop`	42	set 6-data 1-stop bit
`mcC5Data1Stop`	43	set 5-data 1-stop bit
`mcC8Data2Stop`	44	set 8-data 2-stop bit
`mcC7Data2Stop`	45	set 7-data 2-stop bit
`mcC6Data2Stop`	46	set 6-data 2-stop bit
`mcC5Data2Stop`	47	set 5-data 2-stop bit
`mcCBaudDflt`	50	set baud rate to control panel setting
`mcCBaud50`	51	set 50 baud
`mcCBaud75`	52	set 75 baud
`mcCBaud110`	53	set 110 baud
`mcCBaud134`	54	set 134.5 baud
`mcCBaud150`	55	set 150 baud
`mcCBaud300`	56	set 300 baud
`mcCBaud600`	57	set 600 baud
`mcCBaud1200`	58	set 1200 baud
`mcCBaud1800`	59	set 1800 baud
`mcCBaud2400`	60	set 2400 baud
`mcCBaud3600`	61	set 3600 baud
`mcCBaud4800`	62	set 4800 baud
`mcCBaud7200`	63	set 7200 baud
`mcCBaud9600`	64	set 9600 baud
`mcCBaud19200`	65	set 19200 baud
`mcCModem`	100	set to modem port
`mcCPrinter`	101	set to printer port

mcCIgnoreDS	200	ignore disk switched errors (They will not be reported at all.)
mcCReportDS	201	report disk switched errors.

Status values for MCGetFeatures

mcFTypes	0	Does frames, times and chapters
mcFStep	1	Maximum step value
mcFRecord	2	Does MCRecord function
mcFVideo	3	Does video
mcFEject	4	Does eject function
mcFLock	5	Does user key lock
mcFVDisplay	6	Does video location display
mcFVOverlay	7	Number of lines of character video overlay
mcFVOChars	8	Number of characters of video overlay
mcFVolume	9	Does volume control

Status values for MCGetStatus

mcSUnknown	0	player unable to determine this status
mcSDeviceType	$0000	device type selector
mcSLaserDisc	1	
mcSCDAudio	2	
mcSCDLaserCD	3	
mcSVCR	4	
mcSCamCorder	5	
mcSPlayStatus	$0001	play status selector value
mcSPlaying	1	
mcSStill	2	
mcSParked	3	
mcSDoorStatus	$0002	players door status
mcSDoorOpen	1	
mcSDoorClosed	2	
mcSDiscType	$0003	disc type selector value
mcS_CLV	1	
mcS_CAV	2	
mcS_CDV	3	
mcS_CD	4	
mcSDiscSize	$0004	disc size selector value
mcSDisc3inch	3	
mcSDisc5inch	5	
mcSDisc8inch	8	
mcSDisc12inch	12	
mcSDiscSide	$0005	disc side selector value
mcSSideOne	1	
mcSSideTwo	2	
mcSVolumeL	$0006	Current left volume selector
mcSVolumeR	$0007	Current right volume selector

MCGetTimes selector values

mcElapsedTrack	0	Elapsed time on current track/chapter
mcRemainTrack	1	Remaining time on current track/chapter

mcElapsedDisc	2	Elapsed time on disc
mcRemainDisc	3	Remaining time on disc
mcTotalDisc	4	Total run time on disc
mcTotalFrames	5	Returns total number of frames on disc
mcTracks	6	Returns the first and last track numbers
mcDiscID	7	Returns a disc identifier

Audio values

AudioOff	0	Audio off
AudioRight	1	Audio right channel only
AudioLinR	2	Audio left in right only
AudioMinR	3	Audio mixed in right only
AudioRinL	4	Audio right in left only
AudioRinLR	5	Audio right in left and right
AudioReverse	6	Audio right in left, left in right
AudioRinLMR	7	Audio right in left, mixed in right
AudioLeft	8	Audio left channel only
AudioStereo	9	Audio both channels (Stereo)
AudioLinLR	10	Audio left in left and right
AudioLinLMR	11	Audio left in left, mixed in right
AudioMinL	12	Audio mixed in left only
AudioMinLRinR	13	Audio mixed in left, right in right
AudioMinLLinR	14	Audio mixed in left, left in right
AudioMonaural	15	Audio mixed in left and right (monaural)

Code	Name	Description
$2601	mcUnImp	Unimplemented for this device
$2602	mcBadSpeed	Invalid speed specified
$2603	mcBadUnitType	Invalid unit type specified
$2604	mcTimeOutErr	Timed out during device read
$2605	mcNotLoaded	No driver is currently loaded
$2606	mcBadAudio	Invalid audio value
$2607	mcDevRtnError	Device returned error (unable to perform)
$2608	mcUnRecStatus	Unrecognizable status from device
$2609	mcBadSelector	Invalid selector value specified
$260A	mcFunnyData	Funny data receive (try again)
$260B	mcInvalidPort	Invalid port specified
$260C	mcOnlyOnce	Scans only once
$260D	mcNoResMgr	Resource Manager not active (must be loaded and started)
$260E	mcItemNotThere	Item not found in CD Remote database
$260F	mcWasShutDown	The tool was already shut down
$2610	mcWasStarted	The tool was already started
$2611	mcBadChannel	An invalid media channel number was specified
$2612	mcInvalidParam	An invalid parameter was specified
$2613	mcCallNotSupported	An invalid media control tool call was attempted

Chapter 12 **Memory Manager Update**

This chapter contains new information about the Memory Manager. The original reference to this tool set is in Volume 1, Chapter 12 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 36 of the *Apple IIGS Toolbox Reference*.

New Features

- System Software 6.0 fixes a problem where a long hang and then a crash could result if an Out-of-Memory-Queue routine freed up the requested 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).

- System Software 6.0 fixes 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.

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 $0000 for the `newID` parameter. The previous ID is always returned, whether the ID is changed or not.

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 converter 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 $5x00-range memory ID for the chunk of code.
6. Use `SetHandleID` to change the code's memory ID to the newly allocated one. Now, when the system disposes of all memory using the Control Panel's memory ID, the code will not be disposed.

Parameters

Stack before call

Previous contents	
space	**Word**—Space for old ID
newID	**Word**—New ID for handle (0 to leave it 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;
```

`newID` New user ID.

`theHandle` Handle to change.

Chapter 13 **Menu Manager Update**

This chapter contains new information about the Menu Manager. The original reference to this tool set is in Volume 1, Chapter 13 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 37 of the *Apple IIGS Toolbox Reference*.

New Features

- Pop-up menu controls now support `ctlMoreFlags` bit 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.

- When a menu item is blinking, the speed is now limited using `WaitUntil` in the Miscellaneous Tools. This way an accelerated machine does not blink the item faster than it should be blinked.

- `EnableMItem` and `DisableMItem` have been patched on ROM 3 to simulate a dispatcher error ($0001) when the Menu Manager has not been started up. This change was made for compatibility with a broken third party application. There was no incompatibility for ROM 1.

- `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`. The menu blink rate can be set from the General control panel.

- When `MenuKey` receives a key press 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` only changes the Scan Line Control Bytes (SCBs) for the scan lines from 0 to `MenuHeight-1`. It used to call `SetAllSCBs`, changing all of the SCBs.

Icons in Menu Items

The Menu Manager now supports icons in menu items, including pop-up menu items. Seven new calls implement this support: `SetMItemIcon`, `GetMItemIcon`, `SetMItemStruct`,

`GetMItemStruct`, `RemoveMItemStruct`, `SetMItemFlag2`, and `GetMItemFlag2`. A few old calls have been modified slightly and an additional menu item structure has been defined.

QuickDraw II Auxiliary must be started if icons are used in a menu. The Menu Manager does not require QuickDraw II Auxiliary if menu icons are not used.

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 *Apple IIGS Toolbox Reference: Volume 3* for more details on the structure of a menu item record and menu item 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 the attributes of this structure
$02	— *itemTitleRef* —	**Long**—Reference to item name
$06	— *itemIconRef* —	**Long**—Reference to icon associated with item

`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 additional 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 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 *Apple IIGS Toolbox Reference: Volume 3*.

△ **Important** If your `itemStruct` records are referenced as `rItemStruct` resources, the Menu Manager makes their handles purgeable after each use. △

If you use `SetMItemIcon`, `SetMItemName` or `SetMItemFlag2` to modify `itemStruct` structures specified as resources, the resources must be locked (with the `resLocked` attribute) or the structures may be purged after the Menu Manager releases them. If the structures are purged, they'll be reloaded from disk next time they're needed, and the copies on disk won't have your changes included. Δ

The following existing calls have been modified to work with the new `itemStruct` record. All these calls still perform as documented. Internally these calls have 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

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**—Menu item blink setting
	<—SP

Errors None.

C

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

GetMItemFlag2 $4C0F

`GetMItemFlag2` returns the `itemFlag2` field for the `itemStruct` record associated with the menu item indicated. If bit 10 is not set then the value returned is not valid.

See "Icons and Menu Items," earlier in this chapter, for a description of the `itemFlag2` field, which is a part of the new menu item record.

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

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` Returned if bit 10 of `itemFlag` is not set.

C

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

`itemID` Menu item number whose `itemFlag2` value will be returned.

GetMItemIcon $480F

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

See "Icons and Menu Items," earlier in this chapter, for a description of the `itemFlag` field, which is a part of the new menu item record.

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

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 None.

C

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

`itemID` Menu item number whose icon reference will be returned.

GetMItemStruct $4A0F

`GetMItemStruct` returns the reference to the `itemStruct` record of the menu item specified. If there is no structure – that is, if bit 10 of `itemFlag` is set to zero – then zero will be returned as the reference.

See "Icons and Menu Items," earlier in this chapter, for a description of the `itemStruct` record.

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

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 None.

C

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

`itemID` Menu item number for the menu item whose `itemStruct` reference will be returned.

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
— *pathPtr* —	**Long**—Pointer to GS/OS input pathname
deviceNum	**Word**—Device number the path is on, if known
menuID	**Word**—Menu ID of the menu to insert into
afterID	**Word**—Menu item ID of the item to insert after
startingID	**Word**—Menu item ID to use for the first item inserted
— *resultPtr* —	**Long**—Pointer to the result buffer
	<—SP

Stack after call

Previous contents	
	<—SP

Errors Errors from `InsertMenuItem2` and the Memory Manager are returned unchanged.

C

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

`flags` This parameter is a flags word.

bits 15-5 Reserved; set to 0.

bit 4 If this bit is set, `pathPtr` must point to a fully-expanded pathname. If the bit is clear, `pathPtr` can point to a pathname that is not fully expanded.

bit 3 If this bit is set, `deviceNum` is a valid device number; if it is clear, `InsertPathMItems` calls `GetDevNumber` to get the device number.

bit 2 — If this bit is set, open folder icons will be used beside folder icons. If the bit is clear, closed folder icons will be used.

bit 1 — Reserved; set to 0.

bit 0 — If this bit is set, the items are inserted with the device at the top and the file at the bottom; if the bit is clear, the opposite order is used.

`pathPtr` Pointer to a GS/OS input 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 off line.

`menuID` The menu ID for the menu to insert the menu items into. This is passed to `InsertMItem2`.

`afterID` The menu item ID for the menu item to insert the new items after. Pass $0000 to insert the items at the top of the menu.

`startingID` The menu ID for the first item to be created and inserted in the menu. The menu ID is incremented by one for each additional item added to the menu.

The items are always inserted working from left to right in the pathname, whether bit 0 of `flags` is set or clear. The first menu item inserted has a menu item ID of `startingID`, the second has a menu item ID of `startingID`+1, and so forth.

`resultPtr` Pointer to a 10-byte result buffer with this format:

$00	*menuItemID*	**Word**—Highest menu item ID inserted
$02	— *theHandle1* —	**Long**—First handle to dispose
$06	— *theHandle2* —	**Long**—Second handle to dispose

The handles are dynamically allocated memory areas that you must dispose of after the menu items are no longer needed.

RemoveMItemStruct $4B0F

`RemoveMItemStruct` 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 `RemoveMItemStruct`.

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 control, you must first set the current menu bar to your Pop-up control handle.

Parameters

Stack before call

Previous contents	
itemID	**Word**—Menu item ID
	<—SP

Stack after call

Previous contents	
	<—SP

Errors None.

C

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

`itemID` Menu item number for the menu item whose `itemStruct` will be returned.

SetMItemFlag2 $4D0F

SetMItemFlag2 sets the itemFlag2 field for the itemStruct record of the indicated menu item to the value passed. If you want 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.

See "Icons and Menu Items," earlier in this chapter, for a description of the itemStruct record.

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

Parameters

Stack before call

Previous contents	
newValue	**Word**—New value for itemFlag2 field
itemID	**Word**—Menu item ID
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $0F03 menuNoStruct Returned if bit 10 of itemFlag is not set.

C

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

newValue New value for the itemFlag2 field of the itemStruct record.

ItemID Menu item number for the menu item to change.

SetMItemIcon $470F

`SetMItemIcon` sets the `ItemIconRef` field in the `itemStruct` record for the menu item indicated.

`SetMItemIcon` can change the width of a menu, so you must call `CalcMenuSize` after calling `SetMItemIcon`.

The parameter `IconDesc` is used by the call to set the `itemFlag2` field correctly.

See "Icons and Menu Items," earlier in this chapter, for a description of the `itemStruct` record.

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

Parameters

Stack before call

Previous contents	
iconDesc	**Word**—Describes how icon is to be referenced
— *iconRef* —	**Long**—Reference to icon
itemID	**Word**—Menu item ID
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $0F03 `menuNoStruct` Returned if bit 10 of `itemFlag` is not set.

C

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

`iconDesc` Describes how the icon is referenced. This value replaces the value used for the `itemflag2` field of the `itemStruct` record.

`iconRef` Reference for the icon. This value replaces the value used for the `itemIconRef` field of the `itemStruct` record.

`itemID` Menu ID for the menu item to change.

SetMItemStruct $490F

`SetMItemStruct` 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.

See "Icons and Menu Items," earlier in this chapter, for a description of the `itemStruct` record.

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

Parameters

Stack before call

Previous contents	
itemStructDesc	**Word**—Describes how `itemStruct` is to be referenced
itemStructRef	**Long**—Reference to `itemStruct`
itemID	**Word**—Menu item ID
	<—SP

Stack after call

Previous contents	
	<—SP

Errors None.

C

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

`itemStructDesc` Describes how the icon is referenced.

$0000	Reference is by pointer.
$0001	Reference is by handle.
$0002	Reference is by resource ID.
$0003	Reserved.

`itemStructRef` New `itemStruct` reference.

`itemID` Menu item number for the menu item to change.

Chapter 14 **MIDI Synth Tool Set**

This chapter documents the MIDI Synth Tool Set. This is new material, not published in Volumes 1 to 3 of the *Apple IIGS Toolbox Reference*.

About the MIDI Synth Tool Set

MIDI Synth is a second generation note synthesizer Tool for the Apple IIGS. By integrating a completely new sequencer, MIDI interface and synthesizer into one program environment, MIDI Synth offers developers a powerful but simple solution to many sound needs. Because of this integration, most of the work required by an application to produce music is handled by this tool. Whether you're writing a music education application to teach elements of music or an arcade game which needs both music and sound effects in the background, you'll find MIDI Synth to be an important tool.

Some important features of MIDI Synth are:

- Integrated synthesizer, sequencer and MIDI interface in one tool.
- Simple programmer's interface – you don't need to understand the complex Apple IIGS sound hardware.
- It's fast, using only 25-30% of the CPU time under typical conditions.
- MIDI Synth does not complicate your program structure, since it runs completely in the background.

Synthesizer

- The synthesizer can produce complex and interesting sounds with a 4 oscillator per voice architecture.
- The synthesizer features two 8-stage volume envelopes per voice with velocity control and variable note position decay.
- Pitch bend up to ± 1 octave.
- "Multi-sample" up to 8 waveforms per instrument across the keyboard range.
- Flexible control of oscillators, with 6 oscillator configurations to choose from.
- Multi-timbral; simultaneously play any combination of instruments, up to 7 voices.
- Sophisticated voice "stealing" algorithm extends the 7 voice limit to sound like more.
- Automatically handles MIDI Volume and Sustain Switch messages.
- Allows for alternate scale tuning arrangements.
- Supports Omni, Poly and Multi MIDI modes.

Sequencer

- 16 track sequencer with 96 ticks per quarter note resolution at tempos from 10 to 265 beats per minute.
- Synchronize to external MIDI devices
- Can wait for a MIDI key input to start sequence.
- Can count off a measure before starting sequence.
- An audible metronome is built in.
- The play and record buffer is limited in size only by available memory.
- Buffer support routines are included, supporting fast merging, locating, and time (ticks) to measure conversions.

Support

- System Software 6.0 comes with synthLAB, an Apple application for designing your own instruments.
- The Apple Instrument Library is available for use in your applications.

Limitations

In order to gain speed and to simplify the programmer's interface, most of the MIDI, sequencer and synthesizer functions have been "packaged" to do specific things. The price paid for this comes in the form of reduced flexibility and "system" considerations. This tool completely takes over the Apple IIGS sound hardware. Sound interrupts, DOC RAM and registers, and serial ports are all exclusively controlled by MIDI Synth and are unavailable for application or system use. Also, MIDI Synth does not support most of the other sound tools. For example, an application which uses this tool can not use the AppleTalk port, support "system beep" initialization programs or use the Note Sequencer (Tool 26) or Note Synthesizer (Tool 25) while this tool is active (although one generator is available for use with the Sound Tool Set).

Previous Sound Tool Set data formats are not compatible with MIDI Synth. Note Sequencer tool sequences, MIDI Tool Set sequences and Note Synthesizer instruments cannot be used with the MIDI Synth.

Overview of the MIDI Synth Tool Set

Referring to the "MIDI Synth Block Diagram", the three components which make up this tool are:

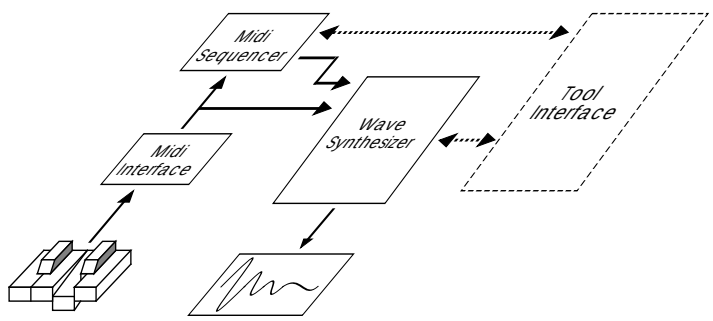
1. MIDI Interface
2. MIDI Sequencer
3. Wave Synthesizer

Your application controls the way these parts interact through the various tool calls.

The Wave Synthesizer produces the output sound by playing a specified **instrument record**. An instrument record tells the Wave Synthesizer how to build a particular sound, how it should behave over time and how it should respond to certain MIDI events. Commands to play an instrument can come from either the MIDI Interface, the Sequencer, directly from your application, or from all three simultaneously.

The Sequencer can either play a pre-recorded MIDI sequence or it can record a MIDI sequence for you. When it records a sequence, the sequencer reads MIDI data from the MIDI port, time-stamps this data to remember when it happened, and then sends this final time-stamped MIDI event (called a **Seq Item**) to a user-specified buffer somewhere in system memory (which is called a **sequence**). Later you can tell the Sequencer to play back this sequence. The Sequencer will now send all the Seq Items in the sequence to either the MIDI port, the Synthesizer or both, in exactly the same order and relative time as it was recorded. In other words, all notes and rhythms are accurately preserved. You can also simultaneously play one buffer while recording MIDI events into a second buffer; that is, play and record at the same time.

The MIDI Interface monitors the MIDI port for input data and notifies the Sequencer and the Wave Synthesizer when data is received. It can also send output data through the same port when the Sequencer is playing a sequence.



MIDI Synth Block Diagram

Starting MIDI Synth

Here is a general outline for starting MIDI Synth. More specific examples can be found in the section “Using MIDI Synth”.

1. Make sure AppleTalk is off.
2. Start the Sound Tool Set (Tool 08) with an `SoundStartUp` call.
3. Start the Miscellaneous Tool Set (Tool 03).
4. Start the MIDI Synth Tool Set with an `MSStartUp` call.
5. Load your Instrument waves into DOC RAM with the Sound Tool Set call `WriteRamBlock`.
6. Define all your Instruments with repeated `SetInstrument` calls.
7. If you’re using MIDI input or output, load a MIDI driver. Call `InitMIDIDriver` to initialize the drivers you load. Enable MIDI I/O with a `SetMIDIPort` call.

When your application finishes, MIDI Synth should be shut-down (`MSShutDown`) before closing the Sound Tool Set.

Interrupts

Since MIDI Synth runs almost exclusively in the background, special consideration must be given if the application disables interrupts. Disabling interrupts will halt most of the tool's operations and MIDI data will be lost to over-run errors. Since the Apple IIGS interrupt handler is not re-entrant, interrupt routines (VBL tasks, MIDI Synth callback routines) should be short and fast. The recommended technique is to only set a flag or increment a counter at interrupt time, servicing any action required while outside the interrupt.

You may want to temporarily stop MIDI Synth, perhaps when you're doing heavy graphic drawing or intense calculations (even when it's not playing a sound, update interrupts are still being serviced while MIDI Synth is active; this takes about 150µs of time every 5ms). Issuing a `MSSuspend` call will turn the 5ms update interrupts off. If any notes are playing at this time they will freeze at whatever level they were when the call was made. If MIDI input is enabled, MIDI input data is still buffered. (Keep in mind that the internal MIDI buffer is only 128 bytes long, so it can easily overflow, causing you to lose notes.) To turn MIDI Synth back on and continue from where you left off, call `MSResume`. If there are any buffered MIDI notes at this time, they will all sound at once.

MIDI Synth Oscillators and Voices

The Apple IIGS hardware supports 32 digital oscillators. MIDI Synth uses one of these (Oscillator #31) as a time-base for background updating every 5ms. Another oscillator (Oscillator #30) is used by MIDI Synth to play the built-in metronome. Two oscillators (Oscillators #28 and #29 - Generator #7) are reserved for your application to play sampled sounds from system memory with the Sound Tool Set (see next section). This leaves MIDI Synth with 28 oscillators (#0 thru #27) to create its Synthesizer voices. Since each MIDI Synth voice uses 4 oscillators (see "Voice Architecture" section), a maximum 7 voices or instruments can be active at any given time.

These 7 voices are **dynamically** assigned to you by MIDI Synth. This means that when a request is given to play a note (either thru MIDI, the Sequencer or by your application), MIDI Synth will decide which of the 28 oscillators it will use to play the requested Instrument. If all 7 voices are currently being used, MIDI Synth will "steal" (turn the least noticeable note off) a voice in order to free four oscillators for the new note. In most cases, your application should not concern itself with voices or oscillators, since MIDI Synth manages these automatically for you.

Using MIDI Synth with the Sound Tool Set

You application may need some sounds which are either impossible to synthesize or too large to fit into DOC RAM as Instrument waveforms. These are special cases where you may want to play back sampled digital recordings stored somewhere in system RAM. MIDI Synth itself cannot play these, but special "hooks" have been provided for your application so you can use Sound Tool Set (Tool #08) calls to play these sampled recordings.

Since MIDI Synth takes full control of the Apple IIGS sound hardware and sound interrupts, there are some restrictions to keep in mind when you use the Sound Tool Set with MIDI Synth. The only available Sound Tool Set calls you can use are:

1. `SoundStartUp` and `SoundShutDown` at the beginning and end of your application.
2. `ReadRamBlock` and `WriteRamBlock`, and only when there are no active MIDI Synth voices.
3. `SetSoundVolume`, but only set the system volume or generator #7.
4. `FFStartSound` and `FFStopSound`, using only generator #7.
5. `FFSoundDoneStatus` and `FFGeneratorStatus` for generator #7 only.
6. You can use the **low level routines** at your risk, and only when there are no active MIDI Synth voices.

You will have to use `WriteRamBlock` to enter your instrument waveforms into DOC RAM. Make sure interrupts are disabled when you do this. Be aware that `FFStartSound` writes to 2 buffers in DOC RAM when playing a sample, so it may wipe out Instrument waveforms used by MIDI Synth. Both the Sound Tool Set and MIDI Synth will be competing for interrupt time when you use `FFStartSound`, so output quality and reliability will probably degrade when you use them simultaneously. The recommended sequence for playing a sampled sound is to turn MIDI Synth off temporarily with a `MSSuspend` call, play your sampled sound, then re-enable MIDI Synth with a `MSResume` call.

MIDI

MIDI Synth handles most of the interface to MIDI for you. Channel Voice messages and System Real-Time messages are both processed automatically by the tool. Since this processing is done during interrupts, your application doesn't have to do anything while the Sequencer is playing or recording, or while MIDI data is received at the MIDI port. This frees your application from the tedious work required to service the Wave Synthesizer, Sequencer and MIDI interface.

The chart "Recognized MIDI Messages" shows the MIDI messages supported by MIDI Synth. They can be sent by external MIDI devices into the MIDI port, played by a sequencer or sent directly by your application using the `MIDIMessage` call. Those Channel Voice messages not shown on the chart (for example a Modulation Wheel) will have no effect on MIDI Synth. They can, however, be recorded and played back through the Sequencer to an external MIDI device connected to the MIDI port.

Status	Data #1	Data #2	Description
$8x	**Note #** 0-127	**Velocity** 0-127	Note Off Velocity is ignored
$9x	**Note #** 0-127	**Velocity** 0-127	Note On Note Off when Velocity =0
$Bx	**Control #** 7 64 123-127	**Value** 0-127 0=Off 127=On 00	MIDI Volume Sustain Switch All Notes Off
$Cx	**Instrument #** 0-15	---------	Program Change
$Ex	**LSB** 00	**MSB** 0-127	Pitch Bend Center = 64
$F8	---------	---------	Sequence Timing Clock
$FA	---------	---------	Start Sequence
$FB	---------	---------	Continue Sequence
$FC	---------	---------	Stop Sequence

Recognized MIDI Messages

Channel Mode messages (Controllers 122-127) function as “All Notes Off” messages only. An All Notes Off message does not immediately shut the voices off. Instead, all voices are forced into the Release 1 segment of the envelope.

System Common messages ($F1-$F6), Active Sensing ($FE) and System Reset ($FF) are not supported and are filtered out by MIDI Synth.

System Real-Time messages (Start ($FA), Stop ($FC) and Continue ($FB)) are supported through callbacks (see “Callback Routines,” later in this chapter). Stop will halt the Sequencer if it is active.

When enabled, the Sequence Timing Clock ($F8) is handled internally. MIDI Synth can only synchronize to an external Timing Clock; it will not generate them while playing. This means that MIDI Synth can only be a slave device and not a master.

System Exclusive messages ($F0 xx xx ...) are supported only through callbacks. Once MIDI Synth receives a Sys Ex message, all further MIDI data is ignored internally until another MIDI Status (most significant bit set) or an EOX ($F7) is received.

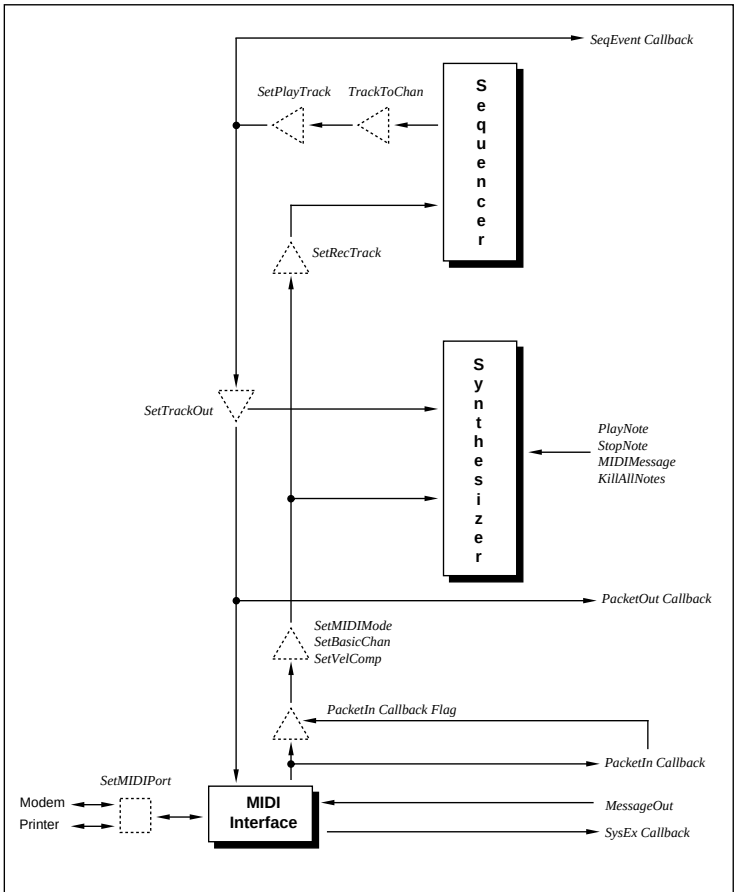
The Program Change message ($Cx) is supported in MIDI Synth callback.

MIDI Data Flow

To get a conceptual idea on how MIDI Synth is structured, you can think of it as consisting of three separate modules which communicate with each other and your application via MIDI messages. The three modules, as mentioned earlier, are the MIDI Interface, the Sequencer and the Synthesizer. Each module is basically independent of the others and can be used by your application without the others. In other words, your application can use the Synthesizer without the MIDI Interface and Sequencer, or you can use the Synthesizer with the Sequencer but without the MIDI Interface. Your application picks the combination, which can be dynamically as the

application runs. By putting these three modules into one Tool, you get the important advantage of efficiency and streamlined operation. Each module intimately knows how the others operate and how to communicate with them.

The internal communication paths are altered by your application via data filters which modify and control the MIDI data. The “MIDI Data Flow” diagram shows the communication paths, location of each filter (dotted triangles in the diagram), the Tool calls which affect the filters, and finally, where your routines get called during callbacks.



MIDI Data Flow

MIDI Input Path

Starting at the bottom left corner in the diagram, the source for MIDI data from external devices is set by the `SetMIDIPort` call. This call selects which SCC port MIDI Synth will “listen” to for MIDI data. If neither the Modem nor the Printer port is selected, the MIDI port is disabled and data cannot flow through this path. The first thing the MIDI Interface does when it receives an input byte is to intelligently accumulate the input data in order to build a MIDI message packet. It examines the current protocol and collects the input until it has a complete MIDI packet.

After the full MIDI message packet is received, MIDI Synth calls your `PacketIn` callback routine. This is where your application can modify and filter the MIDI messages as they come in. If the carry flag is set when your application returns to MIDI Synth, the message will be discarded and

will not be sent to the Synthesizer and Sequencer. If the carry flag is clear when returning from your `PacketIn` callback the message continues to the next filter.

This next filter does several things. It can modify or ignore this MIDI message, depending on the current MIDI mode (Omni, Poly or Multi) and the Basic Channel value. If the mode is Omni, the message channel number is forced to the Basic Channel number. If we're in Poly mode, MIDI Synth compares the message channel number with the Basic Channel, and if they don't match, the message is discarded. The message channel number is not modified if we're in Multi mode. The last thing this filter does is to pad the velocity value (if the message is "note on") with the value set by the `SetVelComp` call. The Synthesizer will play the message if it makes it through this filter

If the Sequencer is currently recording MIDI data, a track number reference (set by the `SetRecTrack` call) is attached to this message by one final filter before the Sequencer receives it. If no tracks are set for Record, then the message gets discarded, and will not be recorded by the Sequencer.

Sequencer Output Path

As shown in the diagram, the first main MIDI communication path is from the MIDI port to the Synthesizer, and finally to the Sequencer. The other major path is the reverse of this, sending MIDI Messages from the Sequencer to the Synthesizer, and finally out through the MIDI port.

After leaving the Sequencer, the message's channel number may get translated to the value set by the `TrackToChan` call. This call enables you to force the channel numbers of all messages (Seq Items) in a particular track to a specified value. Since the message channel number is used as a reference to Instrument numbers in the Synthesizer, this forces all messages in a track to play a specific instrument.

MIDI Synth then checks the track number in the message to see if that track is active (`SetPlayTrack` enables or disables the track). If it is set as active, the message continues down the path; otherwise, it gets discarded here and goes no further.

At this point, MIDI Synth calls your `SeqEvent` callback routine, then checks to see where to output this message based on its track number. You can use the `SetTrackOut` call to specify a track to play only the Synthesizer, or to only output the track thru the MIDI port, or both. Your application can monitor those messages which will be sent out the MIDI port with the `PacketOut` callback routine.

Other MIDI Paths

There are two other message paths in MIDI Synth. The first is between your application and the Synthesizer using the `PlayNote`, `StopNote`, `MIDIMessage` and `KillAllNotes` calls. The second path enables you to send MIDI Messages directly to the MIDI port via the `MessageOut` call.

For more detailed information about these callbacks, see "Callback Routines," later in this chapter.

Voice Architecture

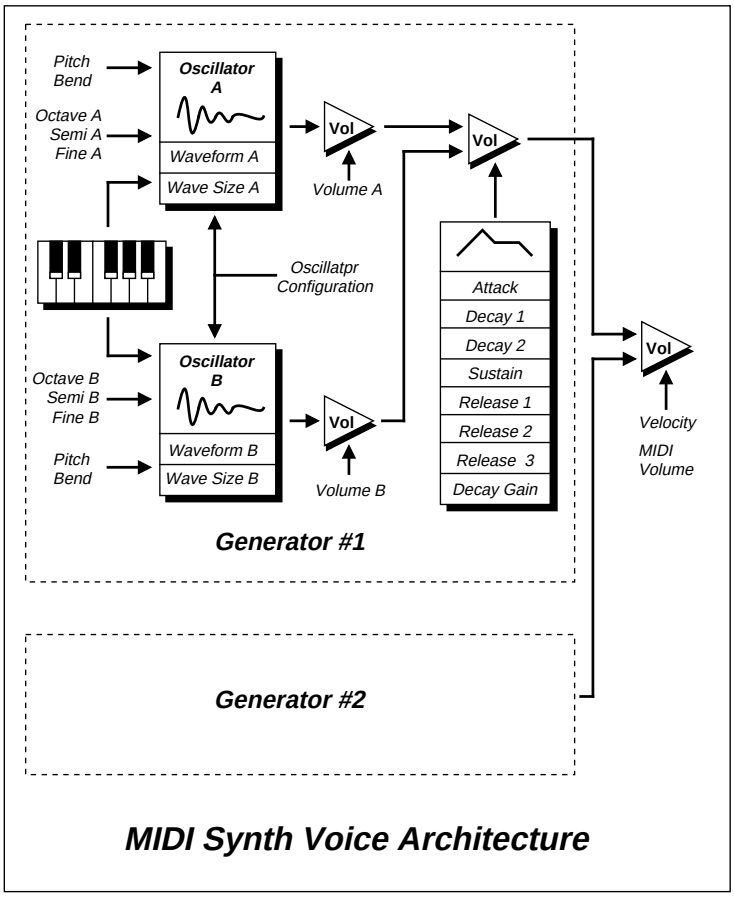
In the early days of music synthesis synthesizers were made by connecting a group of analog sound modules together with wires and adjusting the various parameters with control knobs to create an instrument. A new instrument was created by reconfiguring the sound modules and

setting the control knobs to new values. Modern digital synthesizers are similar, but the sound modules are now subroutines, the knobs are parameters in memory and the wires are control instructions changing the flow of the program. Using this analogy, an instrument record contains instructions for one of the "sound modules," telling where to connect the "wires" and how to adjust the various "knobs". The basic synthesizer structure and its elements is called the voice architecture. This defines what parameters are available to the programmer, where the sound sources come from and the types of control devices that are built in the synthesizer.

The MIDI Synth voice architecture is designed around two identical **generators**. Each generator has two oscillators and level controls for each. The oscillators are mixed into a variable amplifier whose gain is controlled by an eight stage envelope. The envelope is a series of linear ramps that gives the sound a volume contour that changes over time. Finally, both generators are summed into another variable amplifier that is modified by the note velocity parameter and by any ongoing MIDI volume message for that instrument.

The oscillators are the most unique part of this architecture. Each oscillator can play any waveform in DOC memory. The oscillator configuration parameter controls how the two oscillators of each generator behave and how they interact. Some configurations play the oscillators continually, while others force the oscillators to play the waveform only once and stop. Other configurations start an oscillator only when another oscillator has finished its waveform.

The Synthesizer has seven dynamically assigned output voices. This means that no more than seven notes can be active and playing simultaneously. The seven voices can be any combination or number of instruments assigned to the Synthesizer. If the Synthesizer is asked to play more than seven voices, a voice will be "stolen" from an active note to make room for the requested voice. When stealing a note, the synthesizer tries to pick the note that will be least noticed when turned off.



Instrument Records

As mentioned above, the instrument record contains the parameters that define a particular instrument. It tunes the oscillators, points to waves, sets the oscillator mode, controls the amplifiers and defines the envelope. The instrument record itself is made up of two groups of records; they are envelope records and wavelist records. Since there are two generators for each instrument, these records are grouped according to which generator they control.

The **envelope record** contains rate and level values for the ramps that make up the envelope, as well as a few other parameters that affect all wave lists. There are two envelope records per instrument, one for each generator.

The **wavelist record** controls the oscillators. How the oscillators are tuned, what wave they play and what configuration they're in are all controlled by this record. There are eight of these records for each generator, each of which can be active only when the note that they must play falls into their note zone or range. Each wavelist record has a note range parameter called Top Key that sets this zone (see "Wavelist Record," later in this chapter, for an example and details on Top Key).

	Offset	
Generator 1	0	Envelope Record
	16	Wavelist Record
	32	Wavelist Record
	48	Wavelist Record
	64	Wavelist Record
	80	Wavelist Record
	96	Wavelist Record
	112	Wavelist Record
	128	Wavelist Record
Generator 2	144	Envelope Record
	160	Wavelist Record
	176	Wavelist Record
	192	Wavelist Record
	208	Wavelist Record
	224	Wavelist Record
	240	Wavelist Record
	256	Wavelist Record
	272	Wavelist Record

Instrument Record

Offset		Value
0	Attack Level	0-127
1	Attack Rate	0-31
2	Decay 1 Level	0-127
3	Decay 1 Rate	0-31
4	Decay 2 Level	0-127
5	Decay 2 Rate	0-31
6	Sustain Level	0-127
7	Decay 3 Rate	0-31
8	Release 1 Level	0-127
9	Release 1 Rate	0-31
10	Release 2 Level	0-127
11	Release 2 Rate	0-31
12	Release 3 Rate	0-31
13	Decay Gain	0-9
14	Velocity Gain	0-10
15	Pitch Bend Range	0-12

Envelope Record

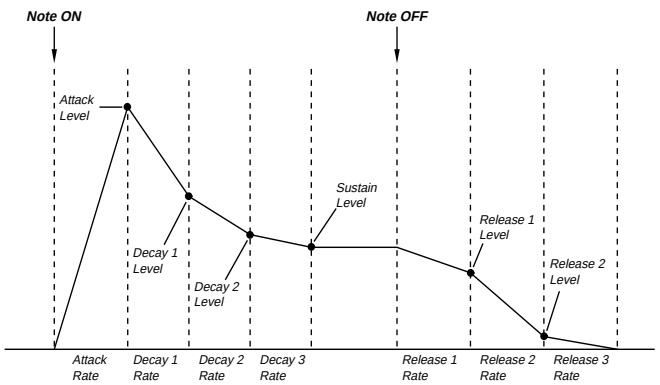
The envelope parameters control how the volume changes for the instrument while the note is playing. All natural sounds have some sort of envelope. They may start softly and gradually become louder or they might start loud and slowly fade out. With the envelope parameters we imitate this by piecing together a series of volume segments to the shape that matches the effect we want. Each segment has a target volume level and a rate value which sets how fast the volume changes until it reaches the target level. In other words, the volume starts at the previous target level, then increases or decreases at a programmed rate until it gets to the new target level. With eight segments in the envelope, very complex contours can be defined.

When a note is started, the volume starts at a zero level (silence) and increases to the attack level with a slope set by the attack rate. From there, it goes to the decay 1 level at the decay 1 rate. Next it moves to the decay 2 level with the decay 2 rate. Using the decay 3 rate, the envelope finally reaches the sustain level. At this point the envelope stops changing and waits for the note to be released (MIDI note off message). It sits at this level until it gets a command to turn the note off. The envelope then steps through the release 1, 2 and 3 segments until it is finally off and stops. Notice that the release 3 segment always decreases to a zero level (that's why there is no release 3 level setting; it's always zero).

The segments can either increase or decrease to the new target depending on whether the new target level is above or below the previous target level. For example, if decay 1 level is higher than the attack level, the decay 1 segment will increase from the attack level to the decay 1 level. If decay 1 is lower, the segment will decrease the volume until it reaches the decay 1 level.

If a note off command is sent to the synthesizer (MIDI Note Off message or a `StopNote` tool call) before the envelope has reached the sustain stage, the envelope will always be forced to the release 1 segment. For instance, if the envelope is somewhere in the attack segment when a note off command is sent, the envelope will immediately start moving from its present point to the release 1 level at the release 1 rate.

The envelope stops whenever it reaches a zero level. If the sustain level is set to zero, the end of the decay 3 segment will be considered the end of the envelope. Once it reaches this point, the envelope will remain at zero, ignoring note off commands. The same is true for any of the envelope segments. However, before reaching this point, a note off will still force the envelope to the release 1 segment.



Envelope Parameters

Once both envelopes finish (both at zero levels) the the oscillators are turned off and the note is considered to be inactive. Sending Note Off, pitch Bend or MIDI Volume commands at this point will have no effect.

One point to keep in mind is that the 31 slopes that can be set for each segment are **rate** values and not **time** values. For example, if a certain segment had a rate value of 3 and a level of 127, it will take 6.92 seconds to complete the segment if it started from a zero level (for example attack). If the segment started from a value of 64 the segment will take half as much time to complete, since it has only half the distance to cover (from 64 to 127). Likewise, if the level is halved, then the time to complete the segment is also halved to 3.46 seconds. The envelope rate chart shows times for each rate when the segment must go full range from 0 to 127.

Rate Value	Time*
31	0.00
30	.01
29	.02
28	.03
27	.04
26	.05
25	.06
24	.08
23	.09
22	.12
21	.16
20	.18
19	.23
18	.28
17	.32
16	.43

Rate Value	Time*
15	.54
14	.64
13	.82
12	1.02
11	1.26
10	1.56
9	1.93
8	2.39
7	2.96
6	3.66
5	4.52
4	5.59
3	6.92
2	8.55
1	10.53
0	13.08

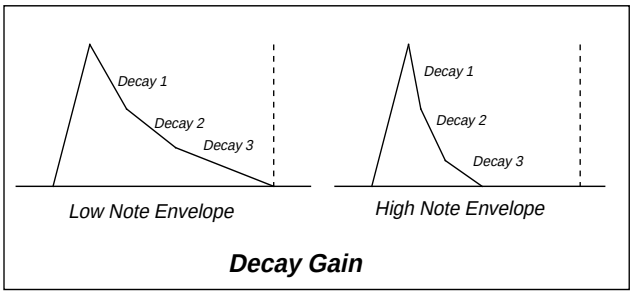
** Time is in seconds to ramp full scale (0-127)*

Envelope Rates

Decay Gain 0-9

In nature, percussive sounds have a unique feature. High frequencies die out much quicker than low frequencies. When playing a piano you will notice that striking and holding down a low note causes the strings to sound and resonate for quite a long time. If you try the same on high notes,

the strings loose their energy quickly and die out. The decay gain parameter imitates this important feature by automatically increasing the entered decay rate values for notes as they go up the scale. With the decay gain set to zero the decay rate for all notes will be the same as the entered value, both low and high notes will decay at the set rates. Increasing the decay gain value will cause higher notes to decay faster than the entered amount. This parameter affects higher notes more than lower notes.



Velocity Gain 0-10

Velocity gain controls the instrument's sensitivity to MIDI velocity data. The larger the value, the more sensitive velocity becomes. A value of zero causes no effect on velocity data. This parameter can be used to match the dynamic characteristics of instruments you wish to create. For example, a piano, which has a very large dynamic range, may have a velocity gain of 1, while a pipe organ, which has no dynamic range, may have gain of 10. Velocity gain can also be used to compensate for performance variations on MIDI keyboards. Some players pound the keys when they play while others have a gentler style. Different manufactures also have different velocity ranges on their keyboards.

Pitch Bend Range 0-12

This parameter sets how far MIDI pitch bend data can bend a note up or down. The values are in semitone increments. A zero value disables the pitch bend feature. **It is recommended that this parameter be set to zero unless you really plan to use it.** Disabling Pitch Bend will decrease overall MIDI Synth CPU overhead.

Wavelist Record

Here's a diagram of the wavelist record:

$00	*topKey*	**Byte**—Top key value (0-127)
$01	*oscConfig*	**Byte**—Oscillator configuration (0-5)
$02	*stereo*	**Byte**—Stereo mode (0-63)
$03	*detune*	**Byte**—relative tuning (0-63)
$04	*waveAddressA*	**Byte**—First wave address ($00-$FF)
$05	*waveSizeA*	**Byte**—First wave size (0-7)
$06	*volumeA*	**Byte**—First wave volume (0-127)
$07	*octaveA*	**Byte**—Tuning factor (see discussion) (0-6)
$08	*semitoneA*	**Byte**—Tuning factor (see discussion) (0-11)
$09	*fineTuneA*	**Byte**—Tuning factor (see discussion) (0-63)
$0A	*waveAddressB*	**Byte**—Second wave address ($00-$FF)
$0B	*waveSizeB*	**Byte**—Second wave size (0-7)

$0C	*volumeB*	**Byte**—Second wave volume (0-127)
$0D	*octaveB*	**Byte**—Tuning factor (see discussion) (0-6)
$0E	*semitoneB*	**Byte**—Tuning factor (see discussion) (0-11)
$0F	*fineTuneB*	**Byte**—Tuning factor (see discussion) (0-63)

The various parameters and how they are used are explained by parameter in the next few sections.

`topKey`

If you built an instrument based on a sampled middle C female voice and played it back with a middle C note, it would sound the same as the original. Play a few notes around middle C and it still will sound much like the female voice you sampled. Although the pitch is different, the same female voice characteristics are still there. Now, if you played this same sample an octave below middle C, you will be shocked to hear this pleasant female suddenly sound like Darth Vader. Play the sample an octave above middle C and the voice now sounds like Minnie Mouse. There is only a very small useful range of notes where the sample sounds like a female voice. This is because in nature the physics of the instrument produces a different waveform for every new pitch. As in the voice, the mouth acts as a variable filter. For every single pitch, the filter has a different effect on the harmonics produced by the voice. From this, it would seem that in order to reproduce any natural instrument, a separate sample for each note is required. Fortunately, you can usually get away with one sample for a range of notes. The size of a range depends on the particular instrument.

The technique of having many samples of the same instrument across the pitch range is called **multi-sampling**. This is the reason why there are eight wave lists for each generator. You can assign each wavelist to be active only in a certain pitch range. As in the voice example, each wavelist could be set to use a different voice sample and each one of the eight wave lists could be programmed to respond to a different pitch range.

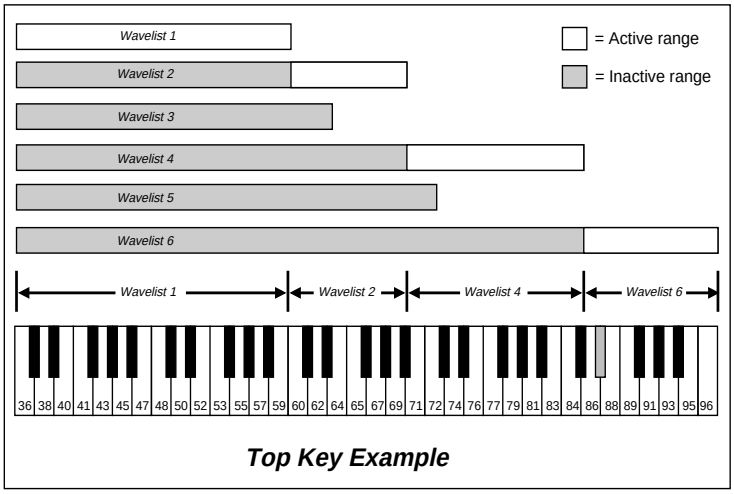
This zone is set by the `topKey` parameter. Its values are MIDI Pitch numbers from 0 to 127 with 60 being middle C. Before an instrument starts playing, the wave lists are scanned to find which one of the eight has a top key greater or equal to the pitch we wish to play. The first one found that satisfies this condition is the one that gets used. The wave lists get scanned in order from one to eight. Only `topKeys` in Generator 1 are used in the scan. When a match is found, the same numbered wavelist for Generator 2 is used. In other words, if Wavelist 4 from Generator 1 is used, then Wavelist 4 from Generator 2 is also used. Nothing will be played if all the `topKeys` are less than the desired pitch.

Suppose the `topKey` parameters from each Wavelist for Gen 1 were set as follows:

Wavelist 1 `topKey`	59
Wavelist 2 `topKey`	70
Wavelist 3 `topKey`	62
Wavelist 4 `topKey`	84
Wavelist 5 `topKey`	72
Wavelist 6 `topKey`	127

It doesn't matter what Wavelist 7 and 8 have for `topKey` because the first 6 cover the entire MIDI range. The last two wave lists will never be scanned because one of the lower wave lists will always claim the note. Now suppose the D# two octaves above middle C is played by an instrument; that's MIDI note 87. Wavelists 1 and 2 won't be used because their `topKeys` are less than 87. Wavelist 3 will never be used for any note because with a value of 62, either Wavelist 1

or 2 will always claim a note in its range. Wavelist 4 is under and Wavelist 5 is always locked out like Wavelist 3 because Wavelists 1,2 and 4 cover its range. Wavelist 6 is used since its range covers notes 84-127.



Top Key Example

`oscConfig`

The `oscConfig` parameter controls how oscillators A and B for each of the two generators are organized and how they interact with one another. A short discussion on how MIDI Synth creates sounds is necessary to understand why these different configurations exist and how to effectively use them.

Synthesizer Basics

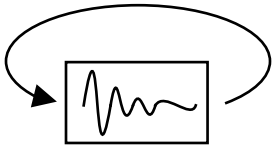
MIDI Synth supports two basic methods of sound generation: additive synthesis and sampling. With additive synthesis, instruments are built from scratch by combining simple elements of the sound together to form a more complex sound at the end. For example, to build an orchestral string sound, you could take a single cycle wave that approximates the timbre of a violin and assign an oscillator to continually play that wave. To get a full orchestral sound, you might add in other oscillators that are playing viola and cello timbres and slightly detune them to create a complex phase-shifting effect. Finally you would add a volume envelope to these oscillators so that their attacks and decays match those of a real orchestra. The advantage of additive synthesis is that it is very flexible, since you have control over the many elements that make the sound. You can change the envelopes, change waves for different timbres or tune the oscillators, giving subtle or drastic changes in the sound. Additive synthesis is also very memory-efficient. Storing only single-cycle waves takes very little memory. Since you control the timbres and since the waves are always full scale, additive synthesis sound quality is usually very clean. The disadvantage of additive synthesis is that it takes a lot of CPU overhead to manage many oscillators and envelopes in real-time.

With sampling, instead of building a sound from scratch, you simply digitize the final sound you want and have the computer play it back as one complete sample. In the orchestral strings example, you would feed the output of a CD player or tape deck into your Apple IIGS and digitally record a section that has orchestral strings in it. When you play it back, all the subtle complexities, richness and characteristics of the sound are reproduced. The obvious advantage with sampling is that you can very easily produce complex sounds without having to painstakingly build the sound from scratch. You can record sounds that would be almost impossible to synthesize. Sampling

requires very little CPU overhead, since all you manage is a single oscillator and the envelope is already part of the sample. But there are many problems with sampling. Since the samples are digital images of the sound, even short samples can take huge amounts of memory. The longer the sample, the more memory it takes. Sampling has no flexibility – if you wanted an orchestral string sound with a harder attack or a longer release, you would have to record another huge sample and load it into free memory (if you could in fact find such a recording). Because of sampling and playback rate limitations, and the fact that the envelope is part of the digital information (lower signal to noise ratio), sampling is usually very noisy and has a perceptible "grainy" quality to it.

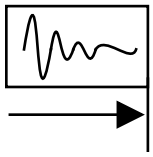
Since MIDI Synth supports both methods, you can overcome many of the problems of each by combining the advantages of the two methods to form a third hybrid synthesis method. For example, to make the orchestral strings, you would sample the attack portion of the strings, capturing all the complexities of bows striking the strings, and so forth, then assign one oscillator to play this sample. The remaining three oscillators can be combined to play the sustain and release portion of the sound using additive synthesis. This gives you a built an instrument that has captured the complexities of the sound (the attack would be very hard to synthesize) and at the same time has an clean sustain that can be played infinitely without taking up large amounts of memory. If you needed a longer release, you simply change the envelope release rate. If you wanted to create something never heard before you could use the string attack, but use a piano sustain.

Each one of the MIDI Synth oscillators operates in one of two basic modes: loop mode or one-shot mode. In **loop** mode, the oscillator scans the wave from the beginning to the end, then jumps back to the beginning and starts scanning the wave again. It does this continually until the oscillator is turned off. If the wave is only a single cycle, you hear a static pitch with a timbre that is characteristic of the given wave. For example, when playing a square wave, you would hear a constant square wave "buzz". On the other hand, if the wave is a recorded sample of a sound with many dynamically varying waves to it, you will hear the same sound constantly repeat itself over and over again. If, for example, you took a sample of the word "hello," playing it in loop mode would give a repeating pattern of "hello hello hello…" Low notes result in a slow repeating pattern, while high notes repeat quickly, since they get scanned faster and therefore finish sooner.



Loop Mode

In **one-shot** mode the oscillator scans the wave once and stops. It stays silent for the remaining part of the note. The "hello" sample would sound only once, and the square wave would produce only a "click" sound. This mode is primarily used to play back short recorded samples.



One Shot Mode

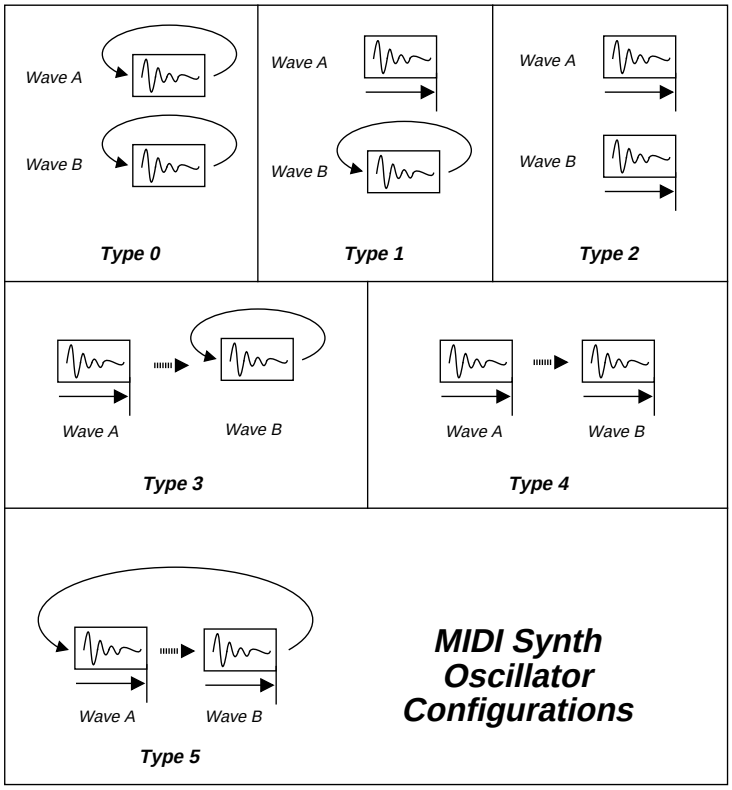
The oscillator configuration parameter organizes the oscillators with the two modes in various ways.

`oscConfig`	Meaning
0	Both Oscillator A and Oscillator B play in loop mode.
1	Oscillator A plays in one-shot mode with Oscillator B in loop mode.
2	Both Oscillator A and Oscillator B play in one-shot mode.
3	Oscillator A plays in one-shot mode, followed by Oscillator B in loop mode.
4	Oscillator A plays in one-shot mode, followed by Oscillator B in one-shot mode.
5	Oscillator A plays in one-shot mode, followed by Oscillator B in one-shot mode, followed by Oscillator A in one-shot mode, and so on.

`stereo`

This parameter sets the three bit “channel” output found inside the computer on the sound expansion connector. Normally the sound is unaffected by this parameter unless you are supporting hardware attached to the connector. Presently, several third-party cards use these bits for stereo positioning of the sound output. (LSB specifies right or left channel).

waveAddressA
waveSizeA
waveAddressB
waveSizeB



All the waves reside inside DOC memory, and the synthesizer needs to know the location and size of the wave in order to play it. The wave size parameter can specify a wave between 256 bytes and 32K in length. Although a full 64K is available for wave storage, only the high byte (page number) is entered for wave address, since the smallest wave is one page (256 bytes) in length. The wave address is highly dependent on the size of the wave. Certain sizes of waves can be stored only on certain boundaries in DOC RAM. Waves are aligned on their size boundaries. A 256 byte wave can reside in any of the 256 pages available in DOC RAM ($00, $01, $02...), a 512 byte wave can reside only on an even page ($00, $02, $04...), a 1K wave is stored on every 4th page boundary ($00, $04, $08...), and so on.

If you're using the metronome, you can't use pages $00 and $01, since the "wood block" wave used for metronome ticks is stored there by MIDI Synth.

volumeA
volumeB

This sets the output level for each oscillator in the generator. A value of 127 outputs the wave at full volume while a value of 0 turns the oscillator off.

```
octaveA
semitoneA
fineTuneA
octaveB
semitoneB
fineTuneB
```

Size Value	Byte Size	Wave Address Byte Range
0	256	xxxx xxxx
1	512	xxxx xxx0
2	1024	xxxx xx00
3	2048	xxxx x000
4	4096	xxxx 0000
5	8192	xxx0 0000
6	16,384	xx00 0000
7	32,768	x000 0000

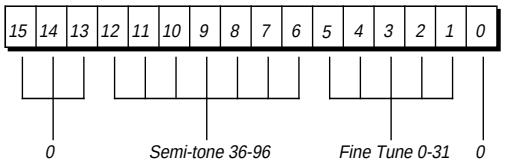
Wave Size and Address

These parameters adjust the base frequency for each oscillator. The octave and semitone parameters tune in increments of octaves and semitones, respectively, while the fine tune parameter tunes the oscillator in fractions of 1/64th of a semitone.

Note Tuning Table

The MIDI Synth Synthesizer plays notes tuned to the equal tempered scale. This tuning arrangement gives 12 balanced semitones per octave (each octave is double the frequency of the previous octave). This works well for keyboard instruments, and most popular western culture music is tuned to this scale; however, MIDI Synth is not limited to this one scale tuning arrangement. An application can customize their own note tuning to any historical, experimental or non-western culture scale available.

The synthesizer uses an internal look-up table with an entry word for each of the 128 MIDI note values. By sending a new table with the `SetTuningTable` call, the pitch of each MIDI note that the synthesizer plays can be defined by the application. Each entry word in the table defines a semitone and fractional semitone value. **Note that the semi-tone must be within the 36 to 96 value range.**



Note Tuning Word

Seq Record

The MIDI Synth sequencer records and plays MIDI events. These events are grouped and stored by the application in what is called a **Seq Record**. To play a sequence, the application passes the sequencer a pointer to where in memory the Seq Record is located. The Sequencer then examines

each MIDI event in the Seq Record and sends it to the Synthesizer or to the MIDI port. Because the Sequencer has to know in what order and when to play a MIDI event, each event has a time-stamp associated with it (see "Time Stamps" in next section). This time stamped MIDI event is call a **Seq Item**. The Seq Items are stored in increasing time stamp order. When playing, the sequencer increments its own internal clock. At every tick of the clock, the sequencer looks at the next Seq Item's time stamp. If the time stamp matches the internal clock, the Seq Item is used and the sequencer examines the next Seq Item. When an item's time stamp is at a value greater than the current internal clock, the sequencer does nothing with the item, waiting until the clock tick increases to match the item's value.

Seq Records have this format:

$00	*trackNumber*	**Byte**—$00-$0F
$01	*timeStampHigh*	**Byte**—$00-$FF
$02	*timeStampLow*	**Byte**—$00-$FF
$03	*timeStampMid*	**Byte**—$00-$FF
$04	*dataByteCount*	**Byte**—$00-$02
$05	*MIDIStatus*	**Byte**—$8x-$Fx
$06	*dataByte1*	**Byte**—$00-$7F
$07	*dataByte2*	**Byte**—$00-$7F

The **`trackNumber`** is a way of grouping Seq Items to some application defined reference. Sixteen tracks are available in the MIDI Synth Sequencer. Using the `SetPlayTrack` call, you can prevent MIDI messages with specified track values from being processed by the sequencer during playback. Likewise, when recording a sequence, the `SetRecTrack` call attaches this track reference value to all incoming MIDI Messages. **`dataByteCount`** specifies the number of bytes in the data byte field of the MIDI message. **`MIDIStatus`** contains MIDI status message (in the high nibble), and the MIDI channel number (in the low nibble). When playing the synthesizer, the MIDI channel field specifies which of the 16 synthesizer instruments gets the MIDI message. A Seq Item is always 8 bytes in length. If the MIDI message has less than 2 bytes, the unused data byte fields will have undefined values.

The end of a sequence must be marked with an $FFFF (**end-of-seq marker**) value. The buffer can be larger than the actual sequence, since an end of sequence marker will always halt playback. This has no effect on the record buffer. If you are playing and recording at the same time, playback will halt at the end marker, while recording will continue until instructed to stop by the `SeqPlayer` call. The sequencer will insert an end marker at the record buffer end when told to stop recording.

To record and playback simultaneously, two buffers are needed: one buffer is used to record the incoming data, while the second is used to playback a previously recorded sequence. It's the application's responsibility to merge the two buffers in correct increasing time-stamp order if both buffers need to be played together at a later time (see the `Merge` call for one way to do this). Likewise, if a certain track is to be re-recorded, the application must remove all Seq Items from the play buffer with the desired track number (see the `DeleteTrack` call for one way to do this).

All running status messages at the MIDI port are constructed back to full MIDI messages. The sequencer does not convert Seq Items to running status messages when sending them out the MIDI port.

Time Stamps

The term "time stamp" is somewhat misleading in that it really does not directly represent time. The sequencer is calibrated so that each tick of the clock represents a unit of 1/96th of a quarter

note. For example, if two Seq Items have a difference of 96 time-stamp units, they are separated from another by a value of a quarter note. So one time-stamp unit is a relative value representing a fraction of a beat. The tempo value is what converts these time-stamp ticks into “real” time by setting the sequencer clock to increment at a certain rate. If you double the tempo, the the clock will now increment twice as fast, which will half the time it takes to play a quarter note. By referencing all your Seq Items to 96 ticks per quarter note, tempos and beats will all work correctly.

If you’re not using tempos and beats in your application, and you need to work only with time, then setting the tempo (see `SetTempo` call) to a value of 125 beats per minute will give each clock tick a time value of approximately 5ms. This way if a Seq Item has a time-stamp of 100, for example, then it will play at about 1/2 second (100 x .005 = .5) from the beginning of the sequence.

Non-MIDI Messages

Besides storing MIDI Messages as Seq Items, the sequencer can also interpret special **sequencer commands**. Since the status field, when used for MIDI messages, always has the high bit set ($80-$FF), this field is extended downward ($00-$7F) to store these special commands. The `dataByte1` and `dataByte2` fields contain arguments for the specific commands.

The various commands are:

`SeqMarker` $00

A call to your `SMarker` callback routine is made whenever the sequencer encounters this command. You can put whatever you need as a reference in the data byte fields. This can be used to synchronize your application to a sequence.

`SetSeqBeat` $02

This command will force the number of clock ticks per beat to the value specified in the `dataByte1` (low) and `dataByte2` (high) fields. It does the same thing as a `SetBeat` call. This call is used when the meter must dynamically change within the sequence.

`SetRelTempo` $04

A signed word displacement value stored in the `dataByte1` (low) and `dataByte2` (high) fields will be added to the current tempo value. This call is used when the tempo must dynamically change within the sequence.

	Trk	Time Stamp			Cnt	St	D1	D2
1	08	00	00	00	02	95	3C	7F
2	00	00	60	00	02	9B	40	7F
3	08	00	20	01	02	85	3C	7F
4	00	00	20	01	02	8B	40	7F
5	FF	FF	FF	FF				

All values are in hex notation

4 Item Sequence

Shown here is a simple 4 item sequence which plays 2 notes. It is assumed that a quarter note is one beat.

1. On the first beat of the sequence, a middle C note is started. It will play instrument #6.

2. On the second beat, a middle E note is started. It will play instrument #12

3,4. On the 4th beat, both notes are turned off.

5. At the end marker ($FFFF), the sequencer stops playing.

Callback Routines

Since MIDI Synth is a control-orientated program, it behaves slightly different from most other tools written for the Apple IIGS. With non-control type tools, all actions are completed one at a time, in sequential order. For example, if you call the Dialog Manager to draw a certain dialog window, it will be drawn before control is returned to your program; QuickDraw will draw a circle at the time it is called, then it will return to your program, and so on. Outside events (key events and mouse movements) are queued and later pulled out and passed to your program only when you ask for them. This is because most tools run in the foreground with your application. You call a tool, it does what was requested, then returns to you. On the other hand, MIDI Synth behaves in a completely opposite way of foreground tools. Since it runs almost exclusively in the background (serviced every 5ms at DOC interrupt time), actions are not synchronized with your application program. A MIDI Synth tool call can start a whole series of actions, all running in the background, independent of your application. A `StartNote` call only queues the request and returns to your program; the note will not be started until the next 5ms interrupt comes along. In the meantime, many more calls to MIDI Synth could be made with no immediate result. But once started, envelopes are generated, DOC registers are updated, the sequencer is serviced and many more actions are all managed in the background, sharing CPU time with your application at intervals of 5ms. Outside MIDI events interrupt the CPU when they occur and are processed by MIDI Synth. From a programmer's point of view, this foreground-background type of processing gives the application a parallel structure. It acts as though there is a separate processor running MIDI Synth in parallel with your application.

Because of this independent parallel architecture, tool-to-program communication must be done somewhat differently. For example, when MIDI Synth needs to report a sequencer error message to your program, it's not going to happen when you make a `SeqPlayer` tool call, since the call returns to your program before it even started to play the sequence. Instead of forcing you to poll at regular intervals to collect messages, MIDI Synth dispatches messages through application defined "callback" vectors. These are application routines that MIDI Synth will call on specified events. You give it the address of your routine and MIDI Synth will call it when it needs to. If you don't need to use a specific callback, set its address in the callback table set to `NIL`.

Several points must be kept in mind when writing callback routines. They run at interrupt time, so severe restrictions are placed on your routines. You cannot make any tool or ROM calls, and you have virtually no stack space available. When returning, the stack pointer, data bank and direct page registers must all be restored to the values they had when your routine was called (you don't have to restore the X register, the Y register or the accumulator). If you plan on using any registers, it's up to you to initialize them correctly. You will get called in native mode with index and memory set at 16 bits wide, and you must return the same way. Exit the call back routine with an `RTL` instruction. Finally, you should never spend much time in a callback routine. Most of the time, all you should do is set a flag somewhere and exit. Let your foreground program examine

the flag, and then process whatever action needs to be taken. Stealing too much time during interrupt service only deteriorates overall performance.

The various callback routines are described in the following sections. You can use the `SetCallBack` call to actually set up the callback address table.

`EndSeq`

Called when the Sequencer encounters an end-of-seq marker during a play sequence. This subroutine does not get called if you're playing and recording, since the sequence ends only when you stop recording.

`UserMeter`

This routine gets called on every beat. The number of ticks per beat is specified by the value passed with the `SetBeat` call.

`Mstart, Mstop`

These get called when a MIDI Stop and MIDI Start message is received by the MIDI interface.

`PacketIn`

MIDI Synth calls this routine every time a complete MIDI message is received through the MIDI interface input port. The contents of the MIDI message can be found by your routine at the `Mpacket` locations in MIDI Synth's direct page (see `GetMSData`).

`SeqEvent`

This routine gets called every time the Sequencer processes a Seq Item. The MIDI message contained by the Seq Item can be found in the `SeqItem` variables in MIDI Synth direct page (see `GetMSData`). The track number can be found at direct page offset `SeqItemTrack`.

`SysEx`

Called while receiving a System Exclusive message. Every byte of the message, as it's received, will be passed to your routine in the accumulator. This includes the Status and EOX bytes ($F0 and $F7).

`PacketOut`

MIDI Synth will call this routine for every MIDI message it sends out the MIDI port. The contents of the MIDI message can be found by your routine at the `Mpacket` locations in MIDI Synth's direct page (see `GetMSData`).

`PgmChange`

This routine gets called whenever a MIDI Program Change message is received at the MIDI port. The accumulator contains the program value.

`SMarker`

The sequencer calls this routine whenever it encounters a `SeqMarker` command. The X register has the bank address and the accumulator has the remaining 16-bit address of the Seq Item with the `SeqMarker` command.

MIDI Synth Tool Set Housekeeping Calls

MSBootInit $0123

`MSBootInit` the MIDI Synth Tool Set. This call is made by the Tool Locator.

▲ **Warning** An application must never make this call. ▲

Parameters

The stack is not affected by this call.

Errors None.

C

```
extern pascal void MSBootInit();
```

MSStartUp $0223

`MSStartUp` is called by the application to start the MIDI Synth Tool Set. This call must be made by the application before any other calls are made to the MIDI Synth Tool Set.

Parameters

The stack is not affected by this call.

Errors	$2301	`msAlreadyStarted`	MIDI Synth already started
	$2303	`msNoDPMem`	Can't get direct page memory
	$2304	`msNoMemBlock`	Can't get memory block
	$2306	`msNoSoundTool`	Sound Tool Set not started

C

```
extern pascal void MSStartUp();
```

MSShutDown $0323

`MSShutDown` shuts down the MIDI Synth Tool Set. Any application that starts the MIDI Synth Tool Set with a call to `MSStartUp` must make this call to shut the tool set down. The call should be made after all other calls to the MIDI Synth Tool Set.

Parameters

The stack is not affected by this call.

Errors	$2302	`msNotStarted`	MIDI Synth never started.

C

```
extern pascal void MSShutDown();
```

MSVersion $0423

`MSVersion` returns the version number of MIDI Synth Tool Set.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
	<—SP

Stack after call

Previous contents	
versionInfo	**Word**—Version number
	<—SP

Errors None.

C `extern pascal Word MSVersion();`

`versionInfo` MIDI Synth Tool Set version number.

MSReset $0523

`MSReset` resets the MIDI Synth Tool Set. An application should never make this call.

Parameters

The stack is not affected by this call.

Errors None.

C `extern pascal void MSReset();`

MSStatus $0623

`MSStatus` indicates whether the MIDI Synth Tool Set is active.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
	<—SP

Stack after call

Previous contents	
activeFlag	**Word**—Tool status
	<—SP

Errors None.

C

```
extern pascal word MSStatus();
```

`activeFlag` The value returned is `TRUE` (non-zero) if the MIDI Synth Tool Set is active, and `FALSE` ($0000) if it is not active.

ConvertToMeasure $2123

`ConvertToMeasure` converts Seq time (Seq Clock ticks) to beats and measures.

Parameters

Stack before call

Previous contents	
Space1	**Word**—Space for result
Space2	**Word**—Space for result
Space3	**Word**—Space for result
ticks	**Word**—Ticks per beat (0-$FFFF)
beats	**Word**—Beats per measure (0-99)
— *clockTicks* —	**Long**—Seq Clock ticks
	<—SP

Stack after call

Previous contents	
remainder	**Word**—Remainder Seq Clock ticks
beatNumber	**Word**—Beat number (0-99)
measureNumber	**Word**—Measure number (0-999)
	<—SP

Errors

$2302	`msNotStarted`	MIDI Synth never started.
$230A	`msParamRangeErr`	Parameter range error.

C

```
extern MeasureRec ConvertToMeasure(ticks, beats, clockTicks);
Word ticks, beats;
Long clockTicks;
```

`ticks` Number of clock ticks in a beat.

`beats` Number of beats in a measure.

`clockTicks` Number of Seq Clock ticks; this is the value to convert.

`remainder` Number of clock ticks not making up a whole beat.

`beatNumber` Number of beats not making up a whole measure.

`measureNumber` Number of measures.

ConvertToTime $2023

`ConvertToTime` converts measures to Seq time (Seq Clock ticks).

Parameters

Stack before call

Previous contents	
— *Space* —	**Long**—Space for result
ticks	**Word**—Ticks per beat (0-$FFFF)
beats	**Word**—Beats per measure (0-99)
beatNumber	**Word**—Beat number (0-99)
measureNumber	**Word**—Measure number (0-999)
	<—SP

Stack after call

Previous contents	
— *clockTicks* —	**Long**—Seq Clock ticks
	<—SP

Errors	$2302	`msNotStarted`	MIDI Synth never started.
	$230A	`msParamRangeErr`	Parameter range error.

C

```
extern pascal Long ConvertToTime(ticks, beats, beatNumber,
   measureNumber);
Word ticks, beats, beatNumber, measureNumber;
```

`ticks` Number of Seq Clock ticks in a beat.

`beats` Number of beats in a measure.

`beatNumber` Number of beats.

`measureNumber` Number of measures.

DeleteTrack $1D23

`DeleteTrack` deletes all Seq Items from the specified track.

Parameters

Stack before call

Previous contents	
track	**Word**—Track number (0-15)
— *seqBufferPtr* —	**Long**—Pointer to the sequence
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $2302 `msNotStarted` MIDI Synth never started.

C

```
extern pascal void DeleteTrack(track, seqBufferPtr);
Word track;
Ptr seqBufferPtr;
```

`track` Track to delete.

`seqBufferPtr` Pointer to the Seq buffer to change.

GetMSData $1F23

`GetMSData` returns the location of MIDI Synth's direct page. This address is used primarily by callback routines to access data.

Parameters

Stack before call

Previous contents	
Space1	**Long**—Space for first result
Space2	**Long**—Space for second result
	<—SP

Stack after call

Previous contents	
reserved	**Long**—Reserved
DP	**Long**—Direct page address
	<—SP

Errors $2302 `msNotStarted` MIDI Synth never started.

C

```
extern GetMSDataOutputRec GetMSData();
```

Direct Page Offsets

Offset	Format	Name	Description
$0C	word	`MpacketStat`	MIDI input Status (low byte)
$0E	word	`MpacketData1`	MIDI input Data #1 (low byte)
$10	word	`MpacketData2`	MIDI input Data #2 (low byte)
$EC	byte	`PacketBytes`	Number of data bytes (0-2)
$12	byte	`SeqClockFrac`	Seq clock fraction
$13	long	`SeqClockInt`	Seq clock integer (high byte always zero)
$31	byte	`SeqItemStat`	Current Seq Item Status
$32	byte	`SeqItemData1`	Current Seq Item Data #1
$33	byte	`SeqItemData2`	Current Seq Item Data #2
$EA	byte	`SeqItemTrack`	Current Seq Item track number
$3F	byte	`MetroVol`	Metronome volume (0-$FF)
$E4	word	`MetroFreq`	Metronome frequency (0-$7FFF)

GetTuningTable $2523

`GetTuningTable` reads the MIDI note tuning table (see “Tuning Table,” earlier in this chapter, for details).

Parameters

Stack before call

Previous contents	
— *tablePtr* —	**Long**—Pointer to buffer
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $2302 `msNotStarted` MIDI Synth never started.

C

```
extern pascal void GetTuningTable(tablePtr)
Ptr tablePtr;
```

`tablePtr` Pointer to a 128 word buffer where the tuning table is stored.

InitMIDIDriver $2723

Before MIDI Synth can use MIDI, your application must pass it a MIDI driver. Any driver that works with the MIDI Tool Set should work with MIDI Synth. (See *Apple IIGS Toolbox Reference: Volume 3*, Chapter 38 for details on MIDI drivers).

Parameters

Stack before call

Previous contents	
slot	**Word**—Slot number (1-7)
internal	**Word**—Slot (use 1) or serial port (use 0)
userID	**Word**—User ID for Memory Manager
— *driverPtr* —	**Long**—Pointer to MIDI driver
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2302	`msNotStarted`	MIDI Synth never started.
$2311	`msDriverAlreadySet`	Driver already set.
$2380	`msDevNotAvail`	The requested device is not available.
$2381	`msDevSlotBusy`	The requested slot is already in use.
$2382	`msDevBusy`	The requested device is already in use.
$2383	`msDevOverrun`	Device overrun by incoming MIDI data.
$2384	`msDevNoConnect`	No connection to MIDI.
$2385	`msDevReadErr`	Framing error in received MIDI data.
$2386	`msDevVersion`	RIM version is incompatible with device driver.
$2387	`msDevIntHndlr`	Conflicting interrupt handler is installed.

C

```
extern pascal void InitMIDIDriver(slot, internal, userID,
   driverPtr);
Word slot, internal, userID;
ProcPtr driverPtr;
```

`slot` This tells the driver where the hardware for the MIDI interface is located. If you're using a SCC serial port interface then Printer = slot 1 and Modem = slot 2. For a card interface, slot number is one of the seven physical card slots inside the Apple IIGS.

`internal` If you're using an external MIDI interface that plugs into one of the serial ports, set this to zero. Otherwise, if the interface is a card that plugs into an internal slot, set this to a value of one.

`userID` User ID used by the driver to allocate memory. You can use the same ID used to load the MIDI driver.

`driverPtr` Your application must load the MIDI Driver from disk into memory. This is the memory location of where the driver was loaded.

KillAllNotes $0D23

`KillAllNotes` turns off all active synthesizer notes.

Parameters

The stack is not affected by this call.

Errors	$2302	`msNotStarted`	MIDI Synth never started.

C

```
extern pascal void KillAllNotes();
```

Locate $1123

`Locate` finds the address of the first Seq Item in a buffer with the specified time-stamp. If it can't find a match, it will return with the address of first item with a value greater than the specified time-stamp. If all Seq Items have time-stamps less than the given time, a pointer to the EOS (end-of-seq) will be returned.

This call can be used before a `SeqPlayer` call to find a starting location in the play buffer.

Parameters

Stack before call

Previous contents	
— *Space* —	**Long**—Space for result
— *time* —	**Long**—Time-stamp to match
— *seqBufferPtr* —	**Long**—Pointer to Seq buffer
	<—SP

Stack after call

Previous contents	
— *seqItem* —	**Long**—Pointer to Seq Item
	<—SP

Errors	$2302	`msNotStarted`	MIDI Synth never started.
	$230A	`msParamRangeErr`	Parameter range error.

C

```
extern pascal SeqItemRecPtr Locate(time, seqBufferPtr);
Long time;
Ptr seqBufferPtr;
```

`time` Time to match. The time is given in Seq Clock ticks.

`seqBufferPtr` Pointer to the Seq buffer to search.

`LocateEnd` $1B23

`LocateEnd` finds the end of a sequence. This call returns a pointer to the byte following the EOS (end-of-seq) marker.

Parameters

Stack before call

Previous contents	
— *Space* —	**Long**—Space for result
— *seqBufferPtr* —	**Long**—Pointer to sequence
	<—SP

Stack after call

Previous contents	
— *seqItem* —	**Long**—Pointer to location following the EOS marker
	<—SP

Errors $2302 `msNotStarted` MIDI Synth never started.

C

```
extern pascal Ptr LocateEnd(seqBufferPtr);
Ptr seqBufferPtr;
```

`seqBufferPtr` Pointer to the Seq buffer to search.

Merge $1C23

Merge two Seq Buffers together. Buffer 1 is merged with Buffer 2, with the combined result left in Buffer 2. Make sure that enough space is available beyond Buffer 2 to hold the added Items from Buffer 1.

The buffers are merged so that all Seq Items from both buffers are organized with increasing time-stamp order.

This call can be used at the end of a `SeqPlayer` record function to merge the record buffer with the existing play buffer to form a new play buffer that includes all the newly acquired Seq Items.

Parameters

Stack before call

Previous contents	
— *buffer1Ptr* —	**Long**—Pointer to Buffer 1
— *buffer2Ptr* —	**Long**—Pointer to Buffer 2
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $2302 `msNotStarted` MIDI Synth never started.

C

```
extern pascal void Merge(buffer1Ptr, buffer2Ptr);
Ptr buffer1Ptr, buffer2Ptr;
```

`buffer1Ptr` Pointer to the first Seq buffer.

`buffer2Ptr` Pointer to the second Seq buffer. The buffers are merged and placed in this buffer, so the buffer must be large enough to hold the contents of both buffers.

MIDIMessage $1A23

`MIDIMessage` sends a MIDI Channel Message to the synthesizer, sequencer or MIDI port.

Note See "Non-MIDI Messages," earlier in this chapter, for a discussion of MIDI messages.

Parameters

Stack before call

Previous contents	
destination	**Word**—Destination (0-2)
length	**Word**—Number of valid data bytes (0-2)
status	**Word**—MIDI Status Message ($8x - Ex)
data1	**Word**—Data byte #1 (0-127)
data2	**Word**—Data byte #2 (0-127)
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2302	`msNotStarted`	MIDI Synth never started.
$230A	`msParamRangeErr`	Parameter range error.
$230B	`msMsgQueueFull`	Message queue full.
$230D	`msOutputDisabled`	MIDI output disabled.
$230F	`msOutputBufFull`	MIDI output buffer is full.

C

```
extern pascal void MIDIMessage(destination, length, status, data1,
   data2);
Word destination, length, status, data1, data2;
```

`destination` Specifies where the message goes in MIDI Synth. Use one of these values:

Value	Meaning
0	Message plays only the synthesizer
1	Message plays the synthesizer and will also get recorded if the sequencer is currently recording MIDI data
2	Send the message out the MIDI port

`length` Each message has a message identifier byte, passed in `status`. Depending on the message, there may be 0, 1 or 2 data bytes. This parameter tells how many data bytes are in the message; the data bytes themselves are passed in `data1` and `data2`.

`status` MIDI status message number.

`data1` First data byte.

`data2` Second data byte.

MSResume $2323

`MSResume` enables MIDI Synth update interrupts. See the section “Interrupts,” earlier in this chapter, for details.

Parameters

The stack is not affected by this call.

Errors	$2302	`msNotStarted`	MIDI Synth never started.

C

```
extern pascal void MSResume();
```

MSSuspend $2223

`MSSuspend` disables MIDI Synth update interrupts. See the section “Interrupts,” earlier in this chapter, for details.

Parameters

The stack is not affected by this call.

Errors	$2302	`msNotStarted`	MIDI Synth never started.

C

```
extern pascal void MSSuspend();
```

PlayNote $0B23

`PlayNote` start playing a note with the specified instrument, pitch and volume. The channel number specifies which one of the 16 Instruments to play.

Parameters

Stack before call

Previous contents	
channel	**Word**—Channel number (instrument 0-15)
note	**Word**—MIDI note number (0-127)
velocity	**Word**—Key velocity or volume value (0-127)
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2302	`msNotStarted`	MIDI Synth never started.
$230A	`msParamRangeErr`	Parameter range error.
$230B	`msMsgQueueFull`	Message queue full.

C

```
extern pascal void PlayNote(channel, note, velocity);
Word channel, note, velocity;
```

`channel` This is the number of one of the 16 instruments you have defined using the `SetInstrument` call.

`note` Number for the note to play.

`velocity` Velocity of the note.

RemoveMIDIDriver $2823

`RemoveMIDIDriver` shuts down the MIDI driver. Remember to dispose of any memory used by the driver after you make this call (memory was allocated with the ID passed in `InitMIDIDriver`). You don't need to make this call if you used `MSShutDown`, but you still will need to dispose of the memory.

Parameters

The stack is not affected by this call.

Errors

$2302	`msNotStarted`	MIDI Synth never started.
$2310	`msDriverNotStarted`	Driver not started.

C

```
extern pascal void RemoveMIDIDriver();
```

SeqPlayer $1523

`SeqPlayer` starts and stops the sequencer play and record functions.

The sequencer always assumes that the sequence starts on a beat. All functions related to the beat will be synchronized to the first item in the sequence.

Careful attention must be given to the `theClock` value when playing or recording. It is the caller's responsibility to align the `PbufStart` pointer correctly into the play buffer so it points to a Seq Item which is time-stamped with a value equal or greater than `theClock`. If you wish to start at a specified Seq Clock time, use the result from a `Locate` call before any play or record to properly set `PbufStart`. See the example in the section "Using MIDI Synth," earlier in this chapter.

Parameters

Stack before call

Previous contents	
— *sequence* —	**Long**—Pointer to `SeqPlayRec`
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$2302	`msNotStarted`	MIDI Synth never started.
	$230A	`msParamRangeErr`	Parameter range error.

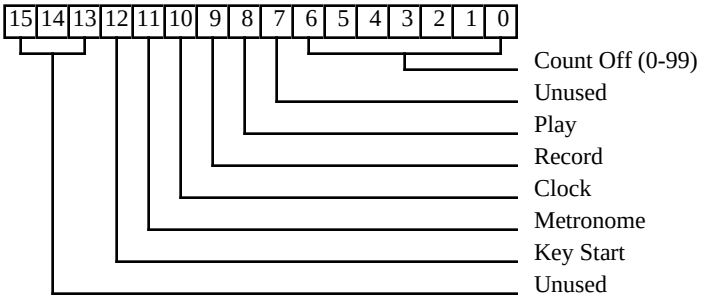
C

```
extern pascal void SeqPlayer(sequence);
SeqPlayRecPtr sequence;
```

`sequence` Pointer to a record with this format:

$00	— *PbufStart* —	**Long**—Pointer to play buffer
$04	— *Reserved* —	**Long**—Reserved, set to zero
$08	— *RbufStart* —	**Long**—Pointer to record buffer
$0C	— *RbufEnd* —	**Long**—Pointer to last byte of record buffer
$10	*SeqFlags*	**Word**—`SeqPlayer` flags
$12	— *theClock* —	**Long**—The Seq Clock gets set to this value

Here's the layout for the `SeqFlags` field:



Count Off	The Sequencer will wait the specified number of beats before starting. The beat callback (`Mupdate`) and metronome are still active during this period.
Play	Set to play, clear to stop playing.
Record	Set to record, clear to stop recording.
Clock	When this bit is set, the sequencer waits for MIDI Timing messages to advance the sequencer clock, synchronizing itself to an external MIDI device. If the bit is clear, the sequencer automatically advances the sequencer clock.
Metronome	If this bit is set, the sequencer sounds the internal metronome on every beat (see `SetBeat`). The sequencer assumes that the sequence always starts on a beat, so the metronome is heard immediately when the sequence starts.
Key Start	If this bit is set, the sequencer waits for a note off message before is starts playing or recording. The note off message that starts the sequencer is not recorded.

SetBasicChan $0923

`SetBasicChan` sets the MIDI Basic channel. This is the channel used while in Poly or Omni modes.

Parameters

Stack before call

Previous contents	
channel	**Word**—Channel number (0-15)
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$2302	`msNotStarted`	MIDI Synth never started.
	$230A	`msParamRangeErr`	Parameter range error.

C

```
extern pascal void SetBasicChan(channel);
Word channel;
```

`channel` MIDI Basic channel number.

SetBeat $1923

`SetBeat` sets the number of Seq Clock ticks per beat. This value is normalized to 96 ticks to a quarter note. This value is used to determine when to call the user routine for the `Beat` callback.

Parameters

Stack before call

Previous contents	
duration	**Word**—Beat duration
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$2302	`msNotStarted`	MIDI Synth never started.
	$230A	`msParamRangeErr`	Parameter range error.

C

```
extern pascal void SetBeat(duration);
Word duration;
```

`duration` Beat duration in Seq Clock ticks. Valid values are 1 to 65535.

SetCallBack $1723

`SetCallBack` sets the user callback vectors. Pointers with values of zero will disable the specific callback function. See the section "Callback Routines," earlier in this chapter, for details.

Parameters

Stack before call

Previous contents	
— *theRec* —	**Long**—Pointer to `callBackRec`
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $2302 `msNotStarted` MIDI Synth never started.

C

```
extern pascal void SetCallBack(theRec);
Ptr theRec;
```

`theRec` Call back record. The format for the record is:

$00	— *EndSeq* —	**Long**—Called at end-of-Sequence
$04	— *UserMeter* —	**Long**—Called on every beat
$08	— *Mstart* —	**Long**—Called when a MIDI 'Start' message is received
$0C	— *Mstop* —	**Long**—Called when a MIDI 'Stop' message is received
$10	— *PacketIn* —	**Long**—Called when any complete MIDI message is received
$14	— *SeqEvent* —	**Long**—Called when playing a Seq Item
$18	— *SysEx* —	**Long**—Called while receiving a System Exclusive Message
$1C	— *PacketOut* —	**Long**—Called when writing a MIDI message
$20	— *PgmChange* —	**Long**—Called when 'Program Change' message received
$24	— *Mcontinue* —	**Long**—Called when 'Continue' MIDI message received
$28	— *SMarker* —	**Long**—Called when playing a `SeqMarker`
$2C	— *RecBufFull* —	**Long**—Called when sequencer record buffer is full
$30	— *reserved1* —	**Long**—Reserved, set to $0000
$34	— *reserved2* —	**Long**—Reserved, set to $0000

SetInstrument $1423

`SetInstrument` sends an instrument record to MIDI Synth. An internal copy of the instrument record is kept by MIDI Synth, so after this call is made, you can dispose of the memory used by the instrument record.

Note Since MIDI Synth uses the internal copy to play an instrument, any changes made to your instrument record will not be used unless you make this call again with the new modified instrument record.

Parameters

Stack before call

Previous contents	
— *instrument* —	**Long**—Pointer to instrument record
number	**Word**—Instrument number (0-15)
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2302	`msNotStarted`	MIDI Synth never started.
$230A	`msParamRangeErr`	Parameter range error.

C

```
extern pascal void SetInstrument(instrument, number);
InstrumentRecPtr instrument;
Word number;
```

`instrument` Pointer to the new instrument record.

`number` MIDI Synth can store up to 16 instruments, numbered 0 to 15. This parameter tells MIDI Synth which of the 16 instruments should be defined or changed by this call.

SetMetro $1E23

`SetMetro` sets the metronome parameters. The default parameter value will be used if parameter value passed is zero. Metronome Wave must be 512 bytes in length.

Parameter	Default
`volume`	$FF
`frequency`	$0180
`wave`	Wood Block

Parameters

Stack before call

Previous contents	
volume	**Word**—Metronome volume (0-$FF)
frequency	**Word**—Metronome frequency (0-$FFFF)
— *wave* —	**Long**—Pointer to metronome wave
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2302	`msNotStarted`	MIDI Synth never started.
$230A	`msParamRangeErr`	Parameter range error.

C

```
extern pascal void SetMetro(volume, frequency, wave);
Word volume, frequency;
Ptr wave;
```

`volume` Metronome volume.

`frequency` Metronome frequency.

`wave` Pointer to the metronome wave; this is a 512 byte wave packet.

SetMIDIMode $0A23

`SetMIDIMode` controls how messages are sent from the MIDI port to the synthesizer. The three MIDI modes are:

Omni mode (0)

Accept messages on all MIDI channels and force them all to channel specified by the Basic Channel parameter. Messages from all channels will be played by only one Instrument since all channel numbers are changed to the Basic Channel value.

Poly mode (1)

Accept only those messages whose channels match the Basic Channel value. Ignore messages on all the remaining channels. Only one Instrument will be played since only one channel is active.

Multi mode (2)

Accept messages on all MIDI channels without modifying the channel number. Multiple Instruments can simultaneously play in Multi mode by sending MIDI messages through different channels. Each channel will play a different Instrument. The Basic Channel is ignored in Multi mode.

Parameters

Stack before call

Previous contents	
mode	**Word**—Omni, Poly or Multi mode (0-2)
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$2302	`msNotStarted`	MIDI Synth never started.
	$230A	`msParamRangeErr`	Parameter range error.

C

```
extern pascal void SetMIDIMode(mode);
Word mode;
```

`mode` MIDI mode.

SetMIDIPort $1323

`SetMIDIPort` lets you enable or disable MIDI input and output. If input is enabled, all MIDI messages will be sent to both the sequencer and synthesizer. If output is enabled, sequencer data will be sent out the specified port.

If you are not using MIDI output, make sure that it is disabled, since this will reduce CPU overhead.

MIDI will not work with AppleTalk enabled. Your program must check for this at start-up time.

Parameters

Stack before call

Previous contents	
enableInput	**Word**—$0001 to enable MIDI input; $0000 to disable input
enableOutput	**Word**—$0001 to enable MIDI output; $0000 to disable output
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2302	`msNotStarted`	MIDI Synth never started.
$2310	`msDriverNotStarted`	Driver not started.

C

```
extern pascal void SetMIDIPort(enableInput, enableOutput);
Word enableInput, enableOutput;
```

`enableInput` This is a boolean flag indicating whether or not MIDI input should be disabled. Use a value of $0001 to enable MIDI input, and a value of $0000 to disable MIDI input.

`enableOutput` This is a boolean flag indicating whether or not MIDI output should be disabled. Use a value of $0001 to enable MIDI output, and a value of $0000 to disable MIDI output.

SetPlayTrack $0F23

`SetPlayTrack` sets the state of the specified track for playback. `TRUE` (non-zero) makes the track active, while `FALSE` (0) turns the track off during playback. Any number or combination of inactive/active tracks can exists at any given time.

Parameters

Stack before call

Previous contents	
track	**Word**—Track number (0-15)
state	**Word**—Track play state
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2302	`msNotStarted`	MIDI Synth never started.
$230A	`msParamRangeErr`	Parameter range error.

C

```
extern pascal void SetPlayTrack(track, state);
Word track, state;
```

`track` Track to turn on or off.

`state` `TRUE` (non-zero) to turn the track on; `FALSE` (0) to turn the track off.

`SetRecTrack` $0E23

`SetRecTrack` sets which one of the 16 tracks is marked for recording. The `Track` field in all new Seq Items will be set to this value while recording.

Parameters

Stack before call

Previous contents	
track	**Word**—Track number
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2302	`msNotStarted`	MIDI Synth never started.
$230A	`msParamRangeErr`	Parameter range error.

C

```
extern pascal void SetRecTrack(track);
Word track;
```

`track` Track to record (0-15).

SetTempo $1623

`SetTempo` sets the sequencer tempo. Tempo values are specified in Beats Per Minute (BPM) minus 10. In other words, a value of zero gives the minimum tempo of 10 BPM while a value of 255 results in the maximum tempo of 265 BPM.

Parameters

Stack before call

Previous contents	
tempo	**Word**—Tempo in BPM-10 (0 - 255)
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$2302	`msNotStarted`	MIDI Synth never started.
	$230A	`msParamRangeErr`	Parameter range error.

C

```
extern pascal void SetTempo(tempo);
Word tempo;
```

`tempo` Tempo in BPM minus 10.

`SetTrackOut` $2623

`SetTrackOut` sets the Sequencer output path for the given track.

Parameters

Stack before call

Previous contents	
track	**Word**—Track number (0–15)
path	**Word**—Path value (0–2)
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2302	`msNotStarted`	MIDI Synth never started.
$230A	`msParamRangeErr`	Parameter range error.

C

```
extern pascal void SetTrackOut(track, path);
Word track, path;
```

`track` Sequencer track number.

`path` Path value; one of:

Value	Meaning
0	Send Seq output to both the Synthesizer and the MIDI port.
1	Send Seq output to only the MIDI port.
2	Send Seq output to only the Synthesizer.

SetTuningTable $2423

`SetTuningTable` sets up a customized MIDI note tuning table (see "Tuning Table," earlier in this chapter, for details). MIDI Synth makes an internal copy of this table, so after this call no further reference is made to the external table.

Parameters

Stack before call

Previous contents	
— *tablePtr* —	**Long**—Pointer to custom note tuning table
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $2302 `msNotStarted` MIDI Synth never started.

C

```
extern pascal void SetTuningTable(tablePtr);
Ptr tablePtr;
```

`tablePtr` Pointer to the 128 word tuning table.

SetVelComp $1223

`SetVelComp` increases the velocity value on all 'Note On' MIDI messages by the amount specified. This only affects messages received through the MIDI port.

Parameters

Stack before call

Previous contents	
offset	**Word**—Velocity offset (0-127)
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $2302 `msNotStarted` MIDI Synth never started.
$230A `msParamRangeErr` Parameter range error.

C

```
extern pascal void SetVelComp(offset);
Word offset;
```

`offset` The velocity is changed by this amount.

StopNote $0C23

`StopNote` stops playing a note started with `PlayNote`.

Parameters

Stack before call

Previous contents	
channel	**Word**—Channel number (instrument 0-15)
note	**Word**—MIDI note number (0-127)
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2302	`msNotStarted`	MIDI Synth never started.
$230A	`msParamRangeErr`	Parameter range error.
$230B	`msMsgQueueFull`	Message queue full.

C

```
extern pascal void StopNote(channel, note);
Word channel, note;
```

`channel` This is the number of one of the 16 instruments you have defined using the `SetInstrument` call.

`note` Number for the note to stop.

SysExOut $1823

`SysExOut` sends a MIDI System Exclusive message out the MIDI port. You must pass a pointer to a complete Sys Ex message, starting with the Status byte and terminating with an `EOX` byte. All data bytes in between must have their most significant bit set to zero.

Parameters

Stack before call

Previous contents	
messagePtr	**Long**—Pointer to Sys Ex message
delay	**Word**—Number of 5ms intervals between byte output
monitor	**Long**—Address of user Monitor routine
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2302	`msNotStarted`	MIDI Synth never started.
$230D	`msOutputDisabled`	MIDI output disabled.
$230E	`msMessageError`	Message error.

C

```
extern pascal void SysExOut(messagePtr, delay, monitor);
Ptr messagePtr;
Word delay;
ProcPtr monitor;
```

`messagePtr` Pointer to the Sys Ex message. This message starts with a byte with a value of $F0. This byte is followed by the message, which consists of any number of bytes; the most significant bit must be off for each of these bytes. The last byte in the message must be set to $F7.

`delay` The number of 5ms ticks to wait between writing bytes in the message. Since many MIDI devices will lose data if you send the message at full speed (`delay` = 0), this parameter should usually be set to a minimum value of 1 tick delay.

`monitor` When you make the `SysExOut` call, control won't return to you until the complete message is sent out. In the meantime, if you need to monitor the message output progress, you can pass the address of your monitor routine in this parameter. This routine will get called after each byte (except for the EOX) is sent. There are no parameters. Return in native mode via an `RTL` instruction.

Pass `NIL` if you do not want to use a monitor routine.

`TrackToChannel` $1023

`TrackToChannel` forces the specified sequencer track to play on a specific channel when playing to the synthesizer. Since each channel is assigned a different instrument, this essentially forces all Seq Items in a track to play the specified instrument.

A Channel number with a value of $FFFF will put the track in a "through" mode. This means that no channel translation will occur. Seq Items will be sent as messages to the synthesizer with their original channel numbers unaltered.

Parameters

Stack before call

Previous contents	
track	**Word**—Track number (0-15)
channel	**Word**—Channel number (0 - 15, $FFFF)
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$2302	`msNotStarted`	MIDI Synth never started.
	$230A	`msParamRangeErr`	Parameter range error.

C

```
extern pascal void TrackToChannel(track, channel);
Word track, channel;
```

`track` Sequencer track.

`channel` One of the 16 instruments established with a `SetInstrument` call, or $FFFF to "play through."

Chapter 15 **MIDI Tool Set Update**

The MIDI Tool Set has not changed. The original reference to this tool set is in Volume 3, Chapter 38 of the *Apple IIGS Toolbox Reference*.

Chapter 16 **Miscellaneous Tool Set Update**

This chapter contains new information about the Miscellaneous Tool Set. The original reference to this tool set is in Volume 1, Chapter 14 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 39 of the *Apple IIGS Toolbox Reference*.

New Features

- `SetVector` now behaves the same on ROM 1 machines and ROM 3 machines, behaving as documented in *Apple IIGS Toolbox Reference Manual: Volume 3*. (There is no error checking on the vector reference number.)

- `UnPackBytes` has been changed to fix to a rare case where it would treat bytes past the end of your source buffer as valid packed data. Because of this change, part of older versions of Apple IIGS Technical Note #94 is now obsolete.

- There are six new calls: `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.

ConvSeconds $3703

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

ConvSeconds 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 values for this parameter are:

0 Convert from seconds to the Miscellaneous Tools `ReadTimeHex` format.
1 Convert from the Miscellaneous Tools `ReadTimeHex` format to seconds.
2 Convert from seconds to the `ReadTimeASCII` format.
3 Not implemented.
4 Convert from seconds to ProDOS date/time format.
5 Convert from ProDOS date/time format to seconds.
6 Return the current time in seconds.
7 Set the current time using a value supplied in seconds.
8 Convert from ProDOS date/time format to the Miscellaneous Tools `ReadTimeHex` format.
9 Convert from the Miscellaneous Tools `ReadTimeHex` format to ProDOS date/time format.
10 Convert from seconds to HyperCard IIGS format.
11 Convert from HyperCard IIGS format to seconds.

Note The 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.

Note In previous documentation (including the 5.0.3 and 5.0.4 release notes) the values of verbs 8 and 9 were interchanged. The values shown here are correct.

Note `ConvSeconds` treats the ProDOS year numbers as documented in ProDOS 8 Technical Note #28. The year values 40 to 99 convert to 1940 to 1999, while the values 0 to 39 correspond to the years 2000 to 2039.

`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 be at least 40 bytes long when used as an ASCII output buffer. It must be at least 8 bytes long when used as an output buffer for any other format. `ConvSeconds` will always overwrite the first 8 bytes of the buffer, even if the value to be written is shorter than 8 bytes. ▲

`secondsOut` This return value is the number of seconds since January 1, 1904 for all of the conversions that return a value in seconds, as well as for the function that reads the current time (`convVerb` = 6). Conversions that do not return a value in seconds do not use this field, but the space must be present on the stack when `ConvSeconds` is called.

ScanDevices $3D03

`ScanDevices` provides an easy way to use the GS/OS system service vector that checks for disk insertions.

`ScanDevices` makes a device status call to each device that has removeable media except for 5.25" disk drives, which cannot be polled quickly. The devices are polled in ascending order. After each qualifying device is polled, `ScanDevices` returns the device number for the first device that reported an insertion.

If you only want to check for recent insertions, start by calling `ScanDevices` and ignoring the result. This forces the system to notice any old insertions, so the next call to `ScanDevices` will only report an insertion if an insertion occurs after the initial call.

As a side effect, `ScanDevices` causes GS/OS to call any notification procedures to inform them of insertions or ejections that are found during the `DStatus` 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` The GS/OS device number of the lowest numbered device reporting an insertion. A zero is returned if no insertions were found.

ShowBootInfo $3C03

`ShowBootInfo` provides a way for system extensions to make their presence known while the system is starting up. (For example, Control Panel NDA 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 for visual consistency.

`ShowBootInfo` displays the icon along the bottom of the Super Hi-Res screen (each icon appears farther to the right), or it displays the text string on the text screen. (Normally only the Super Hi-Res 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 the call is made from a program being installed after boot time, the icon will not interfere with an application's use of the desktop.

Tip 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. The format for the icon is the same as that used by `DrawIcon`. (See *Apple IIGS Toolbox Reference: Volume 2*, page 17-11 for a description of `DrawIcon`.) Pass `NIL` if you don't want to display an icon.

If bit 31 of this pointer is set, the icon overwrites the previous icon. (Pointers on the Apple IIGS use at most 24 bits of a 32 bit long word, so setting this bit does not interfere with the value of the pointer itself.)

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 occurs when 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
— *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
printableLength	**Word**—Number of printing characters in result
	<—SP

Errors $034F `mtBufferTooSmall` The buffer is too small.

C

```
extern pascal Long StringToText(flags, textPtr, textLen,
   resultPtr);
Word flags, textLen;
Ptr textPtr, resultPtr;
```

`flags`

bit 15: (`fAllowMouseText`) 1 = Allow MouseText in the result string; 0 = don't allow MouseText.
bit 14: (`fAllowLongerSubs`) 1 = Allow substitution of several characters for one character.
bit 13: (`fForceLanguage`) 1 = Use the language specified in bits 0-2; 0 = use the current display language.

bit 12: (`fPassThru`) 1 = Pass untranslated high-ASCII characters straight through instead of omitting them; 0 = omit untranslated high-ASCII characters.
bits 11-3: Reserved; set to 0.
bits 2-0: Result language. The value selects one of the following character sets:

0 USA English
1 U.K. English
2 French
3 Danish
4 Spanish
5 Italian
6 German
7 Swedish

`textPtr` Points to the input text. All 8 bits of the characters in the input text are significant. The character set is treated as identical to Shaston 8, which is also the same as the Macintosh standard character set. (See the table below.)

`textLen` Number of characters in the input text buffer.

`resultPtr` Points to the result buffer. This buffer has the same format as a GS/OS output buffer, namely two words followed by the text area. The first word is the total size of the available buffer, including the four bytes used by the initial two integers. The second word is filled in by `StringToText`; it is the actual number of characters returned in the result buffer. The characters returned follow this second integer.

`printableLength` This is the number of printable characters in the result buffer. If there are any MouseText characters in the result buffer this value will be smaller than the size returned in the result buffer.

`resultFlags` Bit 15 is set if any translations were performed, and clear if the result text is identical to the input text. Bits 14-0 are reserved, and should be ignored.

Note The interface files for high-level languages define `StringToText` as returning a single long integer result. `resultFlags` is in the most significant word, while `printableLength` is in the least significant word.

List of Translations

Character	ASCII Code	Becomes	Notes
	$11		Removed if MouseText is not allowed.
	$12		Removed if MouseText is not allowed.
	$12		Removed if MouseText is not allowed.
	$13		Removed if MouseText is not allowed.
#	$23	#	Removed for U.K., French, Spanish, Italian.
@	$40	@	Removed for French, Spanish, Italian, German.
[	$5B	[	“(” for French, Danish, Spanish, Italian, German, Swedish.
\	$5C	\	Removed for French, Danish, Spanish, Italian, German, Swedish.
]	$5D	]	“)” for French, Danish, Spanish, Italian, German, Swedish.
`	$60	`	“‘“ for Italian.
{	$7B	{	“(“ for French, Danish, Spanish, Italian, German, Swedish.
|	$7C	|	Removed for French, Danish, Spanish, Italian, German, Swedish.
}	$7D	}	“)” for French, Danish, Spanish, Italian, German, Swedish.
~	$7E	~	Removed for French, Italian, German.
Delete	$7F	Delete	Translated into the MouseText checkerboard character if MouseText is allowed; otherwise removed.
Ä	$80	A	“Ä” for German, Swedish.
Å	$81	A	“Å” for Danish, Swedish.
Ç	$82	C	
É	$83	E	
Ñ	$84	N	“Ñ” for Spanish.
Ö	$85	O	“Ö” for German, Swedish.
Ü	$86	U	“Ü” for German.
á	$87	a	
à	$88	a	“à” for French, Italian.
â	$89	a	
ä	$8A	a	“ä” for German, Swedish.
ã	$8B	a	
å	$8C	a	“å” for Danish, Swedish.
ç	$8D	c	“ç” for French, Spanish, Italian.
é	$8E	e	“é” for French, Italian.
è	$8F	e	“è” for French, Italian.
ê	$90	e	
ë	$91	e	
í	$92	i	“í” for Italian.
ì	$93	i	
î	$94	i	
ï	$95	i	
ñ	$96	n	“ñ” for Spanish.
ó	$97	o	“ó” for Italian.
ò	$98	o	
ô	$99	o	

Character	ASCII Code	Becomes	Notes
ö	$9A	o	“ö” for German, Swedish.
õ	$9B	o	
ú	$9C	u	
ù	$9D	u	“ù” for French, Italian.
û	$9E	u	
ü	$9F	u	“ü” for German.
†	$A0		Removed.
°	$A1	°	Removed for USA, U.K, Danish, German, Swedish.
¢	$A2	c	
£	$A3	£	Removed for USA, Danish, German, Swedish.
§	$A4	§	Removed for USA, U.K., Danish, Swedish.
•	$A5	*	
¶	$A6		Removed.
ß	$A7	ss	“ß” for German.
®	$A8	(R)	
©	$A9	(C)	
™	$AA	(TM)	
´	$AB	'	
¨	$AC	¨	Removed for all except French.
≠	$AD	<>	
Æ	$AE	AE	“Æ” for Danish.
Ø	$AF	0 (zero)	“Ø” for Danish.
∞	$B0		Removed.
±	$B1	+-	
≤	$B2	<=	
≥	$B3	>=	
¥	$B4	Y	
µ	$B5	u	
∂	$B6		Removed.
∑	$B7		Removed.
∏	$B8		Removed.
π	$B9		Removed.
∫	$BA		Removed.
ª	$BB	a	
º	$BC	o	
Ω	$BD	O	
æ	$BE	ae	“æ” for Danish.
ø	$BF	0 (zero)	“ø” for Danish.
¿	$C0	?	“¿” for Spanish.
¡	$C1	!	“¡” for Spanish.
¬	$C2		Removed.
√	$C3		Removed.
ƒ	$C4	f	
≈	$C5		Removed.
∆	$C6		Removed.
«	$C7	<<	
»	$C8	>>	
…	$C9	…	“...” (three periods) if MouseText is not allowed.
(space)	$CA	(space)	This is the nonbreaking space; it becomes a space character.

Character	ASCII Code	Becomes	Notes
À	$CB	A	
Ã	$CC	A	
Õ	$CD	O	
Œ	$CE	OE	
œ	$CF	oe	
–	$D0	-	
—	$D1	--	“-” if expansion is disabled.
“	$D2	"	
”	$D3	"	
‘	$D4	'	
’	$D5	'	
÷	$D6		Removed.
◊	$D7		Removed if MouseText is not allowed.
ÿ	$D8	y	
	$D9	Y	
	$DA	/	
	$DB	o	
	$DC	<	
	$DD	>	
	$DE	fi	
	$DF	fl	
	$E1	.	
	$E2	,	
	$E3	, ,	
	$E5	A	
	$E6	E	
	$E7	A	
	$E8	E	
	$E9	E	
	$EA	I	
	$EB	I	
	$EC	I	
	$ED	I	
	$EE	O	
	$EF	O	
	$F0		Removed if MouseText is not allowed.
	$F1	O	
	$F2	U	
	$F3	U	
	$F4	U	
	$F5	i	

SysBeep2 $3803

SysBeep2 takes an integer parameter and plays one of several sounds based on that parameter. These sound codes have predefined uses, and SysBeep2 is based on the SendRequest interprocess communications system, so it is possible to create custom actions to handle various error and alert conditions through the system.

SysBeep2 calls SendRequest with a requestCode of $0001 and the least significant word of dataIn set to bits 0 to 13 of the beepType parameter. Bit 31 of dataIn is a flag; if set, any request procedure receiving the call should return without making a sound. This makes it possible to put out a "feeler" to see if a given sound request will be handled or not, and whether your own request procedure will see it.

SysBeep2 first calls SendRequest and directs the request only to the Sound control panel. If the request is rejected, SysBeep2 then broadcasts a global request for any available request procedure to handle.

Note The toolbox installs a GS/OS notification procedure at system setup time. For certain events (shutdown, disk ejection, disk insertion, switch to or from ProDOS 8) this notification procedure calls SysBeep2 with codes in the range $8010 to $802F.

Parameters

Stack before call

Previous contents	
beepType	**Word**—Beep type
	<—SP

Stack after call

Previous contents	
	<—SP

Errors None.

C

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

beepType Bit 15 controls the default action. If this bit is clear and no request procedure handles the beep, the normal system beep plays. If this bit is set and no request procedure handles the beep, no sound is made.

Bit 14 delays the sound until a GetNextEvent call. If the bit is set, the beep is not posted until the next call to GetNextEvent that allows a keyDown, autoKey or mouseDown event; if the bit is clear, the beep is performed right away. This bit is ignored if the Event Manager has not been started.

If bit 14 is set, and there is already a sound waiting for the next call to `GetNextEvent`, the new call to `SysBeep2` is ignored. The original sound will still play when the `GetNextEvent` occurs, but the new sound is ignored completely. This has some useful side effects. For example, if you are about to call `AlertWindow` but wish to override the `SysBeep2` sound `AlertWindow` plays automatically, call `SysBeep2` yourself first to schedule a deferred sound. The deferred `SysBeep2` that `AlertWindow` executes has no effect.

The remaining fourteen bits are a sound code which tells why `SysBeep2` is being called. Sound codes are defined by Apple. If a value does not appear in this table or some later publication by Apple, it should not be used in a call to `SysBeep2`.

Value	Label	Use
$0000	`sbAlertStage0`	Alert stage 0 ($8000, setting bit 15, forces a default of 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 three times)
$0004	`sbOutsideWindow`	Click outside of a dialog or alert
$0005	`sbOperationComplete`	Attention – operation complete
$0006		Reserved
$0007		Reserved
$0008	`sbBadKeypress`	Bad key press
$0009	`sbBadInputValue`	Bad input value
$000A	`sbInputFieldFull`	Input field full
$000B	`sbOperationImpossible`	Operation impossible
$000C	`sbOperationFailed`	Operation failed
$000D-$0010		Reserved
$0011	`sbGSOStoP8`	Switch from GS/OS to ProDOS 8
$0012	`sbP8toGSOS`	Switch from ProDOS 8 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 insert request (example: `AlertWindow` with a disk swap icon)
$0031	`sbSystemStartup`	System start up
$0032	`sbSystemRestart`	Reserved for system restart
$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

Value	Label	Use
$0043	`sbEmptyTrash`	Emptying trash
$0050	`sbAlertWindow`	Attention – user response needed (example: `AlertWindow` with no icons)
$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-$0059		Reserved for `AlertWindow`
$0060	`sbScreenBlanking`	For screen dimmers
$0061	`sbScreenUnblanking`	For screen dimmers
$0100	`sbYouHaveMail`	User has mail
$0Exx	`sbErrorWindowBase`	Called by `ErrorWindow` for errors $0000..$00FF
$0EFF	`sbErrorWindowOther`	Called by `ErrorWindow` for errors $0100..$FFFF
$0Fxx		Reserved for assignment where a visual indication of the sound is appropriate.

VersionString $3903

`VersionString` converts a 32-bit version number into a Pascal string. The version string can be up to nine characters long; these characters and the leading length byte are placed in a ten-byte string buffer.

See the description of the new `rVersion` resource type in Appendix B of this book for a complete definition of the 32-bit version string, as well as examples of the strings returned by `VersionString` for various input values.

Parameters

Stack before call

Previous contents	
flags	**Word**—Reserved (pass 0)
— *theVersion* —	**Long**—The version number to convert
— *stringPtr* —	**Long**—Pointer to a 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;
```

`flags` This parameter is reserved for future use. It should be set to 0.

`theVersion` This is the version number to convert in the 32-bit long word format documented in this manual in the section that describes the `rVersion` resource type.

`stringPtr` `stringPtr` points to a ten-byte buffer. `VersionString` puts the Pascal string representing the version into this buffer. The string consists of the leading length byte and up to nine ASCII characters.

WaitUntil $3A03

`WaitUntil` provides an easy way to make sure that a program does not do things too quickly for the user. It waits for a specified period of time before returning. If the specified delay has already occurred, `WaitUntil` returns right away, and does not significantly impact system performance.

The time is specified as the number of 1/960ths of a second to wait between calls to `WaitUntil`. This works out to 1/16 of a system tick, giving finer control of time with less coding than is possible using the tick count returned by `GetNextEvent`.

Notes

- If interrupts are disabled, `WaitUntil` may return right away, so interrupts must be left enabled when using `WaitUntil`.
- The timing accuracy is only guaranteed to plus or minus one tick with System Software 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` value is multiplied by one less than the Battery RAM value, but the value is reduced to $F000 if the result of the multiply exceeds $F000. Note that a value of $01 multiplies the delay by zero, eliminating the delay.
- Scroll bar controls call `WaitUntil` when their value changes, as does `HiliteControl`. See the Control Manager update in Chapter 3 of this book for details.

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 for 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 from which to wait. The first time `WaitUntil` is called, use zero. For subsequent calls in a series of `WaitUntil` calls, use the value returned by `WaitUntil` on the previous call.

`delayAmount` This is the minimum amount of time to wait before returning. If this amount of time has already expired since `delayFrom`, `WaitUntil` returns right away; if not, `WaitUntil` pauses until the proper amount of time has expired before returning.

The time is in units of 1/960th of a second, although accuracy is currently only guaranteed to 1/60th of a second.

For example, pass $0040 for 4 ticks.

`newTime` This is a new anchor. It should be passed as the `delayFrom` parameter on the next call to `WaitUntil`. Use 0 if there is no new time.

Chapter 17 **Note Sequencer Update**

The Note Sequencer has not changed. The original reference to this tool set is in Volume 3, Chapter 40 of the *Apple IIGS Toolbox Reference*.

Chapter 18 **Note Synthesizer Update**

The Note Synthesizer has not changed. The original reference to this tool set is in Volume 3, Chapter 41 of the *Apple IIGS Toolbox Reference*.

Chapter 19 **Print Manager Update**

This chapter contains new information about the Print Manager. The original reference to this tool set is in Volume 1, Chapter 15 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 42 of the *Apple IIGS Toolbox Reference*.

New Features

- When the user boots from an AppleShare file server, the Print Manager now puts the Printer.Setup file inside the user's network folder, rather than trying to put it into `*:System:Drivers`. (The user doesn't always have access to `*:System:Drivers` on the network folder.) If the user has no network user folder, the `*:System:Drivers` is used, as before.

- Several Print Manager dialogs now use `AlertWindow`, saving disk and RAM space.

Chapter 20 **QuickDraw II Update**

This chapter contains new information about QuickDraw II. The original reference to this tool set is in Volume 2, Chapter 16 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 44 of the *Apple IIGS Toolbox Reference*.

New Features

- `QDStartUp` now uses a `masterSCB` bit (value $0100, bit 8) that avoids clearing the screen 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.

- `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 scan line 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.)

- `QDStartUp` now clears the bank $01 screen when shadowing is use on ROM 1 machines.

- `QDStartUp` now returns the "QD already started" error on ROM 3 machines if QuickDraw II has already been started.

- `QDShutDown` now checks to see if QuickDraw II Auxiliary is active, calling `QDAuxShutDown` if it is. This is needed because the Window Manager and the Standard File Operations Tool Set 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 beach ball cursor effect, without any annoying flicker.

To switch from one cursor in a sequence to the next, call `SetCursor` as usual, but 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 table consisting of 32 bytes of $00, followed by 32 bytes of $11, and so on, for a total of 16 tables of values. These values can be used as solid pen patterns in either 640 or 320 mode. In 640 mode, the tables for $00, $55, $AA and $FF give true solid colors, while the remaining tables give dithered patterns that appear as solid colors when they cover large areas.

See also `Set640Color`, which is easier to use if you intend to set the pen pattern.

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 pen pattern to any of the 16 solid colors.

Each of the 16 colors represents a true solid color in 320 mode, where `Set640Color` does the same thing as the `SetSolidPenPat` call.

In 640 mode the colors 0, 5, 10 and 15 are true solids, while the remaining colors are dithered patterns that appear solid when used to paint large areas of the screen. In some cases the correct dithered color appears with an area as small as two pixels wide.

While this call works in 320 mode, you can also use `SetSolidPenPat`, so this call is named for the case where it is most useful.

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;
```

`colorNum` A pen color number in the range 0 to 15.

Chapter 21 **QuickDraw II Auxiliary Update**

This chapter contains new information about QuickDraw II Auxiliary. The original reference to this tool set is in Volume 2, Chapter 17 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 44 of the *Apple IIGS Toolbox Reference*.

New Features

- FastPort features are now disabled during `DrawPicture`, so pen pattern changes (and other port parameters) work correctly when you start up QuickDraw with the `fastPort` bit set.
- `DrawPicture` no longer crashes when it encounters an invalid picture opcode; instead, it returns a $121F (`badPictureOpcode`) error.
- `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.)
- There are four new calls: `GetSysIcon` returns several predefined icons. `PixelMap2Rgn` constructs a region from a pixel map. `IBeamCursor` defines a new standard cursor. `WhooshRect` animates an opening or closing rectangle.

GetSysIcon $0F12

GetSysIcon returns small icons representing files, devices, and other miscellaneous items. 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
- Off-line 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` No icon is available for the given input.

C

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

`flags` Bits 15-3 are reserved and should be set to 0.
Set bit 2 for an open folder icon; clear it for a closed folder.
Bits 1-0 define the type of the icon:

00 File type icon (`value` is a GS/OS file type.)
01 Device icon (`value` is a GS/OS device ID and not a GS/OS device number.)
10 Miscellaneous icon (See table under `value` for the possibilities.)
11 Illegal value

`value` File type, device ID, or other value, depending on bits 1-0 of `flags`. The miscellaneous icons are:

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` This is the auxiliary type for a GS/OS file type.

IBeamCursor $1312

IBeamCursor sets the current QuickDraw II cursor to an I-beam cursor. The I-beam cursor is traditionally used when the cursor is over editable text.

The cursor comes from an rCursor resource in the system resource file. For 320 mode, the resource ID is $07FF0101; for 640 mode, the resource ID is $07FF0001.

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

PixelMap2Rgn transforms a pixel map into a QuickDraw II region. The points to be included in the region are specified by color.

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.

Note PixelMap2Rgn is stored in a dynamic segment, so the start-up disk may be needed on the first call.

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 indicating 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 are returned unchanged.

C

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

Note All of the coordinates in the `locInfo` record must be positive or zero. `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`.

`bitsPerPixel`
The number of bits per pixel. This will normally be 4 for 320 mode and 2 for 640 mode, but you can specify 4 in 640 mode to force `PixelMap2Rgn` to treat dithered colors as individual pixels. This parameter must have a value of 2 or 4.

`colorsToInclude`
A word of bit flags, one bit per color. Bit 0 is color zero, bit 1 is color 1, and so on. If a bit is set, pixels with a color corresponding to that bit are included in the region; if the bit is clear, the pixels are not included. Only bits 0 to 3 are used if `bitsPerPixel` is 2.

`theRgn` The handle to a new QuickDraw II region built from the pixel map.

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.

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
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 Set bit 31 to zoom out, from a small rectangle to a large rectangle. Clear this bit to zoom in.

Set bit 30 to use the local coordinates of the current grafPort; clear this bit for global coordinates.

Set bit 29 to cancel the default call to SysBeep2; leave this bit clear to call SysBeep2. SysBeep2 is called with a beepType of $8040 if bit 31 is set, and with a beepType of $8041 if bit 31 is clear.

Bits 28-0 are reserved, and should be set to 0.

Note If bit 30 is clear (global coordinates), rectangles are drawn in a port owned by the system with very little clipping. If bit 30 is set (local coordinates), the visRgn and clipRgn of the current port are used.

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

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

Note `smallRect` does not have to be smaller than `bigRect`.

Chapter 22 **Resource Manager Update**

This chapter contains new information about the Resource Manager. The original reference to this tool set is in Volume 3, Chapter 45 of the *Apple IIGS Toolbox Reference*.

New Features

- The Resource Manager now protects all open resource files from being accidentally closed by applications. (In System 5.0.4 and earlier, only the system resource file was protected.)
- `ResourceStartUp` now returns error $1E12, `resDupStartUp`, if the Resource Manager has already been started up for the specified memory ID.
- Four new calls have been added 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 the system resource file and actually have them preload. This feature is only available for the system software, though.
- If the resource map for the system resource file, 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 ID $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 $000000.)
- `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, file type, 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 system 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 want.) To ask for the value, set bit 31 of the `foundRec` pointer; the `foundRec` is then defined as follows:

$00	*resourceType*	**Word**—Type of resource
$02	*resourceID*	**Long**—ID of resource
$06	*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 disassociate a resource from a name.

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

New Resource Manager Calls

LoadResource2 $291E

`LoadResource2` works very much like `LoadResource`. The difference is that `LoadResource2` 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
	<—SP

Errors

$1E03	`resNoConverter`	No converter routine found for the resource type.
$1E06	`resNotFound`	The specified resource was not found.

GS/OS errors and Memory Manager errors are returned unchanged.

C

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

`flags` Reserved; this parameter must be 0.

`bufferPtr` Pointer to a one word output buffer. The previous attributes word of the handle is placed in this word. If the handle did not exist before this call, $FFFF is returned.

`resourceType` Resource type of the resource to load.

`resourceID` Resource ID of the resource to load.

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` resources, as described in *Apple IIGS Toolbox Reference Volume 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 where file number will be stored
	<—SP

Stack after call

Previous contents	
resourceID	**Long**—ID of resource found
	<—SP

Errors

$1E10	`resNameNotFound`	The named resource was not found.
$1E11	`resBadNameVers`	Bad version in `rResName` resource.

C

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

`rType` Resource type of the resource to load.

`namePtr` Resource name for the resource to find. The resource is stored as a Pascal string.

`fileNumPtr` Pointer to a one-word buffer. The file ID of the resource file containing the named resource is stored in this word.

`resourceID` Resource ID for the matching resource. The result is not defined if the resource is not found.

RMGetResourceName $2B1E

`RMGetResourceName` returns the name of the specified resource.

Resource names are stored in `rResName` resources. The format for this resource type is described in *Apple IIGS Toolbox Reference Volume 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 the receive name
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$1E10	`resNameNotFound`	The named resource was not found.
$1E11	`resBadNameVers`	Bad version in `rResName` resource.

C

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

`rType` Resource type of the resource to find.

`rID` Resource ID of the resource to find.

`namePtr` Pointer to a Pascal string buffer. The name of the resource is placed in this buffer. The buffer should be at least 256 characters long.

RMLoadNamedResource $2C1E

`RMLoadNamedResource` loads a resource, given the name and type of the resource. The current resource file and search depth are respected.

Resource names are stored in `rResName` resources. The format for this resource type is described in *Apple IIGS Toolbox Reference Volume 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 of the resource loaded
	<—SP

Errors

$1E10	`resNameNotFound`	The named resource was not found.
$1E11	`resBadNameVers`	Bad version in `rResName` resource.

GS/OS errors and Memory Manager errors are returned unchanged.

C

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

`rType` Resource type for the resource to load.

`namePtr` Pointer to the resource name for the resource to load. The resource name is a Pascal style string.

`resHandle` The handle for the resource loaded.

RMSetResourceName $2D1E

`RMSetResourceName` sets the name for a resource. If the name string has a length of zero, any existing resource name is removed. If an `rResName` resource becomes empty, the entire resource is removed.

Resource names are stored in `rResName` resources. The format for this resource type is described in *Apple IIGS Toolbox Reference Volume 3*, Appendix E.

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`	The named resource was not found.
$1E11	`resBadNameVers`	Bad version in `rResName` resource.
$1E13	`resInvalidTypeOrID`	The resource type or ID was not valid.

C

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

`rType` Resource type for the resource to load.

`rID` Resource ID for the resource to load.

`namePtr` Pointer to the new resource name. The resource name is a Pascal string.

Chapter 23 **SANE Tool Set Update**

This chapter contains new information about the SANE Tool Set. The original reference to this tool set is in Volume 2, Chapter 18 of the *Apple IIGS Toolbox Reference* and *Apple Numerics Manual*.

New Features

- `SANEVersion` now returns the same version number on ROM 1 and ROM 3 machines.

Chapter 24 **Scheduler Update**

This chapter contains new information about the Scheduler. The original reference to this tool set is in Volume 2, Chapter 19 of the *Apple IIGS Toolbox Reference*.

New Features

- The system now clears the Scheduler's private "don't dispatch" flag once at boot time, just as `SchBootInit` does on ROM 3 machines. This prevents a bug that could occur on ROM 1 machines, where crashing in a scheduled task would cause the Scheduler to stop dispatching tasks until a power-down or self-test. In this case, rebooting alone would not clear up the problem.

Chapter 25 **Scrap Manager Update**

This chapter contains new information about the Scrap Manager. The original reference to this tool set is in Volume 2, Chapter 20 of the *Apple IIGS Toolbox Reference*.

New Features

- The Scrap Manager now permits individual scraps to exceed 64K. Previously, you could create scraps this large, but `UnloadScrap` would not write them to disk properly.
- `ScrapStartUp` has been changed to zero out the Scrap State, because on ROM 3 `ScrapBootInit` gets called before 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 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

`GetIndScrap` allows utilities to get information about all scraps present, without knowing the scrap type in advance. This could be used, for example, in a Scrapbook NDA to copy or paste all scraps, rather than just a few predefined scrap types.

To get information about all of the scraps that exist, make calls to `GetIndScrap` with index values of 1, 2, 3, and so forth, 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 an index.

C

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

`index` Specifies which relative scrap to get. The first scrap has an index of 1; the index increments by 1 for each succeeding scrap.

`buffer` Points to a ten byte result buffer with this format:

$00	*scrapType*	**Word**—Scrap type
$02	— *scrapSize* —	**Long**—Scrap size
$06	— *scrapHandle* —	**Long**—Scrap handle

Chapter 26 **Sound Tool Set Update**

This chapter contains new information about the Sound Tool Set. The original reference to this tool set is in Volume 2, Chapter 21 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 47 of the *Apple IIGS Toolbox Reference*.

New Features

- `SoundVersion` now returns the same version number on ROM 1 and ROM 3 machines.

Chapter 27 **Standard File Operations Tool Set Update**

This chapter contains new information about the Standard File Operations Tool Set. The original reference to this tool set is in Volume 2, Chapter 22 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 48 of the *Apple IIGS Toolbox Reference*.

New Features

- When you insert a disk that is unformatted or contains an unrecognized file system, or which has the same name as an on-line volume with files open, Standard File calls `HandleDiskInsert` (see the Window Manager update in Chapter 32 of this book) 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 update in Chapter 10 of this book for more information on `ListKey`.) This means that you can select files by typing as many characters from the beginning of a file name as you wish. (Previously, typing a letter would always jump to the first file name starting with that letter.)

- In `PutFile` dialogs, the file list is highlighted with a bold outline when it is receiving keystrokes. 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.

  Command-Tab advances to the next disk, like Tab did in earlier versions.

- 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 `optionList`.

- Standard File now allows 128 volumes to be on line. Previously, it was limited to 20.

- There are two new key equivalents for the Volumes button, Command-D and Command-Esc.

- Standard File's icons come from `GetSysIcon`. (See QuickDraw II Auxiliary update in Chapter 21 of this book for information about `GetSysIcon`.)

- Long file names are drawn more narrowly to allow more of the characters to be seen. (Standard File now 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.
- `SFMultiGet2` would occasionally return error $1705 for no good reason; this problem has been fixed.
- A file name could occasionally be be duplicated many times in the file list; this problem has been fixed.

Chapter 28 **TextEdit Tool Set Update**

This chapter contains new information about the TextEdit Tool Set. The original reference to this tool set is in Volume 3, Chapter 49 of the *Apple IIGS Toolbox Reference*.

New Features

- In 5.0.3 & later versions of the system software, `TEGetText` uses bit 5 ($20) of the `bufferDesc` parameter for the `onlyGetSelection` bit. When this bit is set, `TEGetText` only returns the selected text, not the entire text record. This works for all data formats except `LETextBox2` format (because that would require special handling of the style information).

- `TEScroll` would sometimes scroll to the very end of the text instead of scrolling to a specified character position. The problem only appeared when the text was longer than 64K. The problem has been corrected.

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

- TextEdit controls no longer eat Command key presses they do not use.

Chapter 29 **Text Tool Set Update**

This chapter contains new information about the Text Tool Set. The original reference to this tool set is in Volume 2, Chapter 23 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 50 of the *Apple IIGS Toolbox Reference*.

New Features

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

Chapter 30 **Tool Locator Update**

This chapter contains new information about the Tool Locator. The original reference to this tool set is in Volume 2, Chapter 24 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 51 of the *Apple IIGS Toolbox Reference*.

Clarifications

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

New Features

Inter-process Communication

- Two new calls, `AcceptRequests` and `SendRequest`, 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 the Media Control Tool Set and MIDI Synth Tool Set, 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 start up visibly smooth. If you're starting QuickDraw II **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. It also makes `StartUpTools` work correctly with programs executed as shell application files (file type $B5, EXE) from shell environments.

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

- For both `StartUpTools` and `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` call. `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.

- The `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

- The way `StartUpTools`, `LoadTools`, and `LoadOneTool` examine bits 14 through 12 of version words has been changed. If one of these bits is off in the requested version, it is ignored in the actual version. See Apple IIGS Technical Note #100 for details about this and other forms of version numbers.

`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).

`UnloadOneTool`

- `UnloadOneTool` has been changed to return no error when unloading a tool that already wasn't there, but which had an entry in the default TPT. It usually returned error $00FE before.

Message Type $0011, Pathnames to Open or Print

- There is a new message type, $0011, for passing the pathnames of files to open or print. The new message uses GS/OS input pathnames, supplementing the older ProDOS pathname message. As with the ProDOS pathname message, 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:

$00	— *reserved* —	**Long**—Used by the system
$04	*type*	**Word**—Message type ($0011)
$06	*printFlag*	**Word**—$0000 = Open, $0001 = Print
$08	*path1*	**GS/OS input string**—Pathname of first file
$xx	*path2*	**GS/OS input strings**—Pathnames of any remaining files
$yy	*end*	**Word**—$0000 terminates list (This is also a null string pathname.)

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

New Tool Locator Calls

AcceptRequests $1B01

AcceptRequests and SendRequest provide a straight forward interprocess communications package. The system keeps a list of procedure pointers associated with all the processes that can receive requests.

AcceptRequests tells the system that you can accept requests. When you can no longer accept requests, call AcceptRequests with requestProc set to NIL using the same nameString and userID parameter that you passed originally to tell the system that you could accept requests. An alternate way to remove a request procedure is to pass nameString=NIL, requestProc=NIL, and a user ID. This removes all of the request procedures for the given user ID.

▲ **Warning** If you do not remove your request procedure before your application quits the system will crash. ▲

Only one AcceptRequests call will succeed for any particular 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 more than one request procedure in the same program, as long as they have separate names.)

Since requests can be sent to a select group of request procedures 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 ($8000-$FFFF) that apply to request procedures that match a given prefix string. (See SendRequest for details on the format of this string.)

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.

For debugging purposes only: The list of current request procedures is System Software 6.0 is kept in the Message Center, one named message per request procedure. Each message has a type greater than or equal to $8000, and has this form:

Pascal String	The string consists of a leading length byte, the character whose ordinal value is $FF, and the characters ">YourCompany~YourProduct~"
Word	user ID
Long	pointer to the request procedure

Parameters

Stack before call

Previous contents	
— *nameString* —	**Long**—Pointer to Pascal name string
userID	**Word**—User ID associated with this request procedure
— *requestProc* —	**Long**—Address of the request procedure to install
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$0113	`srqNameTooLong`	The name is too long; it must be 62 characters or less.
$0121	`srqDuplicateName`	The name has already been used.

C

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

`nameString` Request name prefix. See the notes in the discussion for the call, above. Note that your copy of the string does not have to stay around after the call.

`userID` Your user ID.

`requestProc` Address of your request procedure.

Parameters to a RequestProc

Stack before call

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
	<—SP

Stack after call

Previous contents	
result	**Word**—Bit 15 is set if the request was accepted
	<—SP

`result` If your procedure accepts the request, return $8000. To reject the request, return $0000. The result space is pre-initialized to zero. Bits 14-0 are reserved and must be 0.

`reqCode` The request code.

Codes $0000..$7FFF have global meaning. They are defined by Apple Computer.

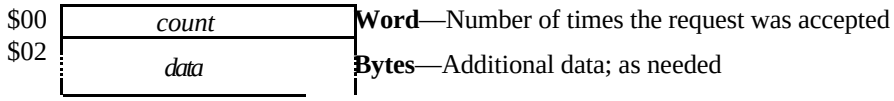
Codes $8000..$FFFF are defined separately for each prefix string. Combined with a unique prefix name for a given company and product, this allows unambiguous message handling by a program or group of programs.

The following global codes are currently defined:

$0001	`systemSaysBeep`	Sound (used by `SysBeep2`).
$0002	`systemSaysUnknownDisk`	See the Window Manager's description of `HandleDiskInsert` for details.
$0003	`srqGoAway`	See the discussion, below.
$0004	`srqGetrSoundSample`	Used by the Sound control panel.
$0005	`srqSynchronize`	Wait for asynchronous operations to complete, such as sounds played with `srqPlayrSoundSample`. `dataIn` and `dataOut` are reserved and should be 0.
$0006	`srqPlayrSoundSample`	Used by the Sound control panel. `dataIn` is an `rSoundSample` handle or a special value.
$0008	`systemSaysNewDeskMsg`	See the Window Manager for details.
$000E	`systemSaysEjectingDev`	Sent by `HandleDiskInsert`; low word of `dataIn` is the device number.
$0502	`systemSaysDeskStartUp`	`DeskStartUp` time. (`dataIn` and `dataOut` are reserved.)
$0503	`systemSaysDeskShutDown`	`DeskShutDown` time. (`dataIn` and `dataOut` are reserved.)
$051E	`systemSaysFixedAppleMenu`	Sent 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 the Finder documentation in Chapter 34 of this book.)

`dataIn` This parameter can be used as input data or as a pointer to an input data buffer.

`dataOut` This parameter is a pointer to an output data buffer. The output buffer consists of a count word, optionally followed by additional data to be filled in or modified by the request procedure. See `SendRequest` for a complete description of this count word and how it is used.



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.

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 user ID. `dataIn` is reserved, and should be passed zero and ignored by accepting procedure until some bits are defined. The `dataOut` buffer has the following format:

$00	*recvCount*	**Word**—Filled in by `SendRequest`
$02	*resultID*	**Word**—Filled in by the accepting procedure
$04	*resultFlags*	**Word**—Filled in by the accepting procedure

`resultID` is the user ID to be used with `AcceptRequests` and `UserShutDown`, or zero if the accepting procedure refuses to go away. It is filled in by the procedure that accepts the request.

`resultFlags` is also filled in by the requesting procedure. It is a bitmapped flag word. Bit 15 is set if the restart flag should be used for the shutdown, and clear if not. The remaining bits are reserved and should be zero.

Note Your code must always accept duplicate `srqGoAway` messages without harm or must always accept the first `srqGoAway`.

For example, a Finder extension must either tolerate multiple `finderSaysGoodbye` and `srqGoAway` requests without problems, or must always accept `srqGoAway`.

Sample Request Procedure in Assembly

```
SampleReqProc   start

oldPage         equ   1
rtl1            equ   3
dataOut         equ   6
dataIn          equ   10
request         equ   14
result          equ   16

                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
                    end
```

Installing a request procedure from assembly:

```
                    pea   ^myNameString
                    pea   myNameString
                    pha
                    _MMStartUp
                    pea   ^mySampleReqProc
                    pea   mySampleReqProc
                    _AcceptRequests
                    ...

myNameString        dw    'YourCompany~YourProduct~'
```

Note `dw` is a macro from the ORCA/M macro library. It defines a Pascal string.

Removing a request procedure from assembly:

```
                    pea   ^myNameString
                    pea   myNameString
                    pha
                    _MMStartUp
                    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("\pYourCompany~YourProduct~", MMStartUp(),
   MyRequestProc);
```

Removing a request procedure from C:

```
AcceptRequests("\pYourCompany~YourProduct~", 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));
   otherwise: ;
   end; {case}
if result <> 0 then
   result := $8000;
MyRequestProc := result;
end;
```

Installing a request procedure from Pascal:

```
AcceptRequests('YourCompany~YourProduct~', MMStartUp, @MyRequestProc);
```

Removing a request procedure from Pascal:

```
AcceptRequests('YourCompany~YourProduct~', MMStartUp, NIL);
```

GetMsgHandle $1A01

`GetMsgHandle` returns the handle to a message in the Message Center. The handle will be `NIL` if an error occurred, and the handle to the system's copy of the message if no error occurred.

An application should not modify the message handle and should not assume the handle is locked (it isn't), but it can look at the information to find the data and the message type and contents. (The type word is at an offset of four bytes into the block.) If the type is $8000 or above, the message begins with a Pascal string name.

Tip To delete a message when you don't know the message type, use `GetMsgHandle` to find the message, then look four bytes into the data to find the message type. Pass this message type 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 The specified message was not found.

C

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

`flags` Bits 15-2 of the Flags word are reserved, and should be set to zero. Bits 0 and 1 specify the type of reference in the `messageRef` parameter:

- 00 Get the nth message, where `messageRef` = n and the messages are counted from 1.
- 01 Get the message with the message type matching the least significant word of `messageRef`.
- 10 Get the message with the name `messageRef` points to. The name is a Pascal string.
- 11 Reserved.

`messageRef` The message to fetch. The exact meaning of this parameter depends on the setting of the `flags` parameter. See its description for the various meanings for `messageRef`.

`theHandle` Handle for the message.

SendRequest $1C01

`SendRequest` sends a request to a request procedure. Along with `AcceptRequests`, `SendRequest` implements an interprocess communications package.

When a `SendRequest` call is made, the system checks to see if there are any request procedures installed that can handle the request. (Request procedures are installed using `AcceptRequests`.) The `sendHow` and `target` parameters, taken together, control which request procedures are called.

Calling the request procedures is a **synchronous** operation. SendRequest does all its work before returning to the caller.

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` and `target`.

C

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

`reqCode` Passed to all qualifying request procedures.

`sendHow` The `sendHow` parameter uses bits 15 and 1-0 to control how the other parameters are used:

- 15 A value of 1 causes the process of calling request procedures to stop after some request procedure accepts the request. If the bit is zero, all qualifying request procedures are called.
- 14-2 Reserved for future use. Set these bits to 0.
- 1-0 These bits control who gets the request.
  - 00 `sendToAll` Send requests to all request procedures. `target` is reserved and should be set to 0.
  - 01 `sendToName` Select qualifying request procedures based on the prefix of name. `target` is a pointer to a Pascal string.
  - 10 `sendToUserID` Select qualifying request procedures by user ID. Pass the user ID in the low word of `target`.
  - 11 Reserved.

`target` This specifies which request procedures should receive the request. The specific value varies, depending on the `sendHow` parameter. See the description of `sendHow`, above, for details.

`dataIn` Passed to all qualifying request procedures.

The `dataIn` parameter is defined separately for each request code. It can be any value, but is typically a pointer to a buffer or handle to a buffer. The lifetime and ownership of the data is defined separately for each request code.

`dataOut` Passed to all qualifying request procedures.

The `dataOut` parameter is a pointer to a data buffer. The data buffer starts with a word parameter that is the number of times the request is accepted. This count word is filled in by the system after all request procedures have been called, not by the request procedures themselves. Since the value is not filled in until after all request procedures have been called, it is not possible for a request procedure to examine this field to determine how many times, or even if, a request has already been handled. This count word is optionally followed by other information which varies depending on the request code.

Requests are offered to the various qualifying request procedures working from the most recently installed procedure to the oldest procedure. This allows the system to provide a default request procedure for certain actions, and then allow individual applications to install request procedures that can process a request before (or instead of) the system.

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 code is accepted, you may pass `NIL` for `dataOut`.

Chapter 31 **Video Overlay Tool Set**

This chapter contains information about the Video Overlay Tool Set. This tool set is not documented in volumes 1 to 3 of the *Apple IIGS Toolbox Reference*.

This chapter contains the information needed by an Apple IIGS programmer to write software for Apple's Apple II Video Overlay Card. Information about the card from a hardware developer's point of view, or from the point of view of an Apple II programmer, can be found in the *Apple II Video Overlay Card Developer Notes*. Information about installing and using the the Apple II Video Overlay Card can be found in *Apple II Video Overlay Card Owner's Guide*, which comes with the card.

New Features

- `VDGGStatus` used to crash when given a selector value of $11 (`LineInterupts`); this selector value now works correctly.

Terminology

This chapter uses quite a few abbreviations and specialized terms. Even if you're an experienced video developer, some of the terms may different from those you're accustomed to. This section provides a brief description of the most important terms you'll come across in this chapter.

aliasing: In computer graphics, a phenomenon that causes jaggies on angled lines. Aliasing is manifested in the Apple II Video Overlay Card during certain dissolve operations, making a graphic image appear "chopped up."

composite video: A single-channel video signal compatible with NTSC monitors, such as the AppleColor® monitor.

dissolve: The extent to which a video or graphics pixel is made transparent. Dissolve is accomplished during overlay by changing the mixture of the two display signals such that the signal being dissolved decreases while the other signal increases.

external video source: An NTSC video source connected to the NTSC IN connector on the Apple II Video Overlay Card.

FIFO (first in, first out): A circuit used to regulate the flow of data from one asynchronous system to another. The function of a FIFO circuit is analogous to that of a queue.

genlock: A condition in which the Apple II Video Overlay Card's graphics generator has been synchronized to the operating frequency of an external video signal.

graphics generator: The circuitry that does all of the work involved in displaying video and graphics for a computer or peripheral card. The graphics generator drives the display.

IRE: A standard unit of measurement for video signal amplitude. For purposes of comparison, 0 IRE is blanking level, 100 IRE is white reference level, 7.5 IRE is black reference level, and -40 IRE is synchronization level.

key: An electronic signal in the Apple II Video Overlay Card that selects between an external video source and the Mega II-VGC graphics generator.

Mega II: A proprietary Apple custom chip that is part of the Apple II Video Overlay Card's graphics generator.

NTSC (National Television System Committee): The organization that defines the standards and controls governing TV transmission and reception in the United States. NTSC video is the 525-line video standard used for broadcast television in the United States, Canada and Japan.

overlay: The process where one display signal is mixed with another discrete video signal to produce a single display signal. The combination of signals is displayed on one monitor to create an overlaid effect.

RGB: Red, green, blue; the three primary colors that are mixed additively to make all other colors. Apple II Video Overlay Card RGB video is a three-channel video signal compatible with RS-170 monitors, such as the AppleColor RGB monitor.

VGC (Video Graphics Controller): A proprietary Apple custom chip that is part of the Apple II Video Overlay Card's graphics generator.

About the Apple II Video Overlay Card

The Apple II Video Overlay Card is a video card providing the Apple IIe and Apple IIGS computers with NTSC-video genlock and overlay capabilities, as well as improved composite video output.

How Overlay Works

The Apple II Video Overlay Card superimposes the host Apple II system graphics (menus, text, and so on) onto NTSC video, creating a combined RGB or composite video signal. This is accomplished by controlling the extent to which any given display's color is made up of graphics or video. Because graphics are overlaid onto the video signal, they must be dissolved in order to reveal the video they are covering. The more you dissolve the color, the more you can see through to the underlying video. At 100% dissolve, the graphics are not visible (no overlay), while at 0% dissolve, only graphics are completely visible (full overlay).

Key Color

To determine where overlay will occur, one display color is selected as a key color. The key color has a separately defined dissolve value, and so may appear differently than all other display colors, which share a single dissolve value. The Apple II Video Overlay Card's graphics generator uses the key color as a reference to determine to what extent a pixel should be dissolved. If the pixel is any color other than the key color, the dissolve value for non-key colors is used. If the pixel is the key color, the key color dissolve value is used. This is very useful for creating windows, titles in videos, and similar effects.

Consider, for example, the Apple II Video Overlay Card's start-up settings for key color and dissolve levels: The key color is set to black, the key color dissolve value is set to 100% dissolve (no overlay), and the non-key color dissolve value is set to 0% (full overlay). Any graphics pixel

that is black is made completely transparent and will display as the underlying NTSC video. All other colors, however, are not mixed with the video at all, and so display the image created by the host computer.

Changing Operating Parameters

Once you have installed the Apple II Video Overlay Card, it is always on; it does not require any driver code to operate, and as long as you do not want to change the start-up settings for its operating parameters, you do not need to write a driver program or add driver code to an existing application. However, if you want to change the mixture of graphics and video overlay you must either write a separate driver, add a driver to your application, or use the VideoMix desk accessory.

Although the Apple II Video Overlay Card is always on, the overlay function can be turned on or off through an appropriate tool call.

Remember that, even while overlay is enabled, the Apple II Video Overlay Card performs its overlay function only if there is an external NTSC signal present at its input connector. Without an external signal, there is no video to overlay and the card will display only the computer's normal graphics output.

Controlling the Apple II Video Overlay Card

To control the NTSC overlay features, you must write an application that manages the Apple II Video Overlay Card's operating parameters through the Video Overlay Tool Set tool calls. This section gives an overview of the parameters and how the calls are used to change them.

▲ **Warning** The information in this chapter is subject to change without notice. Software that reads from, writes to, and controls the Apple II Video Overlay Card hardware registers directly will not be compatible with future hardware products that operate within the Apple II video architecture. ▲

Startup

At power-up or rest, the Apple II Video Overlay Card's operating parameters are set as follows:

- Genlock is enabled.
- Key color is enabled.
- Key color is black (0000,0000,0000 RGB).
- Dissolve value is set for 100% video (full overlay) on the key color, and 100% system graphics (no overlay) on all other colors.
- Enhanced key mode is disabled.
- Output setup is enabled.
- Blank interval source is set to external.

- Chroma-crosstalk filter is enabled.
- Auxiliary page is selected for display (normal).
- MAIN page linearization is disabled.
- VBL interrupts ate disabled.
- Interlace mode is disabled.
- Graphics generator Apple II bus interface is enabled.
- Line interrupts are disabled.

FIFO Operation

The FIFO circuits provide data buffering between the host Apple II CPU and the Apple II Video Overlay Card's graphics engine. Before the Apple II Video Overlay Card can display an external NTSC video signal on its monitor, it must achieve genlock. In order to achieve genlock, the Apple II Video Overlay Card must halt its clock circuits while it tries to match the incoming video sync signal, which means that it operates asynchronously from the CPU. Because the Apple II Video Overlay Card is not clocking data into its registers at the same rate such data is being sent to it, there must be some way of buffering the incoming data until the Apple II Video Overlay Card's clock circuits restart and received data can be loaded into the registers. The FIFO circuits perform all of the functions necessary to accomplish this asynchronous exchange.

The Apple II Video Overlay Card is designed so that the clock circuits are never halted longer than six scan-line times; the FIFO circuits can easily handle the amount of data received during this time period.

NTSC Video

Although the Apple II Video Overlay Card always generates a high-quality video signal, in some cases the signal may not conform to the details of the NTSC standard. In order to generate NTSC video that meets FCC standards, the Apple II Video Overlay Card must have achieved genlock to a legal NTSC-video source, the blanking interval must be set to external video (the default), the setup adder must be enabled, and the RGB colors used for the overlay graphics must be legal NTSC colors.

Note Not all RGB colors that can be generated by an Apple IIGS will convert to legal NTSC colors. Approximately 18% of the 4096 possible colors are not legal NTSC colors. Illegal NTSC colors are only a problem (generally) with video intended for broadcast, because of transmitter limitations and FCC rules. However, because NTSC monitors display illegal colors without difficulty and NTSC CTRs and VCRs record them accurately it is hard to determine which colors are illegal when creating videos with the Apple II Video Overlay Card and the Apple IIGS.

Illegal NTSC colors created on the Apple IIGS are not an issue unless you intend to broadcast directly from the output of the Apple IIGS. Professional quality processing amplifiers and

broadcasting equipment will attenuate any illegal NTSC colors into legal colors, but, of course, this will result in a slightly different NTSC output fro the RGB output seen on the Apple IIGS display.

Video Detection

Whenever NTSC video is present at the Apple II Video Overlay Card's input connector, the graphics engine sets the video detect state (`VDInStatus(vdVideoDetect)`) to `TRUE`. This allows your application to detect video on the Apple II Video Overlay Card as soon as it is applied to the NTSC input connector. The Apple II Video Overlay Card does its best to identify the presence of a video signal accurately, but certain noise patterns, such as snow received from a vacant TV channel, can fool the video detect circuit to believe video is present. Also, a very poor video signal (such as that received from a VCR in scan mode) may not reliably activate the video detect circuit.

Achieving Genlock

Before the Apple II Video Overlay Card can display NTSC video on its monitor, it must achieve genlock with the incoming NTSC signal. In order to start the synchronization process, both the video detect state (`VDInStatus(vdVideoDetect)`) and the genlock enable state (`VDInStatus(vdGenlock)`) must be `TRUE`. The video detect state is set to `TRUE` automatically by the graphics engine when the incoming video signal is first present at the NTSC input connector. The genlock enable state is enabled at start up, but if you have subsequently disabled it with the `VDInControl` call, you must be sure to enable it again.

Genlock is achieved gradually in order to prevent rolling on the output display monitor (both RGB and composite monitors). This makes is possible to lose and recover video without causing significant disturbances on the monitor. (An example for achieving smooth transitions when changing an incoming video frame or sequence is provided at the end of this section.) Despite this gradual genlock process, the Apple II Video Overlay Card will display the external video within approximately one second after the incoming video signal is present at the input connector. When the Apple II Video Overlay Card achieves genlock, the graphics engine sets the status of the genlocked state (`VDInStatus(vdGenlocked)`) to `TRUE`.

Achieving Overlay

Once video detection and genlock have been achieved, the Apple II Video Overlay Card is capable of performing overlay. To do so, the key color enable state (`VDKeyControl(vdKColorEnable)`) must be set to `TRUE` (its start-up value). Once this state is set, the Apple II Video Overlay Card is able to perform overlay. The actual behavior of overlay is now controlled by the key color, the dissolve value, and the enhanced dissolve mode. These parameters are set with the tool set routines `VDKeyControl`, `VDKeySetKDiss`, `VDKeySetNKDiss` and `VDKeySetKCol`.

Setting the Key Color

The key color is a single display color that has a unique dissolve value set through `VDKeySetKDiss`. Typically the key color dissolve value is different from the dissolve value of all other display colors (set through `VDKeySetNKDiss`). The key color is used as a reference by the Apple II Video Overlay Card's graphics engine to determine which dissolve value to use for each pixel when generating the overlay display image. If the color value of a pixel is equal to the key color, the key color dissolve value is used to determine the extent to which the overlay is

dissolved. If the color value of the pixel is anything else, the non-key color dissolve value is used (see the `VDKeySetNKDiss` routine).

The key color can be any of the 4096 colors supported in the Apple IIGS.

Note If the key color enable state (`VDKeyControl(vdKColorEnable)`) is disabled, all colors are dissolved using the non-key-color value; there is no key color in this case.

Setting the Dissolve Value

There are two dissolve values used by the Apple II Video Overlay Card's keyer circuit. One controls the dissolve for the key color (`VDKeySetKDiss`) and one controls the dissolve for all other colors (`VDKeySetNKDiss`). The Apple II Video Overlay Card implementation supports seven dissolve levels that can be applied to either value, but future implementations may support more dissolve levels. The seven dissolve levels are evenly distributed within the unsigned integer range (0-65535) of the two set dissolve calls. You may specify any dissolve level within that range and the Video Overlay Tool Set will round off to the closest dissolve level supported by the Apple II Video Overlay Card. In this way your application can benefit from more dissolve levels as the are supported in future video overlay implementations. The default dissolve-level settings are 65535 (100% video) for the key color and 0 (100% graphics) for the non-key colors. The dissolve levels supported by the Apple II Video Overlay Card are:

Value	Video	Graphics
0-1561	0%	100%
5462-16383	17%	83%
16384-27307	33%	76%
27308-38229	50%	50%
38230-49151	67%	33%
49152-60074	83%	17%
60075-65535	100%	0%

The Apple II Video Overlay Card dissolve mechanism is implemented on the RGB output through a rapid alternation between Apple II graphics and the external video. This rapid alternation can sometimes beat against an alternating pattern in the Apple II graphics and cause aliasing effects. In practice, however, this condition has usually been found to be unnoticeable. With enhanced dissolve mode (`VDKeyControl(vdKeyEnhDiss)`) enabled, the alternating pattern is inverted each frame, virtually eliminating any remaining aliasing.

Note Future video overlay implementations may utilize a different RGB dissolve mechanism, and enhanced dissolve mode may no longer apply. Enabling enhanced dissolve mode on these systems will have no effect. Use `VDGetFeatures(vdKeyEnhDiss)` to determine if the video system supports enhanced dissolve mode.

Although the dissolve behavior in the RGB and NTSC output signals is not quite liner (that is, each output signal naturally dissolves at a slightly different rate), the Apple II Video Overlay Card's keyer circuit corrects for differences so that reasonably comparable images are displayed on the RGB and NTSC outputs. If a non-standard dissolve method is used, either by writing directly to the Apple II Video Overlay Card I/O registers or by alternating the key color with non-key colors in the display (that is, dithering the key color), the NTSC video display dissolve values will be somewhat different from those of the RGB display. This occurs because the Apple II Video

Overlay Card cannot compensate for the non-linearity when it is not in control of the dissolve operation.

Interrupts

This section describes the Mega II-VGC, vertical blanking, and scan-line interrupts on the Apple II Video Overlay Card.

Mega II-VGC Interrupts

Mega II and VGC interrupts are not directly supported by the Apple II Video Overlay Card except for diagnostic purposes, and then only on an Apple IIGS. Even so, the latency in response to these interrupts is considerable (the FIFO circuits must be completely empty) so they are not very useful.

The Apple IIGS main logic board Mega II-VGC interrupts are not synchronous to the Apple II Video Overlay Card video and therefore should not be used for time synchronization with the Apple II Video Overlay Card video output.

△ **Important** Interrupts on the Apple IIGS main logic board Mega II-VGC must be used cautiously with line interrupts (see "Scan-line Interrupts," later in this chapter). △

Note that except for time synchronization and scan-line interrupt status, the state of the Apple II Video Overlay Card Mega II-VGC should be identical to the host Apple IIGS Mega II-VGC, which can be read directly.

Vertical Blanking (VBL) Interrupt

The Apple II Video Overlay Card video output runs asynchronously to the video output from the host Apple II. The Apple II Video Overlay Card provides a special interrupt (VBL interrupt) to enable the host Apple II to detect the Apple II Video Overlay Card vertical blanking. The VBL interrupt is sent to the start of the Apple II Video Overlay Card's vertical blanking interval when the VBL interrupt is enabled (`VDGGControl(vdVBLInterrupt, vdEnable)`). This interrupt can be identified by reading the VBL interrupt request status (`VDGGStatus(vdVBLIntRequest)`). If it is true, a VBL interrupt has been generated. It is cleared by setting `VDGGControl(vdClearVBLInt, vdNil)`. The vertical blanking state can be ascertained through polling. `VDOutStatus(vdVerticalBlank)` indicates whether the Apple II Video Overlay Card is currently in vertical blanking or active video intervals.

Synchronizing the Apple II Video Overlay Card and Apple II vertical blanking is useful when displaying animated Apple II graphics that you want to run in sequence with segments of the incoming Apple II Video Overlay Card video overlay. VBL synchronization prevents the tearing side effect that occurs when combining changing graphic images from the Apple II with the Apple II Video Overlay Card video overlay.

Scan-line Interrupts

VGC scan-line interrupts are supported by the Apple II Video Overlay Card, but special care must be taken to prevent the Apple II Video Overlay Card VGC interrupts from getting mixed up with the Apple IIGS VGC interrupts. This is accomplished by using the Video Overlay Tool Set routines to set up scan-line interrupts for the Apple II Video Overlay Card's graphics generator

and utilizing the standard VGC registers to set up 1-second interrupts for the Apple IIGS main logic board's VGC.

To enable VGC scan-line interrupts on the Apple II Video Overlay Card, call `VDGGControl(vdLineInterrupt, vdEnable)`. To clear the scan-line interrupts, call `VDGGControl(vdClearLineInt, vdNil)`. To disable scan-line interrupts, call `VDGGControl(vdLineInterrupt, vdDisable)`, and to poll scan-line interrupts, call `VDGGStatus(vdLineIntRequest)`.

To use the Apple IIGS main logic board VGC one-second interrupts, set up the VGC registers as described in the *Apple IIGS Toolbox Reference*, except before writing to the VGC interrupt register ($C023), call `VDGGControl(vdGGBus, vdEnable)`, and when done, call `VDGGControl(vdGGBus, vdDisable)`. Also, be certain not to change the VGC scan-line interrupt clear bit in $C023. This will prevent the Apple II Video Overlay Card VGC and Apple IIGS main logic board VGC interrupts from being mixed up.

Note The Apple IIGS VGC scan-line interrupts cannot be used while the Apple II Video Overlay Card scan-line interrupts are enabled. Also, scan-line interrupts may not be supported in future video overlay implementations. Issue `VDGetFeatures(vdLineInterrupt)` to determine if interrupts are supported on the system.

Graphics Generator Apple II Bus Interface Control

The Apple II bus interface to the Apple II Video Overlay Card graphics generator can be disabled so that it does not receive further writes or reads from the 65816. This can be useful, as it allows the Apple II main logic board hardware to be controlled for certain applications without disturbing the Apple II Video Overlay Card. This bus control has no effect on Video Overlay Card Tool Set routines; they go through normally.

The call `VDGGControl(vdGGBus, vdEnable)` disables reads and writes, while `VDGGControl(vdGGBus, vdDisable)` enables reads and writes.

Note Apple II bus interface control may not be supported in future video overlay implementations. Use the call `VDGetFeatures(vdGGBus)` to determine if bus control is supported on the system.

Interlace Mode

When interlace mode is enabled with the `VDGGControl(vdInterlaceMode, vdEnable)` call, the Apple II Video Overlay Card generates an interlaced video signal without being genlocked to an external video signal. If the interlace mode is not enabled, the Apple II Video Overlay Card generates a non-interlaced video signal (like the Apple IIGS without the card) when not genlocked. The default state is interlaced mode disabled. `vdRAMPageSel` determines whether 200 lines are displayed in 2 fields (auxiliary or main) or 400 lines are displayed in 2 fields (interlace). See "Double-Vertical Resolution," later in this chapter for details.

Selecting the RAM Page

You can select which of the RAM pages (main or auxiliary) the Apple II Video Overlay Card's graphics engine will use for its graphics RAM by setting the appropriate value for `vdRAMPageSel`

with the `VDGGControl` call. The default value is `vdAux`, for the page in auxiliary memory (normal page used for the Apple IIGS). The values and associated results for the `VDGGControl(vdRAMPageSel, ggCtrlValue)` call are:

ggCtrlValue	Page Selected
vdAux	Auxiliary (default)
vdMain	Main memory
vdInterlace	Auxiliary for vdField0 and Main for vdField1 (See "Double Vertical Resolution")

The `vdInterlace` value provides RAM for twice the vertical resolution of auxiliary or main modes by using both RAM pages on an alternating line basis. It is used only with double vertical resolution.

Double Vertical Resolution

The Apple II Video Overlay Card has a double vertical resolution mode that doubles the vertical scan-line count by using both RAM pages. In order to use the double vertical resolution mode, the Apple II Video Overlay Card must be in interlace mode and not genlocked, or must have achieved genlock with a NTSC-video signal that uses an odd number of scan lines per frame. Double vertical resolution is enabled with the call `VDGGControl(vdRAMPageSel, vdInterlace)`. The auxiliary page is used if this option is selected when Apple II Video Overlay Card is not genlocked and not in interlace mode. If the incoming NTSC signal uses an even number of lines per frame, the Apple II Video Overlay Card stays in normal mode and uses only one RAM page. (If the number of incoming scan lines divided by two is an even number, auxiliary page is used; otherwise, the main page is used.)

In double vertical resolution mode, the first scan line is held in the auxiliary bank, the second in the main bank, the third in the auxiliary bank, and so on until 400 active scan lines have been displayed.

Note When the Apple II Video Overlay Card displays scan lines from the main bank, logical-to-physical address translation is performed. Therefore, you must remember to load memory in a non linear fashion, similar to that used in the older Apple II computers. An alternative to this scheme is to enable the main page linearization state with `VDGGControl(vdMainPageLin, vdEnable)`. This creates a linearized map of the main memory page.

Dual-Output Displays

The graphics generator on the Apple II main logic board may be controlled separately from the graphics generator on the Apple II Video Overlay Card. This provides the facility to drive a second display monitor with different graphics – for example, a text only display used to interactively control graphics and overlay an RGB monitor. Two methods can be used to achieve this result.

To use the first method, use the call `VDGGControl(vdRAMPageSel, vdMain)` to select main RAM. Because the Apple II main logic board has no RAM page selection capability, it will continue to display auxiliary RAM, allowing for two separate displays with different graphics on each display monitor.

To use the second method, disable the Apple II Video Overlay Card graphics generator interface to the Apple II bus with the call `VDGGControl(vdGGBus, vdEnable)`. This allows the Apple II main logic board Mega II-VGC to be placed in a different display mode, which uses a different area of RAM than the Apple II Video Overlay Card display mode. Then, when the bus is enabled again with `VDGGControl(vdGGBus, vdDisable)`, two different displays are generated.

Note Dial graphics generators and bus interface control may not be supported in the future. Use `VDGetFeatures(vdGGBus)` and `VDGetFeatures(vdDualOut)` to determine if these features are supported in the system.

NTSC Output Filters

The chroma-crosstalk filter and chroma-channel switch provide video signal filtering for the Apple II Video Overlay Card's NTSC output. The chroma-crosstalk filter operates on the black-and-white portion of the video, while the chroma-channel switch affects color video.

The chroma-crosstalk filter is a special filter that eliminates virtually all color fringing (that is, rainbows on high-frequency edges) carried to the NTSC video from the Apple II system graphics. These visual artifacts are due to crosstalk from the chroma subcarrier onto the luma carrier of the video signal, hence the name chroma-crosstalk. The chroma-crosstalk filter precedes the RGB-NTSC converter and affects NTSC output only. You can turn it on and off using the `VDOutControl` command to set `vdOutChromaFltr` to either `vdEnable` (turning the filter on) or `vdDisable` (turning the filer off).

Note When enabled, the chroma-crosstalk filter reduces the sharpness of the Apple IIGS graphics on the composite output.

The chroma-crosstalk switch works by separating the color portion of the video out of the signal and either passing it (on) or deleting it (off), thus enabling or disabling color video. It is controlled by the monochrome/color register of the graphics generator.

Apple IIGS Monochrome/Color Register

The Apple IIGS monochrome/color register is used on an Apple IIGS to turn off the color burst on the composite video output, producing a black-and-white composite video signal and maximizing the band width. Unlike the Apple IIGS, the Apple II Video Overlay Card is responsible for the integrity of an external video signal and so cannot arbitrarily disable the composite output color burst and make that signal turn black and white. To provide a graphics image similar to what an Apple IIGS normally displays, the Apple II Video Overlay Card does, however, make the overlaying graphics turn black and white (albeit with no gain in band width) when the monochrome/color register is set to composite gray scale mode. Further, if no external video signal is present, the Apple II Video Overlay Card will disable the color burst, just like an Apple IIGS, because no external video will be disturbed.

If the monochrome/color register is set to color mode, the chroma-channel switch is on; otherwise, it is off. The chroma-crosstalk filter is left enabled unless disabled by the call `VDOutControl(vdOutChromaFltr, vdDisable)`. One exception is when the Apple II Video Overlay Card has not achieved genlock and the monochrome/color register is set to monochrome mode. In this case the chroma-crosstalk filter is disabled regardless of the state of

vdOutChromaFltr, and the chroma-channel switch is off, maximizing band width (for example, when generating sharp, 80 column text).

Text Monochrome Override

The Apple II switches the composite output from color mode to monochrome mode when text mode is selected. This is done to prevent color fringing (caused by chroma-crosstalk). The Apple II Video Overlay Card, however, has a chroma-crosstalk filter that filters out unwanted color artifacts without killing the color of the text. By enabling text monochrome override with the call VDOutControl(vdTextMonoOver, vdEnable) the Apple II Video Overlay Card displays color text modes on the composite monitor with the same appearance as RGB output, despite the fact that an Apple II would normally display text in monochrome.

Note Text monochrome override is only available when auxiliary RAM is selected. To avoid temporary display disturbance, only enable text monochrome override while in text mode with auxiliary memory selected.

Blanking Source Select

When the Apple II Video Overlay Card is genlocked, it sends the blanking interval video from either the internal VGC graphics (vdGraphics) engine or the external video source (vdExternal), depending on the state of the external blanking source select (vdOutExtBlank). Although it is generally desirable to output the blanking video from the external source, conditions may exist where the external video blanking source is of poor quality and would be improved if output by the VGC graphics engine.

Note The blanking interval video is always output by the VGC when the Apple II Video Overlay Card is not genlocked.

Future implementations may not support external blanking source select, in which case blanking interval video from the graphics engine will always be output on the composite video output. If the system does not support external blanking source select, though, it would be expected to generate legal NTSC output, eliminating the need for the feature. The call VDGetFeatures(vdOutExtBlank) can be used to see if support for external blanking source select exists.

Color Tint and Saturation Adjustments

The Apple II Video Overlay Card's NTSC-to-RGB converter is equipped with programmable color hue (vdInHueAdj with the VDInConvAdj call) and color saturation (vdInSatAdj with the VDInConvAdj call) adjustments. These controls generally do not need to be changed, but from time to time poor video sources are used, and they need to be corrected. These adjustments have a 99-step range and can be increased (using the vdAdjInc parameter) or decreased (using vdAdjDec). Each adjustment is stored in nonvolatile RAM.

Note These adjustments only affect RGB output; they have no effect on composite output. The control values are write-only; there is no way to read the settings.

The nonvolatile RAM does have a limited life of approximately 10,000 storage cycles. The controls can be adjusted without storing each change. Allowing the user to adjust the settings without storing each new value, then saving the value once, will extend the life of the non-volatile RAM.

The current implementation of the video overlay using the Apple II Video Overlay Card suffers from a side effect in which switching from one adjustment to the other (hue to saturation or saturation to hue) causes an automatic save of the adjustment from which you have switched. This side effect may not be present in future implementations. Use `VDGetFeatures(vdAdjSideEffect)` to see if the side effect is present.

NTSC Setup Adder

Apple II's normal output black on the composite output at 0 IRE; however, most studio video equipment in use today places black at 7.5 IRE. If the Apple II Video Overlay Card video is merged with the video from a typical equipment source, the blacks will be noticeably different. Enabling the setup adder with `VDOutControl(vdOutSetup)` will move the Apple II Video Overlay Card black to 7.5 IRE. (This does not scale the value of white, which remains at 100 IRE.)

Achieving Smooth Transitions with Incoming Video

The Apple II Video Overlay Card has built-in features that allow you to make smooth transitions when changing video frames in your application. For example, suppose you have an interactive application that uses a video disc. The application allows the user to randomly move from one video frame or sequence to another. When the user makes a new selection, the video disc drive has to seek the requested frame or start of the new sequence, reinsert the video signal, and begin play. The changes that take place during the abrupt insertion of video can cause the transitions from one video image or sequence to another to be very distorted.

To make this transition appear smoother the controlling application can use the Apple II Video Overlay Card's dissolve feature. While the video is stable and genlocked, dissolve from 100% video to 100% graphics. Then seek to the start of the requested video frame and monitor the video-in status. When the status indicates the video signal is present, dissolve from 100% graphics to 100% video. This gives a smooth transition between the video sequences.

See "Setting the Dissolve Value," earlier in this chapter, for details about setting the dissolve value.

Video Overlay Tool Set Housekeeping Routines

VDBootInit $0121

`VDBootInit` initializes the Video Overlay Tool Set; called only by the Tool Locator.

▲ **Warning** An application must never make this call. ▲

Parameters

The stack is not affected by this call.

Errors None.

C

```
extern pascal void VDBootInit();
```

VDStartUp $0221

`VDStartUp` starts up the Video Overlay Tool Set. This call must be made before any other calls to the tool set.

Parameters

The stack is not affected by this call.

Errors

$2110	`VDNoVideoDevice`	No video device present in the system.
$2111	`VDAlreadyStarted`	Video overlay tool set already started.

C

```
extern pascal void VDStartUp();
```

VDShutDown $0321

`VDShutDown` shuts down the Video Overlay Tool Set. If an application starts this tool, this call must be made after all other calls to the Video Overlay Tool Set.

Parameters

The stack is not affected by this call.

Errors None.

C

```
extern pascal void VDShutDown();
```

VDVersion $0421

`VDVersion` returns the Video Overlay Tool Set software release version number.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
	<—SP

Stack after call

Previous contents	
version	**Word**—Version number
	<—SP

Errors None.

C

```
extern pascal Word VDVersion();
```

VDReset $0521

`VDReset` resets the Video Overlay Tool Set.

▲ **Warning** An application must never make this call. ▲

Parameters

The stack is not affected by this call.

Errors None.

C

```
extern pascal void VDReset();
```

VDStatus $0621

`VDStatus` returns `TRUE` if the Video Overlay Tool Set is active, `FALSE` if not.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
	<—SP

Stack after call

Previous contents	
activeFlag	**Word**—$0000 if the tool is inactive, non-zero otherwise
	<—SP

Errors None.

C

```
extern pascal Word VDStatus();
```

VDGetFeatures $1B21

`VDGetFeatures` returns the available features of the current video system.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
selector	**Word**—Selects which feature to return information about
	<—SP

Stack after call

Previous contents	
VDFeature	**Word**—Feature information
	<—SP

Errors $2112 `VDInvalidSelector` An invalid selector was specified.

C

```
extern pascal Word VDGetFeatures(selector);
Word selector;
```

`selector` Used to select the type of information returned. The various values allowed for `selector` are shown below, along with the feature about which information is returned.

selector	VDFeature	Feature
vdVideoOverlay	vdAvail/vdNotAvail	Is the video overlay is available in the system?
vdFrameGrabber	vdAvail/vdNotAvail	Is a video frame grabber is available in the system?
vdInVStandards	InStandards	Video input standards supported by the system. See the table below for details.
vdOutVStandards	OutStandards	Video output standards supported by the system. See the table below for details.
vdKeyDissLevels	NumLevels	Number of dissolve values supported by the system for the key color.
vdNKeyDissLevels	NumLevels	Number of dissolve values supported by the system for non-key colors.
vdAdjSideEffect	vdYes/vdNo	Does the system have the inadvertent side effect of saving the input video adjustment when switching from one adjustment mode to another?
vdKeyColorBits	NumBits	Number of bits supported by the system for the key color.
vdInHueAdj	vdAvail/vdNotAvail	Does the system support input hue adjustment?
vdInSatAdj	vdAvail/vdNotAvail	Does the system support video input saturation adjustment?

vdInContrastAdj	vdAvail/vdNotAvail	Does the system support video input contrast adjustment?
vdInBrightAdj	vdAvail/vdNotAvail	Does the system support video input brightness adjustment?
vdOutSetup	vdAvail/vdNotAvail	Does the system support setup on the composite output?
vdOutChromaFltr	vdAvail/vdNotAvail	Does the system support chroma-crosstalk filtering?
vdOutExtBlank	vdAvail/vdNotAvail	Used to pass external video's blanking interval through a composite output.
vdKeyEnhDiss	vdAvail/vdNotAvail	Does the system support enhanced dissolve mode?
vdLineInterrupt	vdAvail/vdNotAvail	Does the system support scan line interrupts?
vdGGBus	vdAvail/vdNotAvail	Can the Apple II bus interface to the graphics generator be disabled?
vdDualOut	vdAvail/vdNotAvail	Are two graphics generators supported (one on the Apple IIGS mother board and one on the overlay card) for dual graphics display?
vdTextMonoOver	vdAvail/vdNotAvail	Does the system support text monochrome override?

Note Unimplemented feature selectors will always return `vdNotAvail`. Therefore, applications can always query for the existence of new features from Video Overlay Tool Set versions.

InStandards	Meaning
vdNone	Input not available
vdNTSC	NTSC video standard
vdPAL	PAL video standard
vdSECAM	SECAM video standard
vdSNTSC	SNTSC (Y/C) video standard
vdSPAL	SPAL (Y/C) video standard
vdSSECAM	SSECAM (Y/C) video standard
vdRGB60	RGB (60 Hz)
vdRGB50	RGB (50 Hz)

OutStandards	Meaning
vdNone	Output not available
vdNTSC	NTSC video standard
vdPAL	PAL video standard
vdSECAM	SECAM video standard
vdSNTSC	SNTSC (Y/C) video standard
vdSPAL	SPAL (Y/C) video standard
vdSSECAM	SSECAM (Y/C) video standard

Note InStandards and OutStandards are bit mapped values, so more than one standard may be indicated in a given value. For example, if a card supports both the PAL video standard and the SPAL (Y/C) video standard, but no others, the value returned will be `vdPAL` ored with `vdSPAL`.

VDGGControl $1D21

`VDGGControl` sets the graphics generator control values.

Parameters

Stack before call

Previous contents	
selector	**Word**—Selects the graphics generator control value to set
ggCtrlValue	**Word**—Graphics generator control value
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2112	`VDInvalidSelector`	An invalid selector was specified.
$2113	`VDInvalidParam`	An invalid parameter was specified.

C

```
extern pascal void VDGGControl(selector, ggCtrlValue);
Word selector, ggCtrlValue;
```

`selector` Selects the graphics generator control value to set.

`ggCtrlValue` Graphics generator control value.

selector	ggCtrlValue	Meaning
`vdMainPageLin`	`vdEnable` or `vdDisable`	Main memory page linearize
`vdRAMPageSel`	`vdAux` or `vdMain` or `vdInterlace`	RAM page select
`vdVBLInterrupt`	`vdEnable` or `vdDisable`	VBL Interrupt
`vdClearVBLInt`	`0`	Clear the VBL interrupt
`vdLineInterrupt`	`vdEnable` or `vdDisable`	Scan line Interrupt
`vdClearLineInt`	`0`	Clear scan line Interrupt
`vdInterlaceMode`	`vdEnable` or `vdDisable`	Interlace mode enable
`vdGGBus`	`vdEnable` or `vdDisable`	Disables Apple II bus graphics generator interface

Note The line interrupt on the Apple II Video Overlay Card can only be used in special circumstances.

Line interrupts may not be supported in future video overlay implementations. Use `VDGetFeatures(vdLineInterrupt)` to see if the feature is available.

`vdGGBus` may not be supported in the future. Use `VDGetFeatures(vdGGBus)` to see if the feature is available.

VDGGStatus $1E21

`VDGGStatus` reads the current settings of the graphics generator control.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
selector	**Word**—Selects which status information to return
	<—SP

Stack after call

Previous contents	
ggCtrlValue	**Word**—Status information
	<—SP

Errors $2112 `VDInvalidSelector` An invalid selector was specified.

C `extern pascal Word VDGGStatus(selector);`

`selector` Selects which status information to return.

selector	ggCtrlValue	Meaning
vdMainPageLin	vdEnable or vdDisable	Main memory page linearize
vdRAMPageSel	vdAux or vdMain or vdInterlace	RAM page select
vdVBLInterrupt	vdEnable or vdDisable	VBL Interrupt
vdLineInterrupt	vdEnable or vdDisable	Scan line Interrupt
vdInterlaceMode	vdEnable or vdDisable	Interlace mode enable
vdGGBus	vdEnable or vdDisable	Apple II bus graphics generator interface
vdDisplayField	vdField0 or vdField1	Currently displaying field
vdVBLIntRequest	vdTrue or vdFalse	VBL Interrupt request
vdLineIntRequest	vdTrue or vdFalse	Line Interrupt request

VDInControl $1C21

`VDInControl` sets the video input control values.

Parameters

Stack before call

Previous contents	
selector	**Word**—Selects the input control value to set
inCtrlValue	**Word**—The input control value
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2112	`VDInvalidSelector`	An invalid selector was specified.
$2113	`VDInvalidParam`	An invalid parameter was specified.

C

```
extern pascal void VDInControl(selector, inCtrlValue);
Word selector, inCtrlValue;
```

`selector` Selects the input control value to set. The only value currently allowed is `vdGenlock`.

`inCtrlValue` Pass `vdEnable` to enable genlock, or `vdDisable` to disable genlock.

VDInConvAdj $0C21

`VDInConvAdj` performs video input adjustment.

Parameters

Stack before call

Previous contents	
selector	**Word**—Selects what to adjust
adjFunction	**Word**—The function to perform
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2112	`VDInvalidSelector`	An invalid selector was specified.
$2113	`VDInvalidParam`	An invalid parameter was specified.

C

```
extern pascal void VDInConvAdj(selector, adjFunction);
Word selector, adjFunction;
```

`selector` Selects what to adjust. Possible values are:

`vdInHueAdj`	Selects hue adjustment
`vdInSatAdj`	Selects saturation adjustment

△ **Important** Switching selector from one mode to another forces a save to occur on some implementations, such as the Apple II Video Overlay Card. Saves should be kept to a minimum (on some implementations, the nonvolatile memory used to retain adjustment values has a limited number of save cycles before wearing out). It is recommended that saves be performed only once per full adjustment, not after every increment or decrement. △

`adjFunction` Selects the kind of adjustment. Possible values are:

`vdAdjInc`	Performs an increment function
`vdAdjDec`	Performs a decrement function
`vdAdjSave`	Performs a save function (saves the value to nonvolatile memory)

VDInGetStd $0B21

`VDInGetStd` gets the currently selected video input standard.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
	<—SP

Stack after call

Previous contents	
inStandard	**Word**—Video input standard
	<—SP

Errors None.

C `extern pascal Word VDInGetStd();`

`inStandard` See `VDInSetStd` for a list of the values that can be returned.

VDInSetStd $0A21

`VDInSetStd` sets the input video standard.

Parameters

Stack before call

Previous contents	
inStandard	**Word**—Video standard
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $2113 `VDInvalidParam` An invalid parameter was specified.

C

```
extern pascal void VDInSetStd(inStandard);
```

`inStandard` One of these values:

inStandard	Meaning
vdNone	Input not available
vdNTSC	NTSC video standard
vdPAL	PAL video standard
vdSECAM	SECAM video standard
vdSNTSC	SNTSC (Y/C) video standard
vdSPAL	SPAL (Y/C) video standard
vdSSECAM	SSECAM (Y/C) video standard
vdRGB60	RGB (60 Hz)
vdRGB50	RGB (50 Hz)

Note Call `VDGetFeatures` to determine which input standards are supported by the video system.

VDInStatus $0921

`VDInStatus` gets the current input status information.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
selector	**Word**—Selects which status information to return
	<—SP

Stack after call

Previous contents	
VDInStatus	**Word**—Status information
	<—SP

Errors $2112 `VDInvalidSelector` An invalid selector was specified.

C

```
extern pascal Word VDInStatus(selector);
Word selector;
```

`selector` Selects the kind of status information to return. The allowed values, and what can be returned, are:

selector	VDInStatus	Meaning
vdGenlock	vdEnable or vdDisable	genlock control
vdVideoDetect	vdTrue or vdFalse	video detected or not
vdGenlocked	vdTrue or vdFalse	genlocked or not

VDKeyControl $0D21

`VDKeyControl` performs keyer control functions.

Parameters

Stack before call

Previous contents	
selector	**Word**—Selects the video keyer control to set
keyerCtrlVal	**Word**—The keyer control value
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2112	`VDInvalidSelector`	An invalid selector was specified.
$2113	`VDInvalidParam`	An invalid parameter was specified.

C

```
extern pascal void VDKeyControl(selector, keyerCtrlVal);
Word selector, keyerCtrlVal;
```

`selector` Selects the video keyer control to set.

`keyerCtrlVal` The keyer control value.

selector	keyerCtrlVal	Meaning
`vdKColorEnable`	`vdEnable` or `vdDisable`	Key color enable
`vdKeyEnhDiss`	`vdEnable` or `vdDisable`	Enhanced dissolve mode

Note Enhanced dissolve mode causes a frame switching of the rapid alternation pattern used to achieve dissolve on the RGB output of some video overlay implementations. It helps reduce aliasing between the displayed graphics and the alternation pattern in some situations. Enhanced dissolve mode does not apply to implementations that achieve RGB dissolve through other means, and consequently is not supported in these implementations.

Call `VDGetFeatures(vdKeyEnhDiss)` to see if enhanced dissolve mode is supported by the video system.

VDKeyGetKBCol $1221

`VDKeyGetKBCol` reads the blue component of the current key color.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
	<—SP

Stack after call

Previous contents	
color	**Word**—Blue component of the key color
	<—SP

Errors None.

C

```
extern pascal Word VDKeyGetKBCol();
```

Note The significant bits of the color are left justified in the value that is returned. For example, if 4 bits of color are used, the color bits are returned in bits 15-12.

Call `VDGetFeatures` to determine the number of significant bits in each value.

VDKeyGetKDiss $1421

`VDKeyGetKDiss` reads the current key color dissolve value.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
	<—SP

Stack after call

Previous contents	
dissolve	**Word**—Current key color dissolve value
	<—SP

Errors None.

C

```
extern pascal Word VDKeyGetKDiss();
```

VDKeyGetKGCol $1121

`VDKeyGetKGCol` reads the green component of the current key color.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
	<—SP

Stack after call

Previous contents	
color	**Word**—Green component of the key color
	<—SP

Errors None.

C

```
extern pascal Word VDKeyGetKGCol();
```

Note The significant bits of the color are left justified in the value that is returned. For example, if 4 bits of color are used, the color bits are returned in bits 15-12.

Call `VDGetFeatures` to determine the number of significant bits in each value.

VDKeyGetKRCol $1021

`VDKeyGetKRCol` reads the red component of the current key color.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
	<—SP

Stack after call

Previous contents	
color	**Word**—Red component of the key color
	<—SP

Errors None.

C

```
extern pascal Word VDKeyGetKRCol();
```

Note The significant bits of the color are left justified in the value that is returned. For example, if 4 bits of color are used, the color bits are returned in bits 15-12.

Call `VDGetFeatures` to determine the number of significant bits in each value.

VDKeyGetNKDiss $1621

`VDKeyGetKDiss` reads the current non-key color dissolve value.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
	<—SP

Stack after call

Previous contents	
dissolve	**Word**—Current non-key color dissolve value
	<—SP

Errors None.

C

```
extern pascal Word VDKeyGetNKDiss();
```

VDKeySetKCol $0F21

`VDKeySetKCol` sets the key color.

Parameters

Stack before call

Previous contents	
redValue	**Word**—Red component of key color
greenValue	**Word**—Green component of key color
blueValue	**Word**—Blue component of key color
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $2113 `VDInvalidParam` An invalid parameter was specified.

C

```
extern pascal void VDKeySetKCol(redValue, greenValue, blueValue);
Word redValue, greenValue, blueValue;
```

`redValue` Red component of the key color.

`greenValue` Green component of the key color.

`blueValue` Blue component of the key color.

Note The color values are unsigned integers. Call `VDGetFeatures` to determine the number of significant bits in each value, then left justify the bits in the value. For example, if there are 4 significant bits of color, bits 15-12 would be used to record the color. Any bits below the significant bits are ignored.

VDKeySetKDiss $1321

`VDKeySetKDiss` sets the key color dissolve value.

Parameters

Stack before call

Previous contents	
kDissolve	**Word**—Key color dissolve value
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $2113 `VDInvalidParam` An invalid parameter was specified.

C

```
extern pascal void VDKeySetKDiss(kDissolve);
Word kDissolve;
```

`kDissolve` Key color dissolve value. A low value gives a high level of graphics and a low level of video, while a high value gives less graphics and more intense video.

Note Call `VDGetFeatures` to determine the number of dissolve levels. While you can specify any value for the `kDissolve` parameter, the value will be pinned to the closest legal value.

VDKeySetNKDiss $1521

`VDKeySetNKDiss` sets the non-key color dissolve value.

Parameters

Stack before call

Previous contents	
nKDissolve	**Word**—Non-key color dissolve value
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $2113 `VDInvalidParam` An invalid parameter was specified.

C

```
extern pascal void VDKeySetNKDiss(nKDissolve);
Word nKDissolve;
```

`nKDissolve` Non-key color dissolve value. A low value gives a high level of graphics and a low level of video, while a high value gives less graphics and more intense video.

Note Call `VDGetFeatures` to determine the number of dissolve levels. While you can specify any value for the `kDissolve` parameter, the value will be pinned to the closest legal value.

VDKeyStatus $0E21

`VDKeyStatus` reads the current keyer control status information.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
selector	**Word**—Selects which status information to return
	<—SP

Stack after call

Previous contents	
keyerStatus	**Word**—Status information
	<—SP

Errors $2112 `VDInvalidSelector` An invalid selector was specified.

C

```
extern pascal Word VDKeyStatus(selector);
Word selector;
```

`selector` Selects which status information to return:

selector	keyerStatus	Meaning
vdKColorEnable	vdEnable or vdDisable	Key color enable
vdKeyEnhDiss	vdEnable or vdDisable	Enhanced dissolve mode

VDOutControl $1921

`VDOutControl` sets the video composite output control values.

Future systems may only support a graphics blanking source, but unlike the Apple II Video Overlay Card, it will conform to video standards. Also note that the Apple II Video Overlay Card uses external as its default, but future implementations may default to graphics if that is all they support.

Monochrome text mode override is only available when the `vdRAMPageSel` is `vdAux` (see `VDGGControl`). Switching from Aux RAM Page will disable text monochrome override. To avoid a temporary display disturbance, don't enable text monochrome mode while in text mode with `vdRAMPageSel` set to `vdAux`.

Parameters

Stack before call

Previous contents	
selector	**Word**—Selects the output control value to set
outCtrlValue	**Word**—The output control value
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2112	`VDInvalidSelector`	An invalid selector was specified.
$2113	`VDInvalidParam`	An invalid parameter was specified.

C

```
extern pascal void VDOutControl(selector, outCtrlValue);
Word selector, outCtrlValue;
```

`selector` Selects the output control value to set.

`outCtrlValue` New output control value.

selector	outCtrlValue	Meaning
`vdOutChromaFltr`	`vdEnable` or `vdDisable`	Graphics chroma-crosstalk filter
`vdOutSetup`	`vdEnable` or `vdDisable`	Setup
`vdOutExtBlank`	`vdExternal` or `vdGraphics`	Blanking source select
`vdTextMonoOver`	`vdEnable` or `vdDisable`	Monochrome text mode override

VDOutGetStd $1821

`VDOutGetStd` reads the currently selected video composite output standard.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
	<—SP

Stack after call

Previous contents	
outStandard	**Word**—Video output standard
	<—SP

Errors None.

C

```
extern pascal Word VDOutGetStd();
```

The value returned is one of:

outStandard	Meaning
vdNone	Output not available
vdNTSC	NTSC video standard
vdPAL	PAL video standard
vdSECAM	SECAM video standard
vdSNTSC	SNTSC (Y/C) video standard
vdSPAL	SPAL (Y/C) video standard
vdSSECAM	SSECAM (Y/C) video standard

VDOutSetStd $1721

`VDOutSetStd` sets the video composite output standard.

Parameters

Stack before call

Previous contents	
outStandard	**Word**—The output standard
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $2113 `VDInvalidParam` An invalid parameter was specified.

C

```
extern pascal void VDOutSetStd(outStandard);
Word outStandard;
```

`outStandard` The output standard to use.

outStandard	Meaning
vdNTSC	NTSC video standard
vdPAL	PAL video standard
vdSECAM	SECAM video standard
vdSNTSC	SNTSC (Y/C) video standard
vdSPAL	SPAL (Y/C) video standard
vdSSECAM	SSECAM (Y/C) video standard

Note Call `VDGetFeatures` to determine which video output standards are supported by the video system. Also note that RGB output is always supported; however, the RGB frame rate is determined by the standard set with the `VDOutSetStd` routine.

VDOutStatus $1A21

`VDOutStatus` gets the video composite output status information.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
selector	**Word**—Selects the status information to return
	<—SP

Stack after call

Previous contents	
outStatus	**Word**—Status information
	<—SP

Errors $2112 `VDInvalidSelector` An invalid selector was specified.

C

```
extern pascal Word VDOutStatus(selector);
Word status;
```

`selector` Selects which status information to return.

selector	outStatus	Meaning
`vdOutChromaFltr`	`vdEnable` or `vdDisable`	Graphics chroma-crosstalk filter
`vdOutSetup`	`vdEnable` or `vdDisable`	Setup
`vdOutExtBlank`	`vdExternal` or `vdGraphics`	Blanking source select
`vdTextMonoOver`	`vdEnable` or `vdDisable`	Monochrome text mode override
`vdVerticalBlank`	`vdVBlank` or `vdActiveVideo`	Currently in vertical blank

Video Overlay Tool Set Constants

Constant	Value	
vdVideoOverlay	1	$01
vdFrameGrabber	2	$02
vdInVStandards	3	$03
vdOutVStandards	4	$04
vdKeyDissLevels	5	$05
vdNKeyDissLevels	6	$06
vdAdjSideEffect	7	$07
vdKeyColorBits	8	$08
vdInHueAdj	9	$09
vdInSatAdj	10	$0A
vdInContrastAdj	11	$0B
vdInBrightAdj	12	$0C
vdOutSetup	13	$0D
vdOutChromaFltr	14	$0E
vdOutExtBlank	15	$0F
vdKeyEnhDiss	16	$10
vdLineInterrupt	17	$11
vdGGBus	18	$12
vdDualOut	19	$13
vdTextMonoOver	20	$14
vdGenlock	50	$32
vdVideoDetect	51	$33
vdGenlocked	52	$34
vdAdjInc	80	$50
vdAdjDec	81	$51
vdAdjSave	82	$52
vdTrue	1	$01
vdFalse	0	$00
vdAvail	True	$01
vdNotAvail	False	$00
vdYes	True	$01
vdNo	False	$00
vdOn	True	$01
vdOff	False	$00
vdNil	False	$00
vdKColorEnable	100	$64
vdVerticalBlank	130	$82

Constant	Value	
vdMainPageLin	200	$C8
vdRAMPageSel	201	$C9
vdVBLInterrupt	202	$CA
vdInterlaceMode	203	$CB
vdClearVBLInt	204	$CC
vdClearLineInt	205	$CD
vdDisplayField	206	$CE
vdVBLIntRequest	207	$CF
vdLineIntRequest	208	$D0
vdNone	0	$00
vdNTSC	1	$01
vdPAL	2	$02
vdSECAM	4	$04
vdSNTSC	8	$08
vdSPAL	16	$10
vdSSECAM	32	$20
vdRGB60	64	$40
vdRGB50	128	$80
vdAux	0	$00
vdMain	16	$10
vdInterlace	48	$30
vdField1	True	$01
vdField0	False	$00
vdEnable	True	$01
vdDisable	False	$00
vdExternal	False	$00
vdGraphics	True	$01
vdVBlank	True	$01
vdActiveVideo	False	$00

Chapter 32 **Window Manager Update**

This chapter contains new information about the Window Manager. The original reference to this tool set is in Volume 2, Chapter 25 of the *Apple IIGS Toolbox Reference*. A previous update to this tool set was published in Volume 3, Chapter 52 of the *Apple IIGS Toolbox Reference*.

New Features

- `WindStatus` now returns with the carry flag set and the Accumulator set to 0 when the Window Manager is started up but a window update is in progress (that is `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.

- Window title clipping no longer draws onto the desktop if the window is extremely narrow. (This is still a cosmetic problem for ROM 3 in System Software 6.0.)

- `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 is drawn. 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 occurred.

`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 the system resource file).

- 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-. (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 as per Human Interface Note #10. 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 on the left, and the last two are clustered on the right. (See 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 insertions. GS/OS sets 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.

Tip `AlertWindow` assumes that the `LoadResource` call for the icon will succeed. You can make sure it will by pre-flighting it (that is, 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` Enhancements

- Passing selector 8, `checkForNewDeskMsg`, to the `Desktop` call causes the system to re-check the Message Center 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)

About the Modal Window Calls

The modal window calls provide extensive modal window capabilities for any application. Unlike the Dialog Manager, the modal window calls work with extended controls created by `NewControl2`. This allows dialog boxes to contain controls new to System Software 5.0 and later, such as Pop-up controls, Picture controls and TextEdit controls. The Modal Window calls also provide new and more robust Apple Desktop Interface support.

System Software 6.0's Window Manager introduces the concept of **movable modal dialog boxes**. Movable modal dialog boxes are modal in nature—the user can't select other application windows until the dialog is dismissed. Unlike standard modal dialog boxes, however, movable modal dialog boxes may be moved on the desktop, allowing the user to see, for example, part of the current document beneath the dialog box.

In standard modal dialog boxes, only actions within the modal dialog window are tolerated. Other user actions result in a beep from the speaker. The modal window calls can allow dialog boxes to use other desktop interface elements, such as desk accessories or the Edit menu. In a standard modal dialog box, the Edit menu is not available, although the Dialog Manager handles the standard Edit menu item key equivalents for LineEdit items. Modal window calls allow any menu item to be selectively available depending on which window is active.

The Modal Window Calls and the Dialog Manager

A **dialog box** appears on the screen when an application needs more information to carry out a command. The modal window calls and the Dialog Manager both handle the appearance and operation of dialog boxes, but the two perform these tasks in different ways.

Dialogs and the Dialog Manager

When using the Dialog Manager, you create dialog boxes with Dialog Manager tool calls. The Dialog Manager has built-in support for many standard human interface components, including editable one-line text items, radio buttons, checkboxes and non-editable text. Custom items can be added by the application, which provides routines to the Dialog Manager to draw and manipulate the custom items.

The Dialog Manager deals with individual **dialog items** within a dialog box. It expects those items to be largely self-contained, and the Dialog Manager has problems when custom items use system resources or fields outside the dialog item's scope. For example, the List Manager implements lists as custom controls, using the control's `refCon` field for list information. The Dialog Manager also uses the control `refCon`, and therefore the two conflict, making it extremely difficult to put a list into a Dialog Manager window. Similarly, it is also difficult to put extended controls (such as a TextEdit control) into a Dialog Manager window. The Dialog Manager has no support for movable modal dialog boxes and does not allow any menu items to be selected while a modal dialog box is frontmost, in accordance with the *Human Interface Guidelines* at the time the Dialog Manager was written.

Dialogs and the Modal Window Calls

The modal window calls provide facilities to treat the frontmost application window as a modal dialog box. The window may be created in whatever manner is most convenient, probably by using the `NewWindow2` tool call. `DoModalWindow` has built-in support for extended controls in windows, and provides further support for editable text in LineEdit or TextEdit controls.

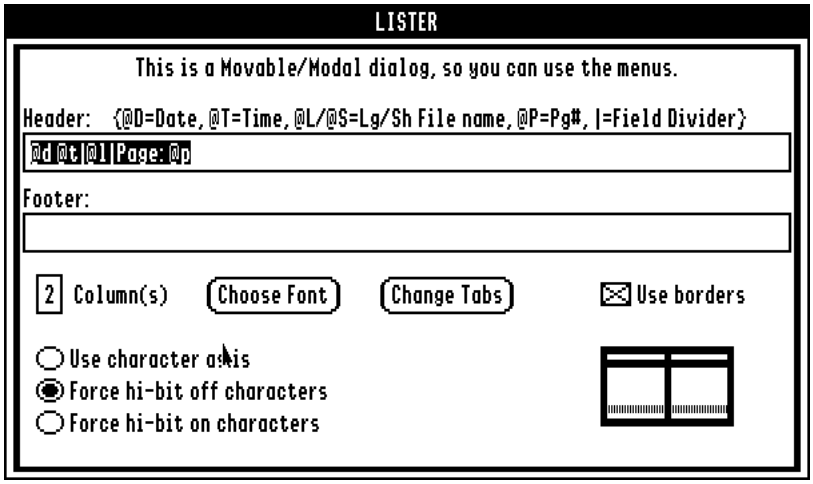
`DoModalWindow` makes no assumptions about the content of the window it treats as a modal dialog box. Its default drawing procedure (`MWStdDrawProc`) automatically draws any controls in the dialog box, but your application can also draw in the window as appropriate. `DoModalWindow` allows users to select menu items while a modal dialog box is frontmost. The modal window calls also have support for movable modal dialog boxes, a recent addition to the Apple Desktop Interface. This is described in the next section, "Introducing Movable Modal Dialog Boxes."

Introducing Movable Modal Dialog Boxes

There are instances in some applications where it is necessary to present the user with an important choice—a decision must be made before the application can proceed further. It is in these instances that modal dialog boxes are used, for the user can take no further action until he dismisses the modal dialog box by making a choice. If only one choice is present, the user's choice indicates that he is aware of the circumstances and is ready to proceed.

However, not all of the situations that require a choice from the user also require all computer functionality to cease until a decision is made. Perhaps the choice the user must make is relevant to a document open on the desktop that's partially obscured by the modal dialog window. For example, a find and replace dialog might obscure the context of the words to be replaced.

This situation is handled by a new human interface object that combines the capabilities of a movable window with the properties of a modal dialog box. Apple calls this new object a **movable modal dialog box.**



The movable modal dialog box features a title bar that can be used to drag the window, like an ordinary window. However, it has no close box or zoom box. The interior of the box contains a rectangle similar to an alert frame for a regular modal dialog box. The visual clues combine to tell the user that he can move the window but it is also modal.

Unlike standard modal dialog boxes, actions may take place outside the window. On a Macintosh™ under MultiFinder™ or System Software 7.0, applications may be switched while a movable modal dialog box is frontmost. In other words, windows belonging to other applications or desk accessories may be selected, but no other windows belonging to this application can be brought to the front. Menu items may be selected if enabled by the application. There is no analog to MultiFinder on the Apple IIGS, but there are desk accessories, and the modal window calls optionally allow them to be opened and selected.

The modal window calls can treat a window as a movable modal dialog box, including drawing the interior alert frame. The inclusion of the alert frame reduces the size of the content area available for other elements of the dialog box; the alert frame is 10 pixels wide by 4 pixels high on each side, reducing the vertical size by 8 pixels and the horizontal size by 20 pixels. In 320 mode, the interior alert frame is 4 pixels wide by 4 pixels high, reducing the vertical and horizontal sizes by 8 pixels each. Note that the window content area is only reduced for movable modal dialog windows; the

Window Manager includes the alert frame rectangle as part of the frame if the `fAlert` bit in the window frame is set.

If you wish the Window Manager to handle the window as a standard modal dialog box, the `fAlert` bit in the window frame should be set. If you wish to have a modal dialog box that doesn't have an alert frame (not recommended), `fAlert` and `fFlex` should both be clear. The `fFlex` bit should be unused for dialog boxes in all other cases, as they should not have scroll bars.

Note The Window Manager's flexibility can easily allow your application to do things in blatant violation of Apple's *Human Interface Guidelines*. Although the *Guidelines* are not absolute rules which can never be broken, they are solid suggestions to provide the level of consistency necessary for the Desktop Interface to be effective. Please carefully consider your interface design when using the modal window calls.

Using the Modal Window Calls

The modal window calls treat the frontmost application window in a modal fashion. Your application can create that window in whatever way is most efficient, probably by using the `NewWindow2` tool call. Your application creates the window and all of its contents; `DoModalWindow` does not create dialogs.

The dialog box can contain anything your application wishes to place in it, but the modal window calls have built-in support for extended controls. The modal window calls return the control ID of an extended control chosen by the user. Since non-extended controls do not have control IDs, you will probably wish to use only extended controls in your dialog windows.

When you call `DoModalWindow`, you pass an extended task record, like the ones used by the Window Manager tool call `TaskMaster`. `DoModalWindow` then calls `GetNextEvent` and finds and tracks any controls the user selects, returning the control ID as a result and in the task record in the `wmTaskData2` field. You may also tell `DoModalWindow` to allow menu selections, in which case it returns the menu ID and menu item ID in the task record. Bit 31 of the four-byte return space distinguishes menu selections from control selections—it is set for menu selections and clear for control selections. All your menu ID values and control ID values must have the high bit (bit 15 for menu IDs, bit 31 for control IDs) clear for `DoModalWindow` to function properly. Menu item IDs are unaffected.

△ **Important** `DoModalWindow` expects controls in its windows to be extended controls, and it performs most operations using the control ID field, which is present only for extended controls. While `DoModalWindow` will operate with non-extended controls, they are not handled as robustly as extended controls. This should only be an issue for custom controls; `DoModalWindow` handles extended custom controls better than non-extended custom controls. △

Your application passes `DoModalWindow` a word of flags which control the way the dialog box is handled. You can control whether or not standard Edit menu actions on LineEdit controls affect the desk scrap, whether or not menu selections and menu keys are allowed, whether or not editable text controls use an I-beam cursor, whether or not desk accessories can be opened while the modal dialog box is present, whether all windows should be updated and whether the dialog box is movable.

Movable modal dialog boxes have both title bars and alert frames. It is the combination of these two human interface components that signals the dual-natured behavior of the movable modal dialog box. The Window Manager will automatically create title bars in a window's frame if the `fMove` bit is set, and it will automatically create an alert frame if the `fAlert` bit is set. However, setting both of these bits is not valid and does not produce a movable modal dialog window. To work around this, the modal window calls uses the `fFlex` bit as a substitute for `fAlert`. The Window Manager creates a movable window with a title bar when `fMove` is set, and the modal window calls draws an "interior" alert frame if `fFlex` is also set. Since modal dialog boxes should not have scroll bars, `fFlex` should be clear in all other cases.

Handling Events in the Dialog Box

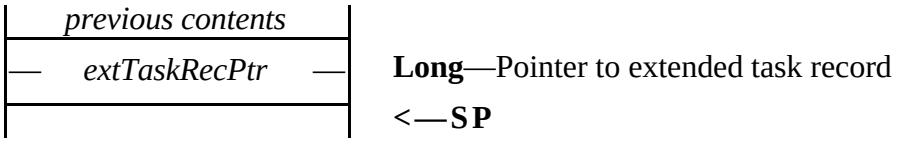
Once the dialog window is created, you can call `DoModalWindow` to handle events for the window. `DoModalWindow` calls the Event Manager routine `GetNextEvent` to determine what to do. Before taking action on the event, `DoModalWindow` first passes it to an optional event hook procedure

supplied by your application. This gives you the chance to filter or modify any events before `DoModalWindow` acts on them.

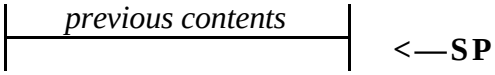
For example, suppose you want your dialog box to have a "Save" button with two standard Human Interface Guidelines key equivalents, Return and Command-S. Although you can specify two key equivalents for an extended simple button control, you can only specify one set of modifiers, making it impossible for one key equivalent and not the other to require the Command key. Your event hook function could check the event record to see if it is a Return key-down event, and if so, change the event record to make it a Command-S key-down event. Then your "Save" button can have a Command-S key equivalent while Return performs the same action.

The event hook procedure has one parameter and returns nothing. It is declared like this:

Stack before call



Stack after call



If there is a mouse down event, `DoModalWindow` calls the Window Manager routine `FindWindow` to see where the mouse button was pressed.

- If `FindWindow` returns the mouse down was within a system window, and desk accessory handling is enabled in `DoModalWindow`, it calls the Desk Manager routine `SystemClick` to allow the desk accessory to process the event.

- If the mouse down was within the content region of the dialog window, the Window Manager calls the Control Manager routines `FindControl` and `TrackControl` to determine if the user is actually selecting a control within the dialog box. If `TrackControl` returns a part code, `DoModalWindow` examines the selected control. If it is a radio button or a checkbox, `DoModalWindow` changes the control's value so checkboxes toggle or the selected radio button is now the chosen one. It then returns the selected control's ID as the result and saves the part code for possible future reference.

- If the mouse was pressed outside the content region of the dialog box, `DoModalWindow` calls the Menu Manager routine `MenuSelect` if menu selections are currently allowed. If menu selections are not allowed, or if there were no menus to track, the Window Manager calls the beep routine as described later in `DoModalWindow`.

If a key is pressed and menu key selections are currently allowed, `DoModalWindow` calls the Menu Manager routine `MenuKey`. If the key is a standard Edit menu key (Command-X, Command-C or Command-V), `DoModalWindow` initiates cut, copy or paste actions respectively for the current target control, even if those Edit menu items are disabled. See the section "Standard Editing Functions" later in this chapter.

When dialog boxes are movable, update events may be generated. If `DoModalWindow` gets an update event, it calls the update procedure specified for the modal dialog box (or the default

procedure `MWStdDrawProc` if no update procedure is supplied). The modal window calls can be used to update only the modal dialog box or to update all windows.

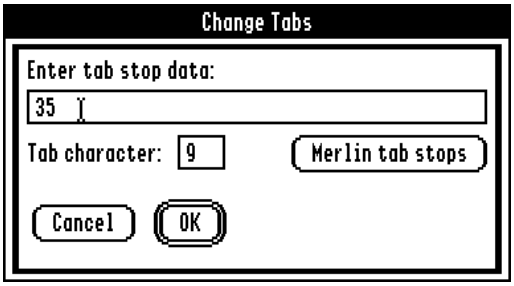
△ **Important** When updating windows other than modal dialog windows, `DoModalWindow` assumes that the `wContDefProc` field in the window record contains the address of a routine to be called to update a window or `NIL` for no action. If you tell `DoModalWindow` to update all windows, each window must have a valid content draw procedure address or `NIL` in the `wContDefProc` field in the window record. △

When it receives an activate event, `DoModalWindow` calls the Window Manager routine `InvalRect` with the boundary rectangles of all known controls that change appearance when activated or deactivated. It also calls menu update routines optionally provided through the routine `MWSetMenuProc`, to allow you to set up menus properly for the new active window.

If the event is one not handled as discussed above, `DoModalWindow` calls the Control Manager routine `SendEventToCtls` to allow controls that accept key presses or other events to handle the event.

Cursor Manipulation

The *Human Interface Guidelines* suggest that the regular "arrow" cursor be replaced with an "I-beam" cursor when the mouse is positioned over an item containing editable text.



`DoModalWindow` can call the QuickDraw II Auxiliary routine `IBeamCursor` to handle this. If bit 3 is set in the `flags` parameter to `DoModalWindow`, an I-beam cursor will automatically be used when the cursor's "hot spot" is over the editable portion of a LineEdit or TextEdit control.

Standard Editing Functions

The modal window calls handle several of the editing functions described in Apple's *Human Interface Guidelines*. The support for editing functions is handled both through menus and without them.

The modal window calls handle commands in the Edit menu properly, provided your application uses the standard Edit menu item numbers, as described with `MWSetUpEditMenu` later in this chapter. When `DoModalWindow` finds an Edit menu command has been chosen (through a pull-down menu or through a menu key, when those functions are available), it performs the necessary Control Manager, Scrap Manager, LineEdit and TextEdit calls to create seamless editing using the clipboard (the desk scrap).

`DoModalWindow` examines the state of LineEdit and TextEdit controls to see what actions are appropriate. For example, Copy is always appropriate but Cut or Paste is not appropriate in a read-only TextEdit control. A full description of how Edit menu items are enabled can be found in the description of `MWSetUpEditMenu` later in this chapter.

`DoModalWindow` also treats Command-X, Command-C and Command-V as Cut, Copy and Paste, even when those menu items are disabled. This allows users to continue to use the familiar and standard editing key commands without forcing you to enable the Edit menu items, in case doing so is impractical in a particular instance.

Because `DoModalWindow` handles some Command key equivalents as both menu keys and editing keys, you should be aware of the following:

- If there is an editable text control in the modal window, `DoModalWindow` always treats Command-X, Command-C and Command-V as editing commands. In these cases, the event is not sent to controls; therefore, extended controls can't have key equivalents of Command-X, Command-C and Command-V if editable text controls are in the same window.

- If you have menu keys enabled and disable a standard Editing function, The modal window calls still treats that key as an editing function. For example, if you disable Paste in the Edit menu, The modal window calls still treats Command-V as a Paste command. Simply disabling Edit menu items is not sufficient; you must also install an event hook procedure to transform the editing keys into other events. In the preceding example, you could change the Command-V key-down event to a null event to prevent `DoModalWindow` from taking any action.

- Although `DoModalWindow` calls `MWSetUpEditMenu` to properly configure the Edit menu items, the behavior described above can be conceptually confusing. For example, when the target control is a read-only TextEdit control, `MWSetUpEditMenu` disables Cut and Paste. However, as described above, `DoModalWindow` still attempts to Cut when Command-X is pressed or Paste when Command-V is pressed. This does not create a problem, since `TEPaste` has no effect on a read-only TextEdit control, and `DoModalWindow` knows not to call `TECut` on a read-only TextEdit control to prevent the selection from being copied to the desk scrap.

New Window Manager Calls

DoModalWindow $640E

DoModalWindow handles user interactions with a window containing extended controls. The window itself, along with any controls, should be created before the call to DoModalWindow; this call handles any actions the user takes and returns information about that action, allowing use of desk accessories, updating windows, and so forth in the process. The modal dialogs you create with DoModalWindow are more flexible than those created with the older Dialog Manager calls.

The appearance of DoModalWindow windows is also a hybrid; they have a title bar for dragging, but they also have an alert frame in the content region.

See "Using Modal Window Calls," earlier in this chapter, for a functional description of all of the calls associated with modal dialogs.

Note DoModalWindow is very similar to the fakeModalDialog call from the Developer Technical Support fakeModalDialog tool set.

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), retrieve any information needed from the controls—radio button states (FindRadioButton), checkbox 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 window, 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
flags	**Word**—Flags
	<—SP

Stack after call

Previous contents	
ID	**Long**—ID of control or menu item that was selected
	<—SP

Errors None.

C

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

`eventPtr` The address of the extended task record `DoModalWindow` will use for calls to `GetNextEvent`.

In System Software 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 the subroutine to be called when the modal dialog window needs to be updated. `DoModalWindow` stores this address in the window's `wContentDraw` field.

If `NIL` is passed, `DoModalWindow` stores the address of the standard draw procedure, `MWStdDrawProc`. `MWStdDrawProc` calls the Control Manager routine `DrawControls` and, if necessary, draws the interior alert frame.

`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 pointed 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-. (Command-period) key presses into Esc key presses before calling any `eventHook` routine. (It's okay to pass $80000000 to translate Command-. 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` The flags parameter is a series of flags that enables or disables several optional features of `DoModalWindow`.

`mwMovable`	bit 15	If this bit is set, the dialog is moveable; if the bit is clear, the user is not able to move the dialog. Even if the bit is set, the window cannot be moved unless it has a drag region.
`mwUpdateAll`	bit 14	If this bit is set, `DoModalWindow` handles update events for all windows on the desktop; if the bit is clear, `DoModalWindow` only handles update events for the frontmost application window, which is the dialog. When `DoModalWindow` updates a window other than the dialog window, it does so by calling the `wContDraw` procedure defined in the window's window record. If `DoModalWindow` is to update other windows, the `wContDraw` field must point to a valid update procedure or must be set to `NIL` for every application window. If `DoModalWindow` is not told to update window contents, and the dialog window is moved, other application windows are left partially blank until the dialog is closed. System windows (generally those created by an NDA) get updated automatically get updated automatically during calls to `GetNextEvent`, regardless of whether this bit is set or clear.
`reserved`	bits 13-6	Reserved. These bits must be set to 0.
`wmWantActivate`	bit 5	If this bit is set, `DoModalWindow` returns to the caller after handling an activate bit. If this bit is not set, all activate events are handled internally, and the caller is never aware of them.
`mwDeskAcc`	bit 4	If this bit is set, `DoModalWindow` automatically handles desk accessories.

`mwIBeam`	bit 3	If this bit is set, `DoModalWindow` automatically switches to the I-beam cursor when the hot spot is over an LineEdit control or TextEdit control.
`mwMenuKey`	bit 2	If this bit is set, menu key events are handled as menu events first. If `MenuKey` returns `FALSE`, the key is handled as a standard key event. `DoModalWindow` handles standard editing menu keys even if this bit is disabled.
`mwMenuSelect`	bit 1	If this bit is set, `DoModalWindow` allows the user to pull down menus; if the bit is clear, menus cannot be pulled down. Regardless of the setting of this bit, `DoModalWindow` still respects the standard editing key equivalents for Cut, Copy and Paste.
`mwNoScrapForLE`	bit 0	If this bit is set, `DoModalWindow` dos *not* use the Scrap Manager for cut, copy and paste operations; if the bit is clear, `DoModalWindow` *does* use the Scrap Manager for these operations. When only a small amount of LineEdit text is being edited and there is no desk scrap or TextEdit controls, set this bit so the user will not destroy the system scrap by doing cut-and-paste editing between LineEdit controls.

`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 is 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 key press 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, `DoModalWindow` automatically converts Command-. (Command-period) key press events to Esc key events. After that, it still calls your event hook routine if the rest of the pointer is non-`NIL`.
- `DoModalWindow` calls `SysBeep2` with a `beepType` of $0004 if the user clicks outside of the dialog window inappropriately.

- `DoModalWindow` uses an event mask of $0FFF when it calls `GetNextEvent`. `fakeModalDialog` used an even mask of $FFFF, claiming `app1` through `app4` events, which is generally not appropriate.

FindCursorCtl $690E

`FindCursorCtl` returns the handle for the control beneath a given point. `DoModalWindow` uses this tool call to determine if it should use the I-beam cursor. If the control under the hot spot is an LineEdit control or a TextEdit control, `DoModalWindow` uses the I-beam cursor; otherwise, it uses the arrow cursor.

The position is given in local coordinates.

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

Note `FindCursorCtl` is very similar to the `fmdFindCursorCtl` call from the Developer Technical Support `fakeModalDialog` tool set.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
— *ctlHandPtr* —	**Long**—Pointer to space for control handle
xLoc	**Word**—local X coordinate of point
yLoc	**Word**—local Y coordinate of point
— *windPtr* —	**Long**—Pointer to the window to check (`NIL` for front window)
	<—SP

Stack after call

Previous contents	
partCode	**Word**—Part code for the control beneath the given point
	<—SP

Errors None.

C

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

`ctlHandlePtr` Pointer to a four-byte buffer where `FindCursorCtl` stores the control handle.

`xLoc` Horizontal location to check. Use local coordinates.

`yLoc` Vertical location to check. Use local coordinates.

`windPtr` Pointer to the window to check.

`partCode` Part code for the control found.

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 utilities. (See the Desk Manager update in Chapter 4 of this book.)

Auxiliary Window memory structure (fields marked with an asterisk (*) are reserved for future use):

$00	*size*	**Word**—Size in bytes (Currently 28, but this may grow in the future.)
$02	*bank*	**Word**—*Bank register value; in least significant byte
$04	*DP*	**Word**—*Direct page register value
$06	*resValue*	**Word**—*Resource application value
$08	— *oldHandle* —	**Long**—*Old update region handle
$0C	— *oldPort* —	**Long**—*Old port (for `EndUpdate`)
$10	— *layer* —	**Long**—*Window layer (for Windoid support)
$14	*minV*	**Word**—Minimum vertical size for a System window
$16	*minH*	**Word**—Minimum horizontal size for a System window
$18	— *NDAPtr* —	**Long**—NDA structure pointer (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 `memErr` could not allocate memory

C

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

`windPtr` Pointer to the window for which to return auxiliary information.

`auxInfoPtr` Pointer to the auxiliary information record.

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

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 on line, initialize it, or erase it.

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

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

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
Space	**Word**—Space for result
flags	**Word**—Input flags
devNum	**Word**—GS/OS device number
	<—SP

Stack after call

Previous contents	
resultFlags	**Word**—Resulting flags
resultDevNumber	**Word**—Resulting device number
	<—SP

Errors GS/OS errors are 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 insertions and ejections. 0 = don't scan the devices.
bit 14 (`hdiHandle`)	1 = identify a disk and handle user interaction if the disk is not usable. 0 = don't check disks.
bit 13 (`hdiUpdate`)	1 = update the status of a particular device or all devices. (Forces `HandleDiskInsert` to check with GS/OS about the on-line status of the devices.)

	0 = don't update the status.
bit 12 (`hdiReportEjects`)	1 = report any detected disk ejections as well as insertions.
	0 = report only insertions, not ejections.
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).
	0 = the status of the disk is not known.
bits 7-1	Reserved, use 0.
bit 0 (`hdiMarkOffline`)	1 = Mark all devices as off line, so that disk insertions are reported for all already on-line volumes.
	0 = don't mark all devices as off line.

`devNum` `devNum` is normally zero. Pass a nonzero device number if you already detected an insert and want the system to check it out, possibly asking 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` `resultDevNum` is the device number of a device that was inserted or ejected. It will be 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 these bits.
Bit 1	Set if `resultDevNum` represents a disk that the user elected to format.
Bit 0	Set if `resultDevNum` represents a disk ejection.

Scanning for Insertions and Ejections

Scanning only occurs if flag bit 15 (`hdiScan`) 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, `hdiHandle`).

- When scanning, 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" insertion 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, `hdiNoDelay`.
- `HandleDiskInsert` keeps an internal table recording its idea of the on-line/off-line status of each device. When a device's status differs from the value recorded in this table, an insertion or ejection 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 (`hdiHandle`) 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 (`hdiScan`).

- When an insert is detected, `HandleDiskInsert` does a `Volume` call on the device. If there's no error and if bit 15 (`hdiScan`) is set, `HandleDiskInsert` keeps looking for additional insertions. If there is nowhere else to look, `HandleDiskInsert` returns with no error.
- If `Volume` returns an I/O error, `HandleDiskInsert` calls `AlertWindow` asking the user to eject or initialize the disk.
  - If the user elects to eject the disk, `HandleDiskInsert` makes a `DControl` call to eject the disk.
  - If the user elects to initialize the disk, `HandleDiskInsert` makes the GS/OS `Format` call to format the disk. The `Format` call lets the user name the disk and specify the file system and format options.
  - If the user selects "Cancel" in the `Format` dialog, `HandleDiskInsert` ejects the disk and returns with no error.
  - If the formatting is successful, `HandleDiskInsert` returns the device number as the `resultDevNum`.
- If `Volume` returns an unrecognized volume error, `HandleDiskInsert` proceeds as described under "Identifying Unknown Disks."
- If `Volume` returns a duplicate volume name error and it is possible to rename the volume, `HandleDiskInsert` gives the user a chance to rename the disk.
- If the `Volume` call returns a strange error, `HandleDiskInsert` leaves it on-line and returns the device number and 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:

$00	*recvCount*	**Word**—Used by `SendRequest`
$02	*reserved*	**Word**—Reserved
$04	*disposition*	**Word**—Result

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

$FFFF	Leave the volume on line, even though it was unrecognized. Report no error to the caller.
$0000	Eject the media in the device.
$0001	Issue a GS/OS `Format` call to the device, letting the user choose the name, file system, and formatting options.
$0002	Issue a GS/OS `EraseDisk` call to 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 .SAMPLE). 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 the call completes, the cursor is restored to its previous appearance. During the call, arrow and watch cursors are used.

The System Unknown Disk Procedure

The system installs an `AcceptRequests` procedure at boot time. When this procedure receives a request to identify a disk it checks the file system. 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 built into System Software 6.0:

> The disk in device .SAMPLE 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 .SAMPLE 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 .SAMPLE 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 .SAMPLE appears to be in MS-DOS format. An MS-DOS File System Translator is required to use this disk.
>
> The disk in device .SAMPLE 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 .SAMPLE 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 .SAMPLE 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 .SAMPLE 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 .SAMPLE appears to be in ProDOS format. Installing the ProDOS File System Translator allows GS/OS to read this disk.

MWGetCtlPart $650E

`MWGetCtlPart` returns the part code from the most recent `TrackControl` call made during the most recent call to `DoModalWindow`. If no `TrackControl` call was made during the most recent call to `DoModalWindow` this call returns $0000.

See "Using Modal Window Calls," earlier in this chapter, for a functional description of all of the calls associated with modal dialogs.

Note `MWGetCtlPart` is very similar to the `fmdGetCtlPart` call from the Developer Technical Support `fakeModalDialog` tool set.

Parameters

Stack before call

Previous contents	
Space	**Word**—Space for result
	<—SP

Stack after call

Previous contents	
ctlPart	**Word**—Part code
	<—SP

Errors None.

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

`ctlPart` Part code from `DoModalWindow`'s last `TrackControl` call.

MWSetMenuProc $660E

`MWSetMenuProc` lets you designate a subroutine that is called if the frontmost window changes while `DoModalWindow` is handling dialog events.

Once a procedure has been installed, passing `NIL` to `MWSetMenuProc` will disable the procedure, so that it is no longer called when the frontmost window changes.

Passing -1 ($FFFFFFFF) does not change the menu procedure, but does return the address of the current menu procedure.

See "Using Modal Window Calls," earlier in this chapter, for a functional description of all of the calls associated with modal dialogs.

Note `MWSetMenuProc` is a combination of `fmSetMenuProc` and `fmdGetMenuProc` from the Developer Technical Support `fakeModalDialog` tool set.

Note If an NDA sets the menu procedure, 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 $FFFFFFFF
	<—SP

Stack after call

Previous contents	
— *oldMenuProc* —	**Long**—Pointer to previous menu procedure
	<—SP

Errors None.

C

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

`newMenuProc` Pointer to a mew menu procedure. Pass `NIL` to remove any existing procedure; pass -1 to force a return of the current menu procedure without changing it.

`oldMenuProc` Pointer to the previous menu procedure.

MWSetUpEditMenu $680E

MWSetUpEditMenu sets the state of the standard edit menu items based on the frontmost window. The standard edit menu items are Undo, Cut, Copy, Paste, Clear and Close.

If the frontmost window is a desk accessory window, MWSetUpEditMenu enables all of the standard edit menu items. If the frontmost window is not a desk accessory menu, MWSetUpEditMenu enables and disables the items based on the target control in the window:

> If the current target control is a LineEdit control, MWSetUpEditMenu enables Cut, Copy and Clear if any text is selected, and enables Paste if there is a text scrap with a length greater than 0.
>
> If the current target control is an editable TextEdit control, MWSetUpEditMenu enables Cut, Copy and Clear. Paste is also enabled if there is a text scrap with a length greater than zero bytes.
>
> If the current target control is a read only TextEdit control, MWSetUpEditMenu enables Copy, but disables Cut, Paste and Clear.
>
> In all other cases, MWSetUpEditMenu disables Cut, Copy, Paste and Clear.

See "Using Modal Window Calls," earlier in this chapter, for a functional description of all of the calls associated with modal dialogs.

Note MWSetUpEditMenu is very similar to the fmdEditMenu call from the Developer Technical Support fakeModalDialog tool set.

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

When DoModalWindow needs to update a dialog, and you have not provided your own update procedure, it calls MWStdDrawProc. If you override this procedure you may want to call it from your own update procedure to do the default update operations.

MWStdDrawProc calls DrawControls to draw the controls in the current port, which must be a window. If the window frame's fAlert bit is clear and its fFlex bit is set, MWStdDrawProc also draws an alert frame in the content area of the window.

See "Using Modal Window Calls," earlier in this chapter, for a functional description of all of the calls associated with modal dialogs.

Note MWSetDrawProc is very similar to the fmdStdDrawProc call from the Developer Technical Support fakeModalDialog tool set.

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

`ResizeInfoBar` sets the vertical size of a standard window's information bar.

▲ **Warning** Do not use this call with custom windows. ▲

Parameters

Stack before call

Previous contents	
flags	**Word**—Flags (reserved, use 0)
newSize	**Word**—New information bar height
— *windPtr* —	**Long**—Pointer to window
	<—SP

Stack after call

Previous contents	
	<—SP

Errors None.

C

```
extern pascal void ResizeInfoBar(flags, newHeight, windPtr);
Word flags, newHeight;
WindowPtr windPtr;
```

`flags` Reserved; pass 0.

`newHeight` This is the new window bar height, in pixels. The value must be greater than 0.

`windPtr` `windPtr` is a pointer to the window to change.

Chapter 33 **GS/OS Update**

This chapter contains new information about GS/OS. The original reference to GS/OS is in the *Apple IIGS GS/OS Reference*.

Internal Enhancements

- Character I/O speed has been improved. This was achieved by folding the functionality of the QuickConsole INIT into GS/OS itself. As implemented, calls to character I/O devices are special cased so the unnecessary processing is removed from the call overhead.

- Pathnames that start with a digit are handled in a more intelligent way. Prior to System 6.0, GS/OS's pathname processing assumed that any pathname that started with a digit was a pathname that started with a GS/OS prefix number. When this assumption was incorrect, it led to some type of pathname error being returned to the caller. (For example, pathname syntax error, file not found error, and so forth.) GS/OS now looks at the character following the leading digit (or leading two digits) and determines if the character is a valid pathname separator. If the character **is** a valid separator, processing proceeds as it previously did, assuming that the number represents a GS/OS prefix designator. If the character is **not** a valid separator, GS/OS assumes that the leading number is **not** a prefix, and does not try to expand the (non-existent) prefix designator. It simply attaches the value of the default prefix (0: or 8:) to the supplied pathname to determine the full pathname.

  Please note that this does not cover all possible situations. It simply reduces the chances of incorrect assumptions. For example, "7/3/59" is a valid HFS file name. If this file name were passed to GS/OS for processing, it would see the valid pathname separator character following the leading digit, and assume that the "7/" portion of the file name refers to prefix 7. It would then build a full pathname by concatenating the "3/59" portion of the file name to the existing value of prefix 7, thereby producing a full pathname that, in all likelihood, does not match any existing file. On the other hand, file names like "5.0 System Disk" will be processed just like any other file name.

- GS/OS now calls `SysBeep2` for disk requests. When GS/OS displays a dialog asking the user to insert a disk, it calls the new `SysBeep2` routine to give the user an audible signal that a disk is required. Normally this is a standard `SysBeep` sound, but a user may customize this with any sound desired.

- GS/OS automatically detects disk insertions. When GS/OS displays a dialog box asking the user to place a particular volume on line, it continually scans all devices which support removable media, looking for the disk insertion. When a disk is inserted into a device which supports disk insertion notification, GS/OS removes the dialog and continues as if the user had pressed the Return key.

- Improvements have been made in the way GS/OS handles the Standard I/O channels. GS/OS special-cases any `Open` calls on the Standard I/O Channel prefix numbers. If an `Open` call is made using **only** the prefix number associated with one of the standard I/O channels, GS/OS stores the reference number as the standard I/O reference number for that particular I/O channel, supporting the `GetStdRefNum` call.

  If the `Open` call fails with a `fileBusy` error, GS/OS performs the following checks to see if both standard output and standard error output are being redirected to the same file. If any of the following tests fail, a `fileBusy` error is returned (as it has been in the past).

First, GS/OS checks to see if the caller is trying to open prefix 11 or prefix 12. (If it's not one of the standard output channels, it's a true `fileBusy` error).

Next, if the caller is making a ProDOS 16 call, or if the `pCount` of a GS/OS call is less than 4, GS/OS continues with the following checks. In the case of a GS/OS call, the `pCount` **must** be less than 4, since GS/OS cannot return all of the information requested by the larger `pCount`.

Next, GS/OS compares the contents of prefixes 11 and 12. If they are not equal (ignoring character case), we have a true `fileBusy` error.

GS/OS then ensures that the alternate output I/O channel already contains a valid reference number. That is, if the caller is opening prefix 12 (standard error output), the prefix 11 reference number is checked to ensure that standard output has already been opened, and vice versa. If the alternate output channel has already been opened, GS/OS concludes that the caller is trying to redirect both standard output and standard error output to the same disk file. In this case, GS/OS copies the existing reference number into the alternate channel reference number variable, and returns this reference number to the caller. No error is returned in this case, and both the standard output and standard error output reference numbers refer to the same physical disk file.

During a `Close` call, if the supplied reference number equals **both** the standard output and standard error output reference numbers, GS/OS performs a quiet close on the standard error output I/O channel. This means that the internal reference number variable for standard error output will be zeroed (as is done during a normal `Close`), but the physical file will not actually be closed. This is because the standard output I/O channel still refers to the currently open file.

- `GetRefNum` has been updated. The `GetRefNum` call now correctly returns one of the standard I/O channel reference numbers when the supplied pathname refers **only** to the standard I/O channel prefix, no matter what the contents of the prefix are. GS/OS simply recognizes when the supplied pathname is "10:", "11:", or "12:" (or equivalent), and performs an internal `GetStdRefNum` call to retrieve the reference number in question.

- `GetPrefix` and `SetPrefix` have been improved. `GetPrefix` and `SetPrefix` now allow a prefix number of -1 ($FFFF), which represents the '@' prefix. This allows "switcher" type programs to set the '@' prefix properly when the current application is changed.

- There is now a global `OSPublicFlags` variable. This new global variable allows GS/OS to communicate certain events to the rest of the system. This 16-bit variable is located at $E100B8. The variable is supplied for **reading only**, and is not to be modified by anything other than GS/OS.

  Currently, only bit 15 is defined. It is defined as the `NoInits` bit. When the user boots the system and requests that no INITs or DAs be loaded (by holding down a Shift key during the booting process), bit 15 of the `OSPublicFlags` variable is set.

  All other bits within the 16-bit value are currently reserved.

Device Dispatcher

Removable media is now ejected automatically during GS/OS shutdown. Previously, to be friendly, any application which issued an `OSShutdown` call (like the Finder) had to first scan the device list looking for any device that contained removable media, and then issue an `Eject` call to each such device. This functionality has been added to GS/OS, so if a complete shutdown is requested (bit 0 of `shutDownFlag` is 0, as opposed to a restart, where this bit is 1), all ejectable disks are ejected from their respective devices.

Automatic ejection does **not** take place during a system restart on the assumption that, since the user requested the restart, he must have **some** reason for leaving the disks in the drives in the first place. If you wish to eject disks during a restart, set bit 2 of the `shutDownFlag`.

Initialization Manager

The Initialization Manager is the part of GS/OS that handles initializing and formatting disks. It has undergone a number of enhancements for System 6.0. You access the Initialization Manager by calling `EraseDisk` and `Format`. Enhancements to the actual calls are described later in this chapter, in the section "New and Updated Calls."

Addition of LineEdit Item to Request a Volume Name

The Initialization Manager dialog now includes a LineEdit item. The new LineEdit item lets you specify a volume name for a newly formatted diskette. If the `volName` parameter supplied is not null, the Initialization Manager uses the supplied volume name as the default value in the LineEdit control. If `volName` is null, or is absent altogether, the Initialization Manager applies a default name of "Untitled" to the LineEdit item.

GS/OS uses the new `JudgeName` facility to ensure the volume name syntax is proper before formatting the media.

Overview

The Initialization Manager supports both a graphics-based interface and a text-based interface. GS/OS displays the graphics interface when the existing system environment can support the graphic dialog box. This means that the Desk Manager must be active (and by implication, Miscellaneous Tool Set, QuickDraw II, Event Manager, Window Manager, Control Manager, and LineEdit Tool Set), the system must be displaying the Super Hi-Res screen, and the master scan-line control byte must indicate that the screen is in 640 by 200 mode. There must also be 64K of memory available (for use by the tools). If one of the above prerequisites is not met, the Initialization Manager produces its own text-based dialog using the Console Driver.

The Graphics Dialog

GS/OS produces the graphics dialog using standard toolbox calls.

The dialog window displays the name of the device to be used for the operation, as well as the volume name to be used. If the application allows the user to edit the volume name, it appears in a LineEdit item. The user may use standard editing actions (mouse clicks, cursor movement, etc) to specify the volume name. If the name is **not** editable, it is displayed as static text within the window.

The dialog also contains two list items – one for a list of FSTs that support formatting, and another for a list of formatting options supported by the driver. In both cases, if any item is determined to be non-selectable (for example, if the caller specified that a specific FST should be used for the operation, or a particular formatting option is not supported by the currently selected FST), that item is dimmed, and cannot be selectable by the user. Each list can be manipulated using standard desktop interface techniques (for example, mouse clicks, arrow keys, and so forth). The initially-selected FST is the one specified by the caller (if a specific FST was requested), or the FST used to boot the system (if the caller did not specify a specific FST). The initially-selected format will be the format specifically requested by the currently selected FST (if appropriate), or the default format as specified by the device driver.

One of the lists or the LineEdit item can be made the active item either by clicking on the item or by using the Tab key to cycle through the items. The currently active item is specified by either a bold outline around the item (in the case of the List items), or by a flashing cursor or inverse text in the LineEdit item.

There are two buttons at the bottom of the window – Cancel and either Initialize or Erase, depending on which operation is being performed. The Cancel button is activated either with a click or by pressing the Esc or Command-. (Command-period) keys. This aborts the operation before any changes are made to the disk. The Initialize or Erase button is activated either with a click or by pressing Command-Return. In this case, the operation proceeds. The button is deactivated (the button title will be dimmed) when the LineEdit item does not contain any text, or if either of the list items does not have a currently selected item.

Various messages are displayed in the area between the list items and the buttons. The initial message tells the user the Initialize or Erase operation will destroy any data on the target disk. This initial message is cleared (or, more likely, is replaced with another message) as soon as the user performs some action within the dialog window.

If the currently selected FST does not support the volume size described by the currently selected format (for example, ProDOS with a 40 megabyte drive), GS/OS displays a message warning the user that the file system will not use the entire capacity of the drive. This message remains on screen until the user selects another format or FST (if the new selections don't also display the message), or until the user makes the LineEdit item the currently active item. This message is referred to as the "Too Big" message in the following paragraphs.

Normally, the message area displays the text supplied by the currently selected FST's `JudgeName` facility. GS/OS updates this message whenever a different FST is selected. If the "Too Big" message is displayed, it is replaced by the `JudgeName` message only when the LineEdit item is the active item. (The reasoning is that users should have some kind of prompt to help them enter a valid volume name while they're actually entering the name.)

When the user selects the Initialize or Erase button, the Initialization Manager performs a `JudgeName` call to the currently selected FST to be sure the volume name is acceptable to the file system. If the `JudgeName` call returns with no error, the operation is carried out. If the `JudgeName` call says the name is no good, the Initialization Manager determines what step to take next depending on whether or not the user can edit the volume name. If not, the Initialization Manager simply continues, and leaves it up to the FST to generate the pathname syntax error for return to the caller. If the volume name **is** editable, the Initialization Manager tells the user the pathname is invalid, and then allows the user to edit the volume name or make a new FST selection.

The Text Dialog

The text dialog is similar in appearance and operation to the graphics dialog. The main difference is the lack of any mouse support. All input and control manipulation must be performed using the keyboard.

The volume name LineEdit item is implemented with the Console Driver's user input routine. All editing keystrokes supported by the user input routine can be used when specifying the volume name. See the *Apple IIGS GS/OS Reference*, page 252 for a list of these commands.

Like in the graphics dialog, you can use the Tab key to cycle between the LineEdit item and the List items. The active item has an active cursor, or the title is displayed in inverse text.

Use the up and down arrow keys to move the selection bar from item to item in the lists. Note that the selection bar does not wrap within the list – once at the bottom of the list, it will stay at the bottom until the user presses the up arrow key. If there are more than four items to be displayed in the list, the items scroll up and down as appropriate to ensure that the selection bar remains visible on the screen.

List items that are not selectable (and would be dimmed in the graphics dialog) have parentheses around the list item. GS/OS does not allow the selection bar to be placed on such items.

Like the graphics dialog, the Esc key is used to cancel the dialog. Command-. (Command-period) is not a cancel command, since that would preclude its recognition as a user input routine editing command. Command-Return accepts the current selections and proceeds with the `Format` or `EraseDisk` operation, just as accepting the graphics dialog does.

One additional key press recognized by the text dialog but not the graphics dialog is the Command-? sequence. (Command-/ will do the same thing.) This key press forces the display of the `JudgeName` prompt if the volume name is editable – there is no easy way to recognize the first keystroke within the LineEdit item like there is with the graphics LineEdit item. By supplying this help key, the Initialization Manager can "hand-hold" the user through the entry of a valid volume name.

GS/OS Booting Changes

Display System Software Version Number on Graphics Splash Screen

GS/OS displays an additional string in the Super Hi-Res splash screen; it shows the current version number of the System Software.

Disabling Setup Files and Desk Accessories

At the beginning of the boot process, GS/OS checks the state of the Shift keys. If either Shift key is depressed, GS/OS displays the message "No Inits/DAs" on the graphics splash screen, and then skip loading of any non-system Initialization Programs and all Desk Accessories. Tool.Setup and Resource.Mgr are **always** loaded.

Changes to Program Launching and Quitting

Quit Call Invokes Program Launcher

GS/OS now allows applications to avoid quitting to themselves if they are the start-up application. If bit 12 of the `flags` word supplied in the `Quit` call is clear, GS/OS automatically executes the Apple IIGS Program Launcher if no application User ID remains on the system Quit stack **and** if GS/OS is active. If ProDOS 8 is active, the system works like it used to, relaunching the default Start application.

If bit 12 of the flags word is set, the application is simply relaunched.

Leave Super Hi-Res Screen Active During Application Launch

GQuit now turns the Super Hi-Res screen off only when quitting to an application that is **not** a desktop application, as defined by its auxiliary file type. Previously, GS/OS did not guarantee any particular screen mode (text vs. Super Hi-Res) when launching an application. Now, if the application being launched does not support the desktop interface (as defined by having $DB in bits 15-8 of the auxiliary type and bit 1 of the lower byte set), GS/OS displays the text screen.

BASIC.Launcher Functionality

When GS/OS launches a ProDOS 8 application that has a start-up buffer (as defined in section 5.1.5 of the *ProDOS 8 Technical Reference Manual*), GS/OS attempts to extract the first pathname from `MessageCenter` message $0001 or $0011 and place it in the ProDOS 8 application's start-up buffer. This means ProDOS 8 applications no longer have to call `MessageCenter` to retrieve pathnames chosen from GS/OS program launchers. This makes the old BASIC.Launcher program obsolete, so it is no longer part of system software.

ProDOS 8 programs that use a start-up buffer for material other than pathnames may not work with this new mechanism, and should be revised.

Error Handling During Application Launch

When GS/OS receives some kind of error from the System Loader after an `InitialLoad` or `Restart` call, it no longer forces a system death message. Instead, it displays a dialog notifying the user of the problem, and gives the user a choice between restarting the system or returning to the last application that has its user ID on the Quit stack.

System Loader and ExpressLoad

The System Loader and ExpressLoad are the parts of GS/OS that load applications. This section describes the changes made to these loaders.

Merge of System Loader and ExpressLoad

The old System Loader and ExpressLoad have been merged into a single System Loader containing the functionality of both of the previous loaders. Where possible, duplicate (or near-duplicate) subroutines have been reduced to a single subroutine which both loaders take advantage of. This allows the removal of the ExpressLoad file from the system, while only using an additional 5.5K for the System Loader. In addition, this 5.5K is located in the upper reaches of bank $01 of memory, freeing up the non-special memory previously occupied by ExpressLoad.

Don't Launch Zero-length Applications

Applications are no longer launched if there is nothing in the data fork. Prior to System 6.0 it was possible to `InitialLoad` a file with a zero-length data fork. The System Loader has been modified to check the length of the file prior to completing the operation. If a non-OMF file is found (which a zero-length fork implies), the Loader returns an error to the caller.

`LGetPathname` and `LGetPathname2` Enhancements

`LGetPathname` and `LGetPathname2` have always returned a pointer to the pathname, but the returned pointer may or may not have been pointing into a private data structure maintained by the Loader. These calls have been enhanced so they now allocate a locked block of memory and copy the pathname into this locked block. The pointer returned from the calls points to this block of memory, and remains valid until the next System Loader call, when the memory block containing the pathname is disposed (using a `DisposeAll` call on a private user ID).

If you wish to claim ownership of the memory block containing the pathname, use the pointer as the argument to a `FindHandle` call, and then make a `SetHandleID` call on the returned handle using the caller's user ID.

`GetLoadSegInfo` and ExpressLoad

The ExpressLoad portion of the System Loader now handles a `GetLoadSegInfo` call when it applies to a load segment that has been loaded by ExpressLoad. Since ExpressLoad doesn't actually maintain the memory segment table, the returned memory segment table entry is synthesized by ExpressLoad.

Segment Loading and Special Memory

When loading a load segment the System Loader always requests non-special memory first. If this fails, and the `SpecialMemoryFlag` of the `InitialLoad` call allows it, the memory allocation request is made again, this time allowing special memory.

Drivers

This section describes changes to the various drivers included with System 6.0. New drivers or drivers with extensive changes are covered in separate sections following this one.

SCSIHD.Driver

- The SCSIHD driver now allows write protection and write enable of fixed hard disk partitions.
- The SCSIHD driver now allows read protection and read enable of fixed hard disk partitions.

AppleDisk3.5

- This driver now supports 800K and 1440K drives attached to the 3.5 Drive Controller Card.

Console.Driver

- Some dead code has been removed.

- A new device-specific `DStatus` call has been added. The status code for this new call is $8007, and it is named `GetVectors`. The status list must be large enough to contain 8 bytes, and the `requestCount` must equal 8.

  The purpose of this call is to query the Console Driver about the locations of two single character Input and Output subroutines. An application may retrieve these addresses from the console driver, and then call the subroutines to perform single character input and output, similar to the methods used when using the built-in monitor firmware.

  The `COut` vector (for Character Out) is returned in the first four bytes (first long word) of the status list. The `COut` subroutine is used to output a single character to the current display device. The character to be displayed must be placed in the low byte of the accumulator before calling `COut`. The subroutine may be entered in either 8- or 16-bit mode, as the Console Driver changes the register width internally as needed. All registers and register widths are preserved by the call.

  The `KeyIn` vector (for Keyboard Input) is returned in the last four bytes (second long word) of the status list. The `KeyIn` subroutine reads a single key press from the keyboard. The subroutine may be called in either 8- or 16-bit mode, as the Console Driver changes the register width internally as needed. All registers (except the accumulator) and register widths are preserved by the call. On exit, the low byte of the accumulator contains the ASCII code for the key press, and the high byte of the accumulator will contain the state of the keyboard modifiers pseudo-register ($C025). Note that, unlike the Monitor subroutine of the same name, `KeyIn` does **not** supply an input cursor on the screen. Like its namesake, though, it **does** increment the pseudo-random number stored at locations $4E-$4F on absolute page zero.

  The contents of memory pointed to by these two vectors consist of 24-bit `JMP` instructions to the actual `COut` and `KeyIn` handlers. This allows an application to intercept the vectors to handle single-character I/O in whatever fashion they wish (for example, to support single character I/O in a graphics environment when supporting applications that normally communicate through the console driver).

- The `ResetTrap` device-specific control call now resets the new `COut` and `KeyIn` vectors in addition to the original Console trap vector.

UniDisk3.5

- This driver now supports UniDisk 3.5 drives attached to the 3.5 Drive Controller Card.

AppleDisk5.25 Driver

New Features

- The media I/O routines have been rewritten to improve performance.
- This driver now supports restartability.
- There is a new `DControl` subcall (`SetNextVolnum`) and a new `DStatus` subcall (`GetLastVolnum`).

DControl and DStatus Subcalls

`GetLastVolnum` (DStatus Subcall)

Status code = $8001

This device-specific status call returns the volume number (read from the address field of a sector) of the last accessed disk in the status list. The `requestCount` field of the parameter block must be 2. If no disk access has occurred prior to issuing this call, a volume number of 0 will be returned.

The status list has the following format:

$00	*volume*	**Word**—Volume number

`SetNextVolnum` (DControl Subcall)

Control code = $8001

This device specific control call allows the caller to specify the volume number to use the next time the `Format` control call is made. The volume number specified must be in the range 0-254. If 0, it will default to 254.

The `Format` call will reset this to zero after performing the format, so the call should be made before each `Format` call to avoid the default volume number.

The status list has the following format:

$00	*volume*	**Word**—Volume number

SCSI Drivers

Several features have been added to the SCSI Drivers for System Software 6.0. The new features allow reading and setting partitions and driver information, as well as controlling some status information for the device (such as whether the device is read or write enabled). These new features are used via `DControl` and `DStatus` calls; the new subcalls are described later in this section.

For general information about `DControl` and `DStatus` calls, see *Apple IIGS GS/OS Reference*.

For general information about disk partitioning, see *GS/OS Device Driver Reference*.

`SetDiskInfo` (DControl Subcall)

Control code = $F000

`SetDiskInfo` sets the Driver Descriptor Map (DDM), Partition Map, and driver for a device. The call must be made to the head device of a partitioned device (in other words, to the device itself, not one of the partitions).

See `GetDiskInfo` (a `DStatus` subcall) earlier in this section for the format and use of the status list.

`GetVolumeStatus` (DStatus Subcall)

Status code = $F001

This call is used to read the current status for a volume. The volume can be a partition on a larger physical device.

The status list has the following format:

$00	*flags*	**Word**—Indicates current read, write status

Upon return, bit 4 will be set if the device is read enabled and bit 5 will be set if the device is write enabled. See `SetVolumeStatus` (a `DControl` subcall) for a way to change the volume status.

`GetDiskInfo` (DStatus Subcall)

Status code = $F000

`GetDiskInfo` reads the Driver Descriptor Map (DDM), Partition Map, and driver from a device. The call must be made to the head device of a partitioned device (in other words, to the device itself, not one of the partitions).

The status list has the following format:

$00	*driverNumber*	**Word**—Device driver number
$02	*DDMBufferLen*	**Word**—DDM buffer length
$04	*DDMTransfer*	**Word**—DDM transfer count
$06	— *DDMBuffPtr* —	**Long**—DDM buffer pointer
$0A	*bufferLen*	**Word**—Driver buffer length
$0C	*transfer*	**Word**—Driver transfer count
$0E	— *buffPtr* —	**Long**—Driver buffer pointer
$12	*dataBufferLen*	**Word**—Driver data buffer length
$04	*dataTransfer*	**Word**—Driver data transfer count
$06	— *dataBuffPtr* —	**Long**—Driver data buffer pointer

`driverNumber` This is the driver number being referenced. This number must be in the range $01 to $0F. It is used as an index into the Driver Descriptor Map (DDM) driver list to determine which Apple_Driver is being referenced.

The structure of the DDM and Driver Info data can be found in the *GS/OS Device Driver Reference*.

`DDMBufferLen` This is a word value indicating the size of the DDM buffer. It must be $0200.

`DDMTransfer` This is a result word returned by the driver. If data is transferred successfully, this value will be set to $0200.

`DDMBuffPtr` This is a pointer to the buffer allocated by the caller to contain the DDM data. It should be $0200 bytes long.

`bufferLen` This is a word value indicating the size of the Driver Buffer. It must be $0200.

`transfer` This is a result word returned by the driver. If data is transferred successfully, this value will be set to $0200.

`buffPtr` This is a pointer to the buffer allocated by the caller to contain the driver information. It should be $0200 bytes long.

`dataBufferLen` This is the size of the driver data buffer. It must be large enough to contain all the driver data, and is a multiple of $0200.

`dataTransfer` This is a result word returned by the driver. If `bufferLen` is large enough to hold the driver, this value is set the the number of bytes read, and the data buffer is filled in. If `bufferLen` is too small, this value is set to the minimum allowed buffer length, and no other information is returned.

`dataBuffPtr` This is a pointer to the buffer allocated by the caller to contain the driver data. This is different from the driver information above in that the driver information is data from the Partition Map Entry for the driver and the Driver Data is the actual driver.

Note Set all three fields (the buffer length, transfer count, and buffer pointer) to zero if the information is not needed.

`SetVolumeStatus` (DControl Subcall)

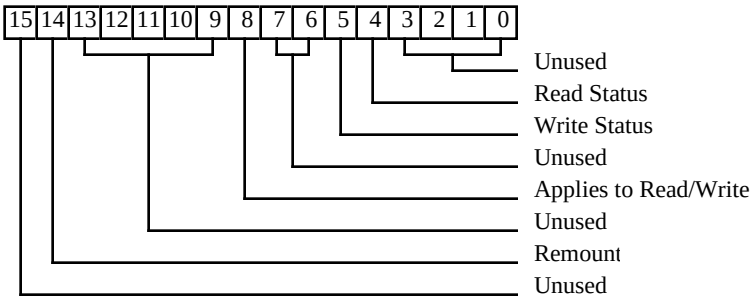
Control code = $F001

This call is used to set the current status for a volume. Reading or writing can be enabled or disabled, and the volume can be remounted with this call.

The volume can be a partition on a larger physical device.

The status list has the following format:

$00	*flags*	**Word**—Indicates current read, write, remount status



Bit 14 is the remount bit; set this bit to force GS/OS to remount the volume.

In some cases you may want to remount the volume without changing it's read/write status. Bit 8 controls whether the call applies to the read/write status for the volume; if the bit is set, the read/write bits are used, but if bit 8 is clear, the read/write bits are ignored.

Assuming bit 8 is set, bit 4 controls the read status for the volume. If bit 4 is set, the volume will be marked as read enabled, and calls that read the device will work. Disabling reading for a volume is analogous to clearing the read enable bit for a file.

Assuming bit 8 is set, bit 5 controls the write status for the volume. If bit 5 is set, the volume will be marked as write enabled, and calls that write to the device will work. Disabling writing for a volume is a software equivalent to write-protecting the volume (but the write-protect bit works even on devices like hard disks that don't have a physical write-protect tab).

File System Translators

This section describes the changes made to the various File System Translators (FSTs). New FSTs are described in separate sections following this one.

`optionList` Changes

The definition of the second field of the `optionList` has been changed. Previously, this field was undefined when doing a `SetFileInfo` call and was set to the actual data size by the FST when doing a `GetFileInfo`, `Open` or `GetDirEntry` call. This field must now be set to the actual data size when doing a `SetFileInfo` call so the FST can determine if the data it expects is really there.

HFS, AppleShare, and ProDOS Use Common `optionList` Format

The HFS FST, AppleShare FST and ProDOS FST now check both the first field (buffer size) and the second field (data size) when doing a `SetFileInfo` call. If the buffer size is less than 36, or if the data size is greater than 32, then the `optionList` data is ignored. If the sizes are OK, the FSTs then check the FST ID to make sure that it's either a $0001 (ProDOS), $0006 (HFS) or $000D (AppleShare). If so, the `optionList` data is assumed to be 32 bytes of Macintosh Finder information.

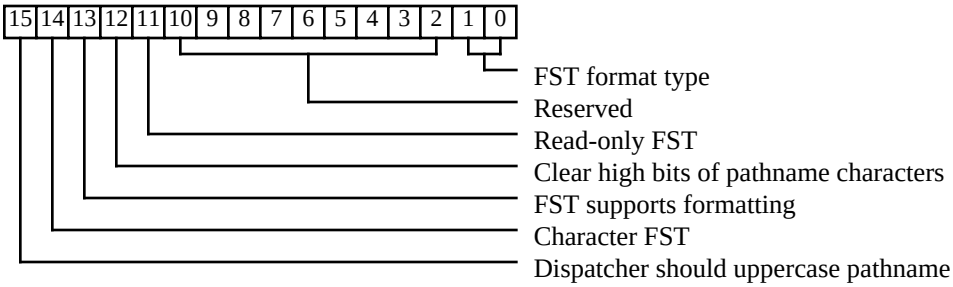
The HFS FST and the AppleShare FST copy the data into the file's catalog record. The ProDOS FST copies the Macintosh Finder information into the extended key block if the file is an extended file; otherwise, the data is ignored. The data is stored immediately after the data fork mini-entry, so it begins at offset $08. The format is:

$00	*size1*	**Byte**—Entry size (18)
$01	*type1*	**Byte**—Entry type (1)
$02	*FInfo*	**16 bytes**—first 16 bytes of Macintosh Finder information
$12	*size2*	**Byte**—Entry size (18)
$13	*type2*	**Byte**—Entry type (2)
$14	*xFInfo*	**16 bytes**—second 16 bytes of Macintosh Finder information
$24	*endMark*	**Byte**—0

This is similar to the format used by the ProDOS File System Manager on the Macintosh. The only difference is that Macintosh ProDOS does not always store both entries. There could be just the `FInfo` entry or just the `xFInfo` entry. The GS/OS ProDOS FST checks for this situation when returning the Macintosh Finder information during a `GetFileInfo`, `Open` or `GetDirEntry` call. If only one of the entries exists, the missing 16 bytes are returned as all zeros. The data size is still set to 32.

FST Attributes

The FST attributes returned by the `GetFSTInfo` call were never documented fully. Here is the format for the flags word:



bit 15 — If this bit is set the GS/OS call dispatcher converts the input pathname to all uppercase characters. Otherwise, the pathname is passed to the FST as supplied by the caller.

bit 14 — If this bit is set, the FST is designed to communicate with character devices only. If the bit is clear, the FST communicates with block devices only.

bit 13 — If this bit is set, the FST supports formatting.

bit 12 — If this bit is set, the GS/OS call dispatcher clears the high bit (bit 7) of characters within the input pathname prior to calling the FST. If the bit is clear, the pathname will be passed as supplied by the caller.

bit 11 — If this bit is set, the FST only supports read operations. Modification of disks is not supported.

bits 1-0 These bits specify the format type that this FST supports:

- 00 Universal format (such as High Sierra).
- 01 Apple format (such as HFS).
- 10 Non-Apple format (such as MS DOS).
- 11 Apple II format (such as ProDOS, DOS 3.3, and Pascal).

Character FST

- During start up the Character FST now locates the Console Driver and determines the location of an internal entry point (using the `AddTrap` and `ResetTrap` calls). When a `Write` call is received with the Console Driver as the destination, the Character FST sends the call directly to the internal entry point, bypassing the Device Dispatcher.

High Sierra FST

- The High Sierra FST supports the enhanced `Volume` call.

ProDOS FST

- The ProDOS FST supports the new `JudgeName` call and the enhanced `ChangePath` and `Volume` calls.
- The ProDOS FST supports HFS-style `optionList` buffers. It stores HFS-style directory information in the key blocks of extended files.

AppleShare FST

Changes in System Software 6.0

- There are two new FST-specific calls: `GetDefaultPrivileges` and `SetDefaultPrivileges`. They get and set the default access levels for new folders created on AppleShare file servers.
- The AppleShare FST supports the new `JudgeName` call and the enhanced `ChangePath` and `Volume` calls.

New File Type Conversions

The AppleShare FST now does some file type conversion in addition to that done by the file server. These new conversions are performed solely by the AppleShare FST and override the normal conversions described in *Apple IIGS GS/OS Reference*. One of the conversions applies to GS/OS-aware applications so that their auxiliary types may be preserved (since they contain important flags). The other conversions are for files that contain information that both an Apple IIGS and a Macintosh can access.

When setting the file type and auxiliary type (such as in a `SetFileInfo` or `Create` call), and no `optionList` with Macintosh Finder information is supplied, the Macintosh file type and creator type are set as shown in the table below. Macintosh Finder information present in an `optionList` always overrides these conversions. If the ProDOS file type and auxiliary type do not match an

entry in the table, no conversion is performed and the Macintosh type and creator type are not explicitly changed (although the server's default file type conversions still apply).

ProDOS File Type	Auxiliary Type	Macintosh Creator Type	File Type
$B3	$DBxy	"pdos"	"p" $B3 $DB $xy
$D7	$0000	"pdos"	"MIDI"
$D8	$0000	"pdos"	"AIFF"
$D8	$0001	"pdos"	"AIFC"

When getting the file type and auxiliary type (such as in a `GetFileInfo`, `Open`, or `GetDirEntry` call), if the server returns an Apple II file type and auxiliary type with both equal to zero, they are derived from the Macintosh file type and creator type as given in the following table. The Macintosh creator type is ignored (any value is acceptable). If the Macintosh type does not match an entry in the table below, no conversion is done and the Apple II file type and auxiliary type are both returned as zeros.

Macintosh File Type	ProDOS File Type	Auxiliary Type
"MIDI"	$D7	$0000
"AIFF"	$D8	$0000
"AIFC"	$D8	$0001

Macintosh Finder information

The `optionList` used by the AppleShare FST contains 32 bytes denoted as Macintosh Finder information. The contents of the Macintosh Finder information are not modified or used in any way by the FST. AppleShare, the file system, and the FST simply provide room for 32 bytes per file or folder. The Macintosh Finder is solely responsible for the definition and interpretation of the contents of these bytes.

If you plan to use the information stored here, it is up to you to make sure the values make sense (and to do something appropriate if they don't make sense). Beware that the values are likely to be zero when an object is created with something other than the Macintosh Finder, and may not have sensible values until you specifically change them via the Macintosh Finder. Even worse, different versions of the Macintosh Finder may store different values in the same fields.

For a description of the Macintosh Finder information, see *Inside Macintosh, Volume IV*.

System Calls

This section describes differences between parameters in the AppleShare FST and the ProDOS FST. Please see *Apple IIGS GS/OS Reference* for more detailed information about these calls. Any calls not documented here behave as specified in *Apple IIGS GS/OS Reference*.

`Create` ($2001)

When a directory is created, its access privileges, owner, and group are set based upon the settings of the last `SetDefaultPrivileges` call. This lets you override the server's default of exclusive access to the owner only. Note: if the default owner or group name is not valid for the server on

which the folder is created, that setting will retain its default value (for example, the user that created the folder or their primary group) and no error will be returned.

`FSTSpecific` ($2033)

There are two new FST specific calls, and one that has been modified. In all cases, the FST number must be $0D (AppleShare). A command of $0000 is invalid. Commands $0010 through $FFFF are reserved.

`GetDefaultPrivileges` (AppleShare `FSTSpecific` Subcall)

Command $000E is the `GetDefaultPrivileges` command. It returns the default access privileges, owner, and group of folders created with the `Create` command.

$00	*pCount*		**Word**—Input value (minimum of 3)
$02	*fileSysID*	**1**	**Word**—Input value; $0D
$04	*commandNum*	**2**	**Word**—Input value; $000E
$06	— *flags* —	**3**	**Long**—Output value; field use flags
$0A	— *access* —	**4**	**Long**—Output value; access rights
$0E	— *owner* —	**5**	**Long**—Input value; GS/OS output string
$12	— *group* —	**6**	**Long**—Input value; GS/OS output string

The format of the parameter list is the same as the `GetPrivileges` command except that the `pathname` field is now used for flags: bits 0,1, and 2 are set if the access rights, owner name, or group name (respectively) will be set with the `Create` command. The minimum `pCount` is 3 so that you can determine which fields will be set, but without determining what values they get set to. If you supply a `NIL` for either the owner name or group result buffer pointers, that string is not returned.

`GetUserPath` (AppleShare `FSTSpecific` Subcall)

This subcall is not new. See *Apple IIGS GS/OS Reference*, page 355 for the original description. Only the changes are described here.

In System Software 6.0 `GetUserPath` checks for the presence of the user's folder. It builds the path internally and then verifies that the folder actually exists and is indeed a folder. If the folder does not exist (or the path exists but is a file), the data unavailable error ($60) is returned. This check does not guarantee you actually have either read or write access to the folder, just that it exists.

This means that the behavior of the "@" prefix has changed (since it uses the `GetUserPath` call). The "@" prefix is now set to the user's folder only if it actually exists. If it does not exist, the "@" prefix will revert to the folder containing the application (the same as if the application was launched from a non-AppleShare volume, or if there was some other error determining the user path).

As a matter of clarification, the returned string is of the form

```
:UserVolume:Users:UserName
```

where `UserVolume` is the name of a volume that is marked as a user volume, and `UserName` is the name of the registered user used when logging on to that server or the string “<Any User>” if they logged on as a guest. If more than one volume is marked as a user volume, the one with the lowest device number is used.

`SetDefaultPrivileges` (AppleShare `FSTSpecific` Subcall)

Command $000F is the `SetDefaultPrivileges` command. It allows you to set the default access privileges of folders created with the `Create` command.

$00	*pCount*		**Word**—Input value (minimum of 3)
$02	*fileSysID*	**1**	**Word**—Input value; $0D
$04	*commandNum*	**2**	**Word**—Input value; $000F
$06	— *flags* —	**3**	**Long**—Output value; field use flags
$0A	— *access* —	**4**	**Long**—Input value; access rights
$0E	— *owner* —	**5**	**Long**—Input value; GS/OS output string
$12	— *group* —	**6**	**Long**—Input value; GS/OS output string

The format of the parameter list is the same as the `SetPrivileges` command except that the `pathname` field is used for flags: bits 0, 1, and 2 should be set if you want to set the access rights, owner name, or group name (respectively). The minimum `pCount` is 3. If the group name is not supplied, the empty string (no group) will be used. If the owner name is not supplied, the empty string (which indicates a guest) will be used. If null pointers are supplied for the owner name or group name, empty strings will be used (meaning guest and no group). If the access rights are not supplied, but its bit is set in the flags, you get an invalid parameter error.

The owner and group names must be less than 32 characters. If the owner name is too long, error $7E (bad user name) will be returned. If the group name is too long, error $7F (bad group name) will be returned.

The owner and group names are GS/OS strings with a leading length word, not the pseudo-result buffer used by SetPrivileges. If you have code that currently uses `SetPrivileges` and you want it to use `SetDefaultPrivileges`, you could just point to two bytes into the pseudo-result buffer (at the actual length word).

`SetFileInfo` ($2005)

The ProDOS file type and auxiliary type will be set to the values given in the call; by default, the Macintosh creator type will be set to “pdos” and the Macintosh file type will be derived according to the rules above. The `optionList` data is the same as for the `GetFileInfo` call, except that only the Macintosh Finder information is used (the other fields cannot be set); any data past the Macintosh Finder information field is ignored.

If the `fileSysID` field is not the same as AppleShare’s file system ID ($0D), the HFS file system ID (6), or the ProDOS file system ID (1), the `optionList` is ignored. All FSTs return their file system ID in the first word of the `optionList` and ignore setting of the `optionList` if they do not understand that file system’s `optionList` format. This allows applications to always get and set the `optionList` as part of the copying process even when copying from one file system to

another. The AppleShare, HFS, and ProDOS FSTs all understand each other's `optionList` formats.

DOS 3.3 FST

This section describes the Apple II DOS 3.3 file system translator for GS/OS.

Overview

The DOS 3.3 File System Translator lets GS/OS use Apple II DOS 3.3 disks. The FST accepts standard calls from GS/OS and executes them on DOS 3.3 disks. The FST operates with 5.25" disks only.

The DOS 3.3 FST is a read-only FST and does not execute any call which requires writing to a disk.

From a user's point of view, the DOS 3.3 FST can be used with the Finder to copy files from DOS 3.3 disks to disks formatted under other file systems (typically ProDOS) and to directly access DOS 3.3 data files from any program that makes GS/OS file calls. Note that AppleSoft programs, which can normally be launched from the Finder, cannot be launched directly from a DOS 3.3 disk (the Finder launches BASIC.System under ProDOS 8, and ProDOS 8 can only find ProDOS and AppleShare volumes). However, an AppleSoft program may be copied from a DOS 3.3 disk to a ProDOS disk and then launched from the ProDOS disk. In this situation it is possible that the AppleSoft program is incompatible with ProDOS and will not run properly, but many programs will work fine.

The DOS 3.3 FST normally strips off bit 7 from characters in text files, but you can use an `FSTSpecific` subcall to prevent the translation. Also, random-access text files contain no record length information, so they appear to be sequential-access to a GS/OS application.

File Types

The following table lists the file types supported by DOS 3.3 and their equivalent GS/OS file types:

DOS 3.3	GS/OS Equivalent		
Text	$04	TXT	ASCII text
Integer BASIC	$FA	INT	Integer BASIC
AppleSoft BASIC	$FC	BAS	AppleSoft program
Binary	$06	BIN	General binary
S-type	$00	——	Typeless file
Relocatable object module	$FE	REL	EDASM relocatable code
A-type	$00	——	Typeless file
B-type	$00	——	Typeless file

The typeless files have no defined function under DOS 3.3 and therefore no GS/OS equivalent. See the description of the `GetFileInfo` call for a discussion of special interpretations for certain pieces of information for those files.

The root directory, which is not interpreted as a file under DOS 3.3, is given a file type of $0F, directory file.

Auxiliary Types

Files which contain a load or ORG address in the header at the beginning of the file (for example, a REL file), will have that value returned in the `auxType` parameter of GS/OS calls.

File Names

DOS 3.3 file names may begin with an upper or lower case alphabetic character or any of the following characters from the standard keyboard:

```
@ \ [ ] { } _ ` ~ | ^
```

After the leading character, any 8-bit ASCII character except a comma (,) may appear in a file name, with a maximum name length of 30 characters. The last non-space character is interpreted as the last character in the file name. Since the colon (:), which can never appear in a GS/OS file name, is a legal character under DOS 3.3, colons in file names are changed into commas before being returned to the caller.

Call Functions

The DOS 3.3 FST accepts standard block FST calls from GS/OS. Each call with notable DOS 3.3-specific behavior is discussed below.

`ChangePath` ($2004)

This call requires write operations and is not supported; it always returns a write protected error ($2B).

`ClearBackupBit` ($200B)

This call requires write operations and is not supported; it always returns a write protected error ($2B).

`Create` ($2001)

This call requires write operations and is not supported; it always returns a write protected error ($2B).

`Destroy` ($2002)

This call requires write operations and is not supported; it always returns a write protected error ($2B).

`EraseDisk` ($2025)

This call requires write operations and is not supported; it always returns a write protected error ($2B).

`Flush` ($2015)

The FST does nothing during this call except verify that the `pCount` (if any) is correct .

Errors `$43` `invalidRefNum` Invalid reference number

`Format` ($2024)

This call requires write operations and is not supported; it always returns an invalid FST operation error ($65).

`FSTSpecific` ($2033)

The DOS 3.3 FST supports these two FST-specific calls:

`get_text_mode` (DOS 3.3 `FSTSpecific` Subcall)

Return the current value of the `text_mode` parameter.

$00	*pCount*		**Word**—Input value; must be 3
$02	*fileSysID*	**1**	**Word**—Input value; $0002 (DOS 3.3 FST)
$04	*commandNum*	**2**	**Word**—Input value; $0002
$06	*text_mode*	**3**	**Word**—Result; see `set_text_mode` for possible values

`set_text_mode` (DOS 3.3 `FSTSpecific` Subcall)

Set the value of the `text_mode` parameter.

$00	*pCount*		**Word**—Input value; must be 3
$02	*fileSysID*	**1**	**Word**—Input value; $0002 (DOS 3.3 FST)
$04	*commandNum*	**2**	**Word**—Input value; $0001
$06	*text_mode*	**3**	**Word**—Input value

text_mode A value of $0000 tells the FST not to clear high bits in bytes transferred from text files, while a value of $0001 (the default) tells the FST to clear the high bits.

`GetFileInfo` ($2006)

This call will return information about the specified file. Certain fields in the parameter block are treated specially, as follows:

`access` Returned as $C3 for an unlocked file, $01 for a locked file.

`fileType` See the discussion of file types, earlier in this section.

`auxType` This field retains its traditional definition for the files whose types are similarly defined in DOS 3.3 and GS/OS, except for text files; the `auxType` field for text files under GS/OS normally specifies the record length for random-access data, where a value of 0 signifies sequential data. Since DOS 3.3 keeps no record length information in its text files, this field is always returned as 0 for these

files. For files whose DOS type has no equivalent in GS/OS, like type-A, type-B and type-S, this field is returned as 0.

`storageType` Returned as $000F for the root directory, $0001 for all other files.

Dates and times Always returned as 0, "unknown."

`optionList` The `optionList` is largely unused by the DOS 3.3 FST. The only thing that is filled in is the required FST ID field.

`eof` This is the number of bytes of actual user data contained in the file. Note that the actual data accounted for by this value may be a subset of the data that physically belong to the file under DOS 3.3 because a certain number of bytes may represent information which is considered directory information under GS/OS. For text files, this is the number of bytes up to and including the last non-zero character in the last data sector of the file. For file types which were never defined, every byte of every data sector is treated as user data, so for these types the EOF will always be a multiple of 256. For the root directory this is the number of bytes occupied by the VTOC and all of the directory sectors of that disk (normally $1000).

`blocksUsed` This is the number of 256-byte blocks occupied by the file on disk.

`resourceEOF` This field is always returned as zero.

`resourceBlocks` This field is always returned as zero.

Note For file types with a meaningful value for `auxType`, functionally the FST must read the first data sector of the file to find the `auxType` value. The same is true for the EOF of AppleSoft, Integer BASIC, binary, and relocatable object files.

For text files there is no EOF value stored at all, so the FST places the EOF just past the last non-zero byte in the file. Since this requires examination of the last data sector in the file, the FST must normally read in every track/sector list for that file to determine its EOF.

Errors

$10	devNotFound	Device not found
$27	drvrIOError	I/O error
$28	drvrNoDevice	No device connected
$2E	drvrDiskSwitch	The disk has been switched
$40	badPathSyntax	Invalid pathname syntax
$44	pathNotFound	Subdirectory does not exist
$45	volNotFound	Volume not found
$46	fileNotFound	File not found
$4A	badFileFormat	Version error (incompatible file type)
$52	unknownVol	Unknown volume type
$53	paramRangeError	Parameter out of range
$58	notBlockDev	Not a block device

`Read` ($2012)

This call attempts to read a specified number of bytes from a file. The FST ignores length and load address information which may be stored before the actual data image in the native format; the calling routine will receive what it expects from any FST.

If data are being transferred from a text file, their values depend on the setting of the `text_mode` parameter (see FST-specific calls). If `text_mode` has the value $0000, bytes are transferred as-is to the user buffer. If `text_mode` has the value $0001 (the default), bytes are transferred to the caller with their high bits cleared.

Errors	`$27`	`drvrIOError`	I/O error
	`$2E`	`drvrDiskSwitch`	The disk has been switched
	`$43`	`invalidRefNum`	Invalid reference number
	`$4C`	`eofEncountered`	End of file encountered
	`$4E`	`invalidAccess`	Access not allowed

`SetFileInfo` ($2005)

This call requires write operations and is not supported; it always returns a write protected error ($2B).

`Volume` ($2008)

DOS 3.3 does not support volume names, but does support volume numbers in the range 1-254. However, most DOS users do not bother to assign a volume number when they format their disks, and so most DOS disks in the world have the default volume number 254. Since DOS 3.3 disks do not have real names, and volume numbers are not unique, the DOS 3.3 FST invents a name for each disk by hashing the directory contents, giving volume names of the form "`DOS 3.3 vXXXX`", where `XXXX` is a decimal number from 0000 to 9999.

Errors	`$10`	`devNotFound`	Device not found
	`$11`	`invalidDevNum`	Invalid device number
	`$27`	`drvrIOError`	I/O error
	`$28`	`drvrNoDevice`	No device connected
	`$2E`	`drvrDiskSwitch`	The disk has been switched
	`$45`	`volNotFound`	Volume not found
	`$4A`	`badFileFormat`	Version error (incompatible file type)
	`$52`	`unknownVol`	Unknown volume type
	`$53`	`paramRangeError`	Parameter out of range
	`$57`	`dupVolume`	Duplicate volume name
	`$58`	`notBlockDev`	Not a block device

HFS FST

Introduction

This section provides information about the GS/OS HFS FST. The HFS disk format is described in detail in Chapter 19 of *Inside Macintosh, Volume IV*.

HFS is the file system currently used on the Apple Macintosh family of computers. The HFS FST gives GS/OS applications the ability to read and write data on HFS disks. Some of the advantages of the HFS file system are that it can access volumes larger than 32M and files larger than 16M, and it allows up to 65535 files and up to 65535 directories at the root level.

Limitations

- The first release of the HFS FST does not allow the Apple IIGS to boot from an HFS volume.
- The HFS FST does not support 5.25" disks. The attributes word in the FST Header has bit 1 set to indicate to the Initialization Manager that HFS is not a valid file system for 5.25" disks.
- The HFS FST does not support volumes created under MFS, the original file system used on the Macintosh.
- The HFS FST does not enable the Apple IIGS to execute applications written for the Macintosh or vice-versa.

Pathname Syntax

The HFS FST imposes the following restrictions on pathname syntax:

- Volume names are limited to a maximum of 27 characters.
- File names are limited to a maximum of 31 characters.
- All characters except ':' are valid in a pathname.

Note that the high bit of a character is significant. Characters with values greater than 127 are considered extended ASCII and typically display as special symbols.

If the first file name in a partial pathname is a number, GS/OS assumes that it is a prefix designator. Since numbers are valid file names in HFS, an explicit prefix designator should always be used when dealing with partial pathnames that begin with a number. For example, the pathname "0:555:Hello" refers to file "Hello" in folder "555" relative to prefix 0.

In some cases, GS/OS will recognize that a number cannot be a valid GS/OS prefix, and treat it as the start of a partial pathname. For example, 555 is too long to be a prefix in System Software 6.0. Full pathname, though, always insure the pathname will be treated correctly.

File Types and Auxiliary Types

The HFS FST handles the translation of ProDOS file types and auxiliary types to and from Macintosh creator types and file types in a manner similar to that used by the AppleShare FST, Apple File Exchange, and the MPW IIGS cross-development tools.

On an HFS volume, ProDOS files are distinguished by a Macintosh creator type of "pdos". The Macintosh file type is assigned based on the ProDOS file type and auxiliary type. If the ProDOS auxiliary type does not have to be preserved, the file can be given a special Macintosh file type, such as "TEXT" or "PS16", which allows the Macintosh to display a unique icon for the file. If

the ProDOS auxiliary type needs to be preserved, the file gets a Macintosh file type of the form $70 $uv $wx $yz ($70 is a lower-case ‘p’) in which $uv is the ProDOS file type and $wxyz is the ProDOS auxiliary type. The Macintosh will not be able to display unique icons for these files, but the advantage is that if the file is later copied onto a ProDOS volume, the auxiliary type information will be correct. ProDOS auxiliary type information is preserved for all SRC ($B0) and EXE ($B5) files and for S16 ($B3) files when the high byte of the auxiliary type is $DB (this signifies that the low byte of the auxiliary type contains valid information).

Macintosh directories do not have creator types or file types. When converted to ProDOS, they have ProDOS file type $0F (directory) and auxiliary type $0000.

The conversion rules are summarized in the following tables. If more than one rule applies, the one closest to the top of the table is used.

ProDOS to Macintosh conversion:

ProDOS File Type	Auxiliary Type	Macintosh Creator Type	File Type
$00	$0000	“pdos”	“BINA”
$04 (TXT)	$0000	“pdos”	“TEXT”
$FF (SYS)	(any)	“pdos”	“PSYS”
$B3 (S16)	$DByz	“pdos”	“p” $B3 $DB $yz
$B3 (S16)	(any)	“pdos”	“PS16”
$D7	$0000	“pdos”	“MIDI”
$D8	$0000	“pdos”	“AIFF”
$D8	$0001	“pdos”	“AIFC”
$E0	$0005	“dCpy”	“dImg”
$FF (SYS)	(any)	“pdos”	“PSYS”
$uv	$wxyz	“pdos”	“p” $uv $wx $yz

Macintosh to ProDOS conversion:

Macintosh Creator Type	File Type	ProDOS File Type	Auxiliary Type
“pdos”	“PSYS”	$FF (SYS)	$0000
“pdos”	“PS16”	$B3 (S16)	$0000
“pdos”	“XYΔΔ”†	$XY	$0000
“pdos”	“p” $uv $wx $yz	$uv	$wxyz
“dCpy”	“dImg”	$E0	$0005
(any)	“BINA”	$00	$0000
(any)	“TEXT”	$04 (TXT)	$0000
(any)	“MIDI”	$D7	$0000
(any)	“AIFF”	$D8	$0000
(any)	“AIFC”	$D8	$0001
(any)	(any)	$00	$0000

† Where X,Y are hexadecimal digits (i.e. ‘0’-‘9’ or ‘A’-‘F’), and Δ is a space.

System Calls

This section lists each of the supported system calls and indicates any areas where the operation of the HFS FST differs from that described in *Apple IIGS GS/OS Reference*.

`ChangePath` ($2004)

A file cannot be renamed if it is locked.

The `ChangePath` call allows a duplicate volume to be renamed if a device name is used as the source pathname.

`ClearBackupBit` ($200B)

Since the HFS file system does not have a "backup needed" bit for files or directories, this call changes the backup date/time field of the file or directory entry to be either the current date/time or one second later than the modified date/time field, whichever is later.

This call does nothing for volume directories.

`Close` ($2014)

This call truncates the file to its physical EOF before closing it.

`Create` ($2001)

The `access` parameter is handled as follows:

- Bit 0 (read enabled or disabled) is ignored. All files and directories are created with read access enabled. The HFS file system does not have the ability to disable read access.

- Bit 5 (backup needed or not needed) is ignored. All files and directories are created with the "backup date/time" field set to 0, which indicates that the file or directory has never been backed up. The HFS file system does not have a backup bit.

- Bit 2 (invisible or visible) is copied into bit 14 of the Macintosh Finder information flag word. (See `GetFileInfo`, later in this section, for a description of the Macintosh Finder information flag word.)

- Bit 1 (write enabled or disabled), bit 6 (rename enabled or disabled) and bit 7 (destroy enabled or disabled) are all mapped into the "locked" bit in the HFS file entry if a file is being created. If any one of these bits is set, the "locked" bit will be clear. That is, the locked bit is set only if the access is no-rename, no-destroy, no-write. The HFS file system does not store separate information for writing, renaming and deleting. A locked file cannot be written to, renamed, deleted, or have its EOF changed. It can have its mark and file info changed and it can be moved within its volume. Bit 1, bit 6 and bit 7 are ignored if a subdirectory is being created, since the HFS file system does not have the ability to lock directories.

The Macintosh creator type and file type are derived according to the rules in the previous section.

Only the low byte of the `fileType` parameter and the low word of the `auxType` parameter are used. If the high byte of the `fileType` parameter or high word of the `auxType` parameter is non-zero, an invalid parameter error is returned.

The `eof` and `resourceEOF` parameters are rounded up to the nearest allocation block when the file is extended.

If the `createDate` parameter contains a year which is earlier than 1904, it will be set to 1904. (The `createDate` parameter is only used in the ProDOS 16 form of the `Create` call.)

If the `storageType` parameter is $8005, the file must already exist and must be a "standard file" (that is, the file's resource fork must have a physical EOF of 0). For this special case, the `access`, `fileType`, `auxType` and `eof` parameters are ignored.

`EraseDisk` ($2025)

The only difference between `EraseDisk` and `Format` is that `EraseDisk` does not issue a format call to the device.

See comments under `Format`.

`Flush` ($2015)

A new GS/OS parameter has been added. If the `fastFlush` parameter is non-zero, the file's modified date/time information will not be updated unless the file entry has to be updated because other information (for example, the logical EOF or physical EOF) has changed.

This call does nothing for directories.

`Format` ($2024)

3.5" disks which have been formatted by the HFS FST will be recognized by both the Macintosh and Apple IIGS, but will not be bootable on either machine (blocks 0 and 1 are zeroed during formatting).

Unpartitioned hard drives which have been formatted by the HFS FST will be recognized by the Apple IIGS but will not be recognized by a Macintosh. In order for a hard drive to be recognized by a Macintosh it must be partitioned and must contain a special Apple_Driver partition which is used by the Mac when mounting the drive on the desktop. The HFS FST does not partition hard drives when formatting them. Therefore, if the user wishes to format a hard drive for use on both an Apple IIGS and a Macintosh, the drive must either be formatted on the Macintosh or partitioned and formatted using the Advanced Disk Utility application (ADU).

The minimum volume size is 10 physical blocks.

`FSTSpecific` ($2033)

This call always returns an invalid FST operation error ($65).

`GetDirEntry` ($201C)

This call is not defined for files, for which it will return error $4A (version error).

The `access`, `fileType`, `auxType`, and `optionList` parameters are as described for the `GetFileInfo` call.

The ProDOS `eof` and `blockCount` parameters are undefined if the `fileType` parameter is returned as $0F (directory).

The GS/OS `eof`, `blockCount`, `resourceEOF` and `resourceBlocks` parameters are undefined if the `fileType` parameter is returned as $0F (directory).

If the `name` output buffer is not big enough to hold the entire name string, `GetDirEntry` stores as much of the string as will fit and will then return all other requested output parameters before reporting error $4F (buffer too small).

`GetEOF` ($2019)

This call is not defined for directories, for which it will return error $4B (unsupported storage type).

`GetFileInfo` ($2006)

The `access` parameter bits are set as follows for files:

bit 0	1
bit 1	1 if the file is unlocked; otherwise 0
bit 2	Same value as the Macintosh Finder information “invisible” flag bit
bit 3	0
bit 4	0
bit 5	1 if the backup date/time field in the HFS file entry contains a date/time which is earlier than the modified date/time field; otherwise 0
bit 6	1 if the file is unlocked; otherwise 0
bit 7	1 if the file is unlocked; otherwise 0
bits 8-15	0

The `access` parameter bits are set as follows for subdirectories:

bit 0	1
bit 1	1
bit 2	Same value as the Macintosh Finder information “invisible” flag bit
bit 3	0
bit 4	0
bit 5	1 if the backup date/time field in the HFS file entry contains a date/time which is earlier than the modified date/time field; otherwise 0
bit 6	1
bit 7	1
bits 8-15	0

The `access` parameter bits are set as follows for volume directories:

bit 0	1
bit 1	1
bit 2	0
bit 3	0
bit 4	0
bit 5	0
bit 6	1
bit 7	1
bits 8-15	0

The `fileType` and `auxType` parameters are derived from the Macintosh creator type and file type according to the rules in “File Types and Auxiliary Types,” earlier in this chapter.

The only possible values for the `storageType` parameter are:

$01	Standard file
$05	Extended file
$0D	Subdirectory
$0F	Volume directory

If the physical EOF of a file's resource fork is 0, the `storageType` is returned as $01 (standard file), otherwise it will be returned as $05 (extended file). This is because the HFS file system has no way to distinguish between a resource fork of length 0 and a missing resource fork.

The GS/OS `optionList` is structured as follows:

$00	*bufferSize*	**Word**—Buffer size
$02	*dataSize*	**Word**—Data size
$04	*fileSysID*	**Word**—File system ID
$06	*finderInfo*	**32 bytes**—Macintosh Finder information in 68000 format
$26	*directoryID*	**Long**—Parent directory ID

The Macintosh Finder information for a file is shown below. Keep in mind that all fields are in the natural order for the 68000; that is, the most significant byte is first. This is the reverse of the byte order used almost universally on the Apple IIGS.

$00	*fileType*	**Long**—4 characters; the Macintosh fileType
$04	*creator*	**Long**—4 characters; the Macintosh creator type
$08	*finderFlags*	**Word**—Finder flags; see description following table
$0A	*iconLoc*	**Point**—Location of the file's icon in the window (local coordinates)
$0E	*fileWindow*	**Word**—File's window (unused with HFS)
$10	*iconID*	**Word**—Icon ID
$12	*unused*	**8 bytes**—Unused
$1A	*commentID*	**Word**—Comment ID
$1C	*directoryID*	**Long**—Directory ID of home folder

The bits in the finderFlags word are defined as follows:

Bit	Meaning
0	File is on the desktop
1	Bit 0 of color
2	Bit 1 of color
3	Bit 2 of color
4	Reserved for color
5	Always switch launch if possible
6	Application file is shareable
7	File should be cached
8	File initialized
9	Changed (obsolete)
10	Busy (obsolete)
11	NoCopy (obsolete)
12	Name is locked
13	File has bundle to export
14	File is invisible
15	Locked (obsolete)

The `directoryID` parameter is only valid if the file is on the desktop. It is used when the Put Away command returns the file to it's original directory.

The Macintosh Finder information for a subdirectory is structured as follows:

$00	*windRect*	**Rect**—Folder's window rectangle (in global coordinates)
$08	*finderFlags*	**Word**—Finder flags
$0A	— *iconLoc* —	**Point**—Location of the file's icon in the window (local coordinates)
$0E	*view*	**Word**—Folder's view
$10	— *position* —	**Point**—Window's scroll position
$14	— *nextID* —	**Long**—ID of next open directory
$18	*unused*	**Word**—Unused
$1A	*commentID*	**Word**—Comment ID
$1C	— *directoryID* —	**Long**—Directory ID of home folder

The Macintosh Finder information for a volume directory will be returned as all zeroes.

The ProDOS `blocksUsed` parameter is estimated for subdirectories, since there is no reasonable way to determine it with the HFS file system.

The GS/OS `eof`, `blocksUsed`, `resourceEOF` and `resourceBlocks` parameters are undefined for subdirectories and volume directories.

`GetMark` ($2017)

This call is not defined for directories, for which it returns error $4B (unsupported storage type).

`JudgeName` ($2007)

This is a GS/OS-style call only. The parameter block is as follows:

$00	*pCount*		**Word**—Input value (minimum is 3)
$02	*fileSysID*	**1**	**Word**—Input value; the call is dispatched only to this FST
$04	*nameType*	**2**	**Word**—Input value; type of name
$06	— *syntax* —	**3**	**Long**—Output value; points to a syntax rule description
$0A	*maxLen*	**4**	**Word**—Output value; maximum length of the specified name type
$0C	— *name* —	**5**	**Long**—Input value; pointer to a GS/OS output string
$10	*nameFlags*	**6**	**Word**—Output value; indicates what was wrong with the name

The `nameType` parameter tells the FST what sort of name is being tried. It should be one of the following (if not, error $53 (parameter out of range) will be reported):

Value	Meaning
0	Unknown
1	Volume name
2	Directory name
3	File name

If `nameType` is 0, the least restrictive rules will be used.

The `syntax` parameter points to a Pascal string the application can display. It tells the user what the correct syntax rules are for the FST.

The `name` parameter points to a GS/OS output string buffer. The FST modifies the name to make it conform to the FST's syntax rules. If the pointer is `NIL`, no error is reported. If the output buffer is not large enough to hold a maximum length name of the specified type, error $4F (buffer too small) is reported.

The `nameFlags` parameter tells what, if anything, was wrong with the name:

Bit	Use
15	Set if the name contained illegal characters
14	Set if the name was too long
13	Set for a syntax error

The nameFlags result is nonzero if and only if the FST modified the name pointed to by `name`.

If the `name` parameter is supplied but the name length is 0, the name string will be set to "A" (which is a legal name) and bit 13 of the `nameFlags` parameter will be set to indicate a syntax error.

The HFS FST modifies the name as follows:

- All colons and nulls are replaced with periods.
- If the name is too long and `nameType` is 1, the first 13 characters are kept, ".." is inserted, and the last 12 characters are kept. If the name is too long and `nameType` is not 1, the first 15 characters are kept, ".." is inserted, and keep the last 14 characters are kept.

`Open` ($2010)

If a directory is being opened, the `requestAccess` parameter must be either $0001 (read only) or $0000. If the `requestAccess` parameter is $0002 (write only) or $0003 (read/write), error $4E (invalid access) will be reported.

If a file is being opened, the following rules apply:

- If the `requestAccess` parameter is $0001 (read only), the file will be opened as read only unless it has already been opened with write access, in which case error $4E (invalid access) will be returned.
- If the `requestAccess` parameter is $0002 (write only), the file will be opened as write only unless it has already been opened or is locked, in which case error $4E (invalid access) will be returned.
- If the `requestAccess` parameter is $0003 (read/write), the file will be opened as read/write unless it has already been opened or is locked, in which case error $4E (invalid access) will be returned.
- If the `requestAccess` parameter is $0000 (as permitted), an attempt will be made to open the file as read/write if it is unlocked or read only if it is locked. If the file has already been opened with write access, error $4E (invalid access) will be returned. If the file has already been opened with read only access, it will be opened again with read only access.

The GS/OS `access`, `fileType`, `auxType`, `storageType`, `optionList`, `eof`, `blockUsed`, `resourceEOF` and `resourceBlocks` parameters are as described for the `GetFileInfo` call.

A file's resource fork can always be opened, since there is no way to distinguish between a fork of length 0 and a fork that doesn't exist.

`Read` ($2012)

This call is not defined for directories, for which it returns error $4B (unsupported storage type).

`SetEOF` ($2018)

This call is not defined for directories, for which it returns error $4B (unsupported storage type).

If the logical EOF is extended, the additional bytes will not be zeroed.

If the logical EOF is extended, the physical EOF will be set to the new logical EOF rounded up to the nearest allocation block. If the file needs to be extended to accommodate the new physical EOF, the number of bytes added to the file will be rounded up to the nearest clump.

If the logical EOF is shortened, the physical EOF is truncated to the logical EOF rounded up to the nearest allocation block and the file is truncated to the end of the extent containing the new physical EOF.

`SetFileInfo` ($2005)

The `access` parameter is handled as follows:

- Bit 0 (read enabled or disabled) is ignored. The HFS file system does not have the ability to disable read access.

- Bit 5 (backup needed or not needed) is ignored.

- Bit 2 (invisible or visible) is copied into bit 14 of the Macintosh Finder information flag word. (See `GetFileInfo` for a description of the Macintosh Finder information flag word.)

- Bit 1 (write enabled or disabled), bit 6 (rename enabled or disabled) and bit 7 (destroy enabled or disabled) are handled differently for files and directories.

  For files, bits 1, 6 and 7 are all mapped into the "locked" bit in the HFS file entry. If any one of the bits is set, the "locked" bit will be clear. The HFS file system does not store separate information for writing, renaming and deleting. A locked file cannot be written to, renamed, deleted, or have its EOF changed. It can have its mark and file info changed and it can be moved within its volume.

  For directories, if bits 1, 6 or 7 are clear a $4E error will be returned, since the HFS file system does not have the ability to lock directories.

The Macintosh creator type and file type are derived according to the rules in "File Types and Auxiliary Types," earlier in this chapter.

Only the low byte of the `fileType` parameter and the low word of the `auxType` parameter are used. If the high byte of the `fileType` parameter or high word of the `auxType` parameter is non-zero, an invalid parameter error is returned.

The `fileType` and `auxType` parameters are ignored for subdirectories.

The GS/OS `optionList` is structured as follows:

$00	*bufferSize*	**Word**—Buffer size
$02	*dataSize*	**Word**—Data size
$04	*fileSysID*	**Word**—File system ID
$06	*finderInfo*	**32 bytes**—Macintosh Finder information in 68000 format

If the `fileSysID` field is not $0006 (HFS), $000D (AppleShare) or $0001 (ProDOS), the option list is ignored. If the `bufferSize` field is less than 36 or if the `dataSize` field is less than 32, the option list is ignored. The Macintosh creator type and file type supplied in the `finderInfo` field have priority over the creator type and file type derived from the `fileType` and `auxType` input parameters. Any data past the `finderInfo` field is ignored. See `GetFileInfo` for a description of the `finderInfo` field.

Any year inputs which are earlier than 1904 will be set to 1904.

This call is not defined for volume directories; it will return error $40 (bad path syntax).

`SetMark` ($2016)

This call is not defined for directories, for which it will return error $4B (unsupported storage type).

`Volume` ($2008)

The `Volume` call will set up the `volname` output parameter even if error $57 (duplicate volume) is being returned.

If a ProDOS 16 `Volume` call is being executed and the `volname` output parameter is longer than 16 characters (15 character name plus a leading slash), error $2F (device off line) is returned.

`Write` ($2013)

This call is not defined for directories, for which it will return error $4B (unsupported storage type).

If the logical EOF needs to be extended to accommodate the new data, the physical EOF will be set to the new logical EOF rounded up to the nearest allocation block. If the file needs to be extended to accommodate the new physical EOF, the number of bytes added to the file will be rounded up to the nearest clump.

Pascal FST

Introduction

The Pascal File System Translator lets GS/OS read Apple II Pascal disks.

The Pascal FST is a read only implementation; it is provided for use as a migration tool to allow users of the Apple II Pascal Filing System to transport their files into the GS/OS environment. Any calls made which attempt to create or alter data on a Pascal volume will usually return with error $2B, disk write protected.

Compatibility

The Pascal FST does not internally support any type of volume partitioning. This is because partitions are handled on the device driver level within GS/OS, and the types of “partitioning” used in the Apple II Pascal Filing System (namely the Pascal Profile Manager and the partitioning scheme described in Apple II ProDOS Technical Note #25, Non-Standard Storage Types) requires the driver to have a knowledge of the file systems on the device. To access the files on any but the first partition of a volume, you must use the Pascal Filing System to copy the files to the first volume in the set, or to another volume entirely. If your partition is setup using the Pascal Profile Manager, you will have use the Pascal Filing System to copy the files to another Pascal volume.

File Types

The file type returned for any given file is determined by whether the Pascal FST is in ASCII Text Mode or Pascal Native Mode. The Pascal FST can be toggled between these two modes by using the `FSTSpecific` call. The following table shows the Pascal file types and their translations under GS/OS by the Pascal FST in each mode:

Pascal Volume	ASCII Mode	Native Mode
$00 untyped	$00 unknown	$00 unknown
$01 xdsk file	$00 unknown	$00 unknown
$02 code file	$02 pascal code file	$02 pascal code file
$03 text file	$04 text file	$03 pascal text file
$04 info file	$00 unknown	$00 unknown
$05 data file	$05 pascal data file	$05 pascal data file
$06 graf file	$00 unknown	$00 unknown
$07 foto file	$00 unknown	$00 unknown

When in ASCII Mode, the Pascal FST strips off the leading 1K header used by the text editor in the Pascal Filing System on text files, expands all DLE blank compressions, and skips all NULL characters, thus returning only ASCII text.

When in Native Mode, the Pascal FST returns Pascal file types $02, $03, and $05 as GS/OS file types $02, $03, and $05, respectively.

Note By default, the Pascal FST is in ASCII Text Mode.

System Calls

This section describes the system calls supported by the Pascal FST.

Unless otherwise noted, all calls may return with one of the following errors:

`$04`	`invalidPcount`	The parameter count is out of range
`$10`	`devNotFound`	Device not found
`$11`	`invalidDevNum`	Invalid device number
`$27`	`drvrIOError`	I/O error
`$2B`	`drvrWrtProt`	Device is write protected
`$2E`	`drvrDiskSwitch`	The disk has been switched
`$40`	`badPathSyntax`	Invalid pathname syntax
`$43`	`invalidRefNum`	Invalid reference number
`$44`	`pathNotFound`	Subdirectory does not exist
`$45`	`volNotFound`	Volume not found
`$46`	`fileNotFound`	File not found
`$4A`	`badFileFormat`	Version error (incompatible file type)
`$4C`	`eofEncountered`	End of file encountered
`$4D`	`outOfRange`	Position out of range
`$4E`	`invalidAccess`	Access not allowed
`$4F`	`buffTooSmall`	Buffer too small
`$50`	`fileBusy`	File is already open
`$51`	`dirError`	Directory error
`$53`	`paramRangeError`	Parameter out of range
`$54`	`outOfMem`	Out of memory
`$57`	`dupVolume`	Duplicate volume name
`$58`	`notBlockDev`	Not a block device
`$5A`	`damagedBitMap`	Block number too large
`$5B`	`badPathNames`	Invalid pathnames for `ChangePath`
`$61`	`endOfDir`	End of directory has been reached
`$62`	`invalidClass`	Invalid FST call class
`$63`	`resForkNotFound`	The file does not contain a required resource

$65	`invalidFSTop`	FST does not handle this type of call
$66	`fstCaution`	FST handled call, but result is weird

`ChangePath` ($2004)

This call is not supported, since the Pascal FST is read-only. It always returns error $2B, disk write protected.

`ClearBackupBit` ($200B)

This call is not supported, since the Pascal FST is read-only. It always returns error $2B, disk write protected.

`Create` ($2001)

This call is not supported, since the Pascal FST is read-only. It always returns error $2B, disk write protected.

`Destroy` ($2002)

This call is not supported, since the Pascal FST is read-only. It always returns error $2B, disk write protected.

`Erase` ($2025)

This call is not supported, since the Pascal FST is read-only. It always returns error $2B, disk write protected.

`Flush` ($2015)

This call normally writes out all buffered information which has not been written to the disk, however this call performs no function in the Pascal FST and will always return with $00, call successful.

`Format` ($2024)

This call is not supported, since the Pascal FST is read-only. It always returns error $65, invalid FST operation.

`FSTSpecific` ($2033)

The Pascal FST supports two FST specific calls; they are used to manipulate the text mode. See "File Types," earlier in the description of this FST, for information about the FST mode.

`get_text_mode` (Pascal `FSTSpecific` Subcall)

This call reads the text mode.

$00	*pCount*		**Word**—Input value; must be 3
$02	*fileSysID*	**1**	**Word**—Input value; $0004 (Apple II Pascal FST)
$04	*commandNum*	**2**	**Word**—Input vale; $0001
$06	*fstMode*	**3**	**Word**—FST mode

`fstMode` is $0000 for ASCII Text mode and $0001 for Pascal Native mode.

`set_text_mode` (Pascal `FSTSpecific` Subcall)

This call sets the FST to either ASCII Text mode or Pascal Native mode.

$00	*pCount*		**Word**—Input value; must be 3
$02	*fileSysID*	**1**	**Word**—Input value; $0004 (Apple II Pascal FST)
$04	*commandNum*	**2**	**Word**—Input vale; $8001
$06	*fstMode*	**3**	**Word**—FST mode

`fstMode` is $0000 for ASCII Text mode and $0001 for Pascal Native mode.

`Open` ($2010)

When a text file is opened, the data read from the file will continue to be in the mode in which the file was opened (ASCII or Native). For example, if you open a text file while in Pascal Native mode, then switch to ASCII Text mode, any data read from the file will be read as if the FST was in Pascal Native mode.

`Read` ($2012)

This call is not permitted on directories; attempting to read a directory will result in an invalid access error ($4E).

`SetEOF` ($2018)

This call is not supported, since the Pascal FST is read-only. It always returns error $2B, disk write protected.

`SetFileInfo` ($2005)

This call is not supported, since the Pascal FST is read-only. It always returns error $2B, disk write protected.

`SetMark` ($2016)

This call is not permitted on directories; attempting to set the mark in a directory will result in an invalid access error ($4E).

`Write` ($2013)

This call is not supported, since Pascal FST is a read-only FST. It will always return with a disk write protected error ($2B).

Error Codes

The following error codes have either been added for this release, or were not previously documented:

Code	Constant	Description
$42	`tooManyFilesOpen`	The AppleShare file server limit of open files has been reached.
$65	`invalidFSTop`	FST does not handle this type of call.
$66	`fstCaution`	FST handled call, but result is weird.
$68	`devListFull`	Device list is full.
$69	`supListFull`	Supervisor list is full.
$6A	`fstError`	Generic FST error.
$88	`networkError`	Generic network error.

The generated driver bank $00 core routines have been modified so that when a Pascal firmware driver reports an error, the error will be translated into a GS/OS error code in the range of $30 through $3F.

For example, assume you have a Super Serial Card in slot 2, and a Pascal `Read` call generates an overrun error ($24). The card returns the error code in the `X` register and the valid character it read in the `A` register. In System Software 5.0.4 and earlier, the GS/OS generated driver would recognize the error code in the `X` register, discard the valid character in `A` and return error $24 to the caller. However, GS/OS error $24 is “driver already open” and has nothing to do with what really happened.

In System Software 6.0 and later, GS/OS’s generated driver routines recognize the error, store the valid character in the caller’s buffer, and translate error $24 to error $34 before returning it to the caller. All $20–$2F errors are translated to $30–$3F errors to prevent conflict with GS/OS error numbers. The following table shows which conditions correspond to which errors:

Code	Carrier Lost	Overrun	Framing Error	Parity Error
$30				
$31				•
$32			•	
$33			•	•
$34		•		
$35		•		•
$36		•	•	
$37		•	•	•
$38	•			
$39	•			•
$3A	•		•	
$3B	•		•	•
$3C	•	•		
$3D	•	•		•
$3E	•	•	•	
$3F	•	•	•	•

ProDOS 8

- An alternate `Quit` call handler has been added to ProDOS 8 for use by GS/OS when launching ProDOS 8 applications from the GS/OS environment. See the section on BASIC.Launcher functionality for a complete description.

- ProDOS 8 now turns off the Super Hi-Res screen just before displaying its splash screen, but only if it's running on an Apple IIGS.

- To gain a little more room for new enhancements and bug fixes, the code has been changed to use 65C02 opcodes when available. As part of the initialization sequence, ProDOS 8 checks the processor type and will refuse (gracefully) to execute if a 6502 is found. (The actual error message is "Relocation/Configuration Error".)

- The start-up code and the kernel have been changed to support more than two devices per slot when the devices are attached to a SmartPort interface.

  During start up, ProDOS 8 performs the normal device scan like it always has. Once this is complete, it performs a second device scan, searching for SmartPort interfaces that have more than two devices connected. When such a device is found, the start-up code stores some necessary information into some internal tables, and then fills in the Device Table with the address of a custom driver residing within the kernel.

  When the custom driver receives control as the result of a ProDOS 8 device call, it translates the ProDOS device call into a SmartPort call, using the device information that had previously been stored into the internal tables.

- The slot clock driver has been updated to work for years 1991-1996.

ChangePath $2004

`ChangePath` changes a file's pathname to another pathname on the same volume, or changes the name of a volume. `ChangePath` cannot be used to change a device name; use the `DRename` call for that purpose.

Note This is not a new call. See the *Apple IIGS GS/OS Reference* for a complete description of this call.

Parameters

$00	*pCount*		**Word**—Input value (minimum 2)
$02	— *pathname* —	1	**Long**—Input
$06	— *newPathname* —	2	**Long**—Input
$0A	*flags*	3	**Word**—Input

Errors See *Apple IIGS GS/OS Reference*.

`pCount` Word input value. `pCount` is the number of parameters in this parameter block. The minimum is 2; the maximum is 3.

`pathname` Long word input pointer. Points to a GS/OS string representing the name of the file or volume whose pathname is to be changed.

`newPathname` Long word input pointer. Points to a GS/OS string representing the new pathname of the file or volume whose name is to be changed.

`flags` Word input value. Bits defining optional actions of the call, as follows:

Bit 15 = 1 When renaming a volume and specifying the location of the volume by using a device name as the first parameter, setting this bit of the flags word tells GS/OS **not** to update any internal data structures with the new volume name. Use this when renaming a known duplicate on-line volume and the new name should **not** be used to refer to the original on-line volume. (For example, renaming a duplicate of the boot volume.) In addition to setting this bit, the first pathname **must** be supplied as a device name.

Bit 15 = 0 Update internal data structures normally.

Bits 14-0 Reserved; set to 0.

Comments

- A file may not be renamed while it is open.
- A file may not be renamed if rename access is disabled for the file.

- A subdirectory *s* may not be moved into another subdirectory *t* if *s* = *t* or if *t* is contained in the directory hierarchy starting at *s*. For example, “rename /v to /v/w” is illegal, as is “rename /v/w to /v/w/x”.

- The `ChangePath` call has been changed for System 6.0 to allow a duplicate on-line volume to be renamed. Previously, this was not possible because GS/OS would not allow both volumes to be on line long enough for even one of them to be renamed.

  To perform the `ChangePath`, the pathname parameter (parameter number 1) must be supplied as a device name corresponding to the device in which the volume is mounted.

EraseDisk $2025
Format $2024

This section describes changes to two existing calls. See *Apple IIGS GS/OS Reference* for a full description of these calls.

Note These are not new calls. See the *Apple IIGS GS/OS Reference* for a complete description of these calls.

Parameters

$00	*pCount*		**Word**—Input value (minimum 1)
$02	— *devName* —	**1**	**Long**—Input
$06	— *volName* —	**2**	**Long**—Input
$0A	*fileSysID*	**3**	**Word**—Output
$0C	*reqFileSysID*	**4**	**Word**—Input
$0E	*flags*	**5**	**Word**—Input
$10	— *realColName* —	**6**	**Long**—Input

Errors See *Apple IIGS GS/OS Reference*.

`pCount` Word input value. `pCount` is the number of parameters in this parameter block. The minimum is 1; the maximum is 6.

`devName` Long word input pointer. Points to a GS/OS string representing the device name of the device containing the volume to be erased.

`volName` Long word input value. Points to a GS/OS string representing the volume name to be assigned to the newly erased or initialized volume. If the pointer is not `NIL`, and the length of the supplied string is not 0, and bit 15 of the flags word is set, GS/OS uses the value of the string as the default value in the LineEdit item of the Initialization dialog box. Otherwise, the supplied volume name is displayed as a non-editable name.

`fileSysID` Word result value. If the call is successful, this parameter identifies the file system that was placed on the disk. If the call is unsuccessful, this result is $0000.

The file system IDs are as follows:

$0000	Reserved	$0008	Apple CP/M
$0001	ProDOS/SOS	$0009	Reserved
$0002	DOS 3.3	$000A	MS/DOS
$0003	DOS 3.2 or 3.1	$000B	High Sierra
$0004	Apple II Pascal	$000C	ISO 9660
$0005	Macintosh (MFS)	$000D	AppleShare
$0006	Macintosh (HFS)	$000E-$FFFF	Reserved
$0007	Lisa		

`reqFileSysID` Word input value. Provides the file system ID of the file system that should be used to format the disk. The values for this parameter are the same as those for the `fileSysID` parameter.

If you supply both this parameter and a valid `volName` parameter, but no flags word, the Disk Initialization dialog box is suppressed. By supplying both this parameter and setting bit 14 of the flags word, the Initialization Manager uses the `reqFileSysID` to set the initial selection in the FST List control.

`flags` Word input value. These bits tell the Initialization Manager how to interpret the `volName` and `reqFileSysID` parameters, as follows:

Bit 15 If this bit is set, `volName` is placed into a LineEdit item. The user is able to change the value. If the bit is clear, the volume name is shown as a Static Text item.

If there is no volume name, this bit is ignored and the user is presented with the volume name "Untitled" in a LineEdit item.

Bit 14 If this bit is set, `reqFileSysID` is treated as the initial selection in the FST list control, and the user is allowed to make another selection. If the bit is clear, `reqFileSysID` is shown as the only selectable file system. All other file systems in the FST list control will be dimmed.

Bit 13 If this bit is set, the "Initializing" dialog is not displayed while the actual format operation is taking place. If the bit is clear, the "Initializing" dialog is shown while the system is busy.

Bits 12-0 Reserved; set to 0.

`realVolName` Long word input value. Pointer to a GS/OS result buffer into which GS/OS places the actual volume name used for the operation. If the user is allowed to edit the supplied volume name, this may be different from the volume name passed as parameter number 2.

GS/OS limits the length of a user-entered volume name to 32 characters. The result buffer should be at least this large. If it's not long enough, a `buffTooSmall` error will be returned, but not until **after** the `Format` or `EraseDisk` operation completes. In this case, it would be extremely unfriendly to simply make the buffer longer and repeat the call (as is usually done). Instead, use the `Volume` call to retrieve the volume's new name.

Comments

The `realVolName` result buffer is not filled in until **after** GS/OS is done using the `volName` parameter. Therefore, it is possible to fill a result buffer with a name, and set `volName` to point to the GS/OS string portion of the buffer, and then set `realVolName` to point to the beginning of the same buffer. In this way, the same memory can be used for both the input and output parameters.

GetLevel $201B

`GetLevel` returns the current value of the system or user file level.

Note This is not a new call. See the *Apple IIGS GS/OS Reference* for a complete description of this call.

Parameters

$00	*pCount*		**Word**—Input value (minimum 1)
$02	*level*	**1**	**Word**—Output
$04	*levelMode*	**2**	**Word**—Input

Errors $59 `invalidLevel` invalid file level

`pCount` Word input value. `pCount` is the number of parameters in this parameter block. The minimum is 1; the maximum is 2.

`level` Word result value. The value of the system file level.

`levelMode` Word input value. Internal range of the file level.

Comments

The `levelMode` parameter is useful when a file needs to be opened that can't be closed with a `Close` call with a reference number of 0.

The steps to open a file at an internal file level are:

1. Call `GetLevel` with `pCount` = 2, `levelMode` = $0000. Save the returned level.
2. Call `SetLevel` with `pCount` = 2, `level` = $0080 and `levelMode` = $0000.
3. Open a file or files with a ProDOS 16 or GS/OS `Open` call.
4. Call `SetLevel` with `pCount` = 2, `levelMode` = $0000, and `level` = saved level.

To close your protected file, simply do a `Close` with the reference number. There is no need to fiddle with the file level when closing by reference number.

GetName $2027

`GetName` returns the file name (not the complete pathname) of the currently running application. It can also return the User ID of the currently running application.

There are two ways to get the complete pathname of the application:

1. Concatenate prefix “9:” with the file name returned by this call. Do this before making any change in prefix “9:”.

2. Pass the returned User ID to the System Loader in an `LGetPathname2` call.

Note This is not a new call. See the *Apple IIGS GS/OS Reference* for a complete description of this call.

Parameters

$00	*pCount*		**Word**—Input value (minimum 1)
$02	— *dataBuffer* —	**1**	**Long**—Input
$06	*userID*	**2**	**Word**—Output

Errors See *Apple IIGS GS/OS Reference*.

`pCount` Word input value. `pCount` is the number of parameters in this parameter block. The minimum is 1; the maximum is 2.

`dataBuffer` Long word input pointer. Points to a result buffer where the file name is returned.

`userID` Word result value. User ID of the currently executing application.

GetRefInfo $2039

`GetRefInfo` returns the access attributes and full pathname for an open file when the reference number is given as input. Optionally, the fork number and file level used to open the file may be returned.

Note This is not a new call. See the *Apple IIGS GS/OS Reference* for a complete description of this call.

Parameters

$00	*pCount*		**Word**—Input value (minimum 1)
$02	*refNum*	**1**	**Word**—Input
$04	*access*	**2**	**Word**—Input
$06	— *pathname* —	**3**	**Long**—Output
$0A	*resourceNumber*	**4**	**Word**—Output
$0C	*level*	**5**	**Word**—Output

Errors See *Apple IIGS GS/OS Reference*.

`pCount` Word input value. `pCount` is the number of parameters in this parameter block. The minimum is 2; the maximum is 5.

`refNum` Word input value. Reference number of the open file.

`access` Word output value. Access attributes of the open file, as follows:

1 = read only
2 = write only
3 = read/write

`pathname` Long word input pointer. Points to a GS/OS output string where GS/OS places the full pathname of the file selected by the `refNum` parameter.

`resourceNumber` Word output value. Defines which fork of the file is opened with the supplied reference number. A value of $0000 means the data fork is open, a value of $0001 means the resource fork is open.

`level` Word output value. The system file level in effect when the file was opened. This value is copied directly out of the file's internal record. Non-protected levels (see `GetLevel` and `SetLevel` in this chapter) have bit 15 set.

JudgeName $2007

`JudgeName` verifies the syntax of a file name, directory name or volume name. The caller can also query the file system about the syntax limitations for a given type of name.

Note Read-only FSTs return an `invalidFSTop` error ($65).

Parameters

$00	*pCount*		**Word**—Input value (minimum 3)
$02	*fileSysID*	1	**Word**—Input
$04	*nameType*	2	**Word**—Input
$06	— *syntax* —	3	**Long**—Output
$0A	*maxLen*	4	**Word**—Output
$0C	— *name* —	5	**Long**—Input
$10	*nameFlags*	6	**Word**—Output

Errors

$4F	`buffTooSmall`	Buffer too small.
$53	`paramRangeError`	Parameter out of range.
$64	`invalidFSTID`	Invalid FST number.
$65	`invalidFSTop`	FST does not handle this type of call.

`pCount` Word input value. `pCount` is the number of parameters in this parameter block. The minimum is 3; the maximum is 6.

`fileSysID` Word input value. File system ID of the FST to which the call is directed.

`nameType` Word input value. Specifies the type of name.

0 = unknown
1 = volume name
2 = directory name
3 = file name

If `nameType` is 0, the least restrictive rules are used. If `nameType` is greater than 3, error $53 (parameter out of range) is reported.

`syntax` Long word result pointer. Points to a displayable Pascal string which describes the FSTs syntax rules.

`maxLen` Word result value. The maximum length of the specified name type.

`name` Long word input pointer. Points to a GS/OS output buffer which contains the name. The FST changes the name to make it conform to the FSTs syntax rules.

If the pointer is `NIL`, no error is reported. If the output buffer is not large enough to hold a maximum length name of the specified type, error $4F (buffer too small) is reported. (This is because the changed name returned by the FST may be longer than that supplied by the caller.)

The supplied name should be a single name only with no separator characters. The `JudgeName` call is not intended to pass judgement on a pathname.

`nameFlags` Word result value. Indicates what, if anything, was wrong with the name.

Bit 15	If this bit is set, the name contained invalid characters.
Bit 14	If this bit is set, the name was too long.
Bit 13	If this bit was set, some general syntax rule was violated.
Bits 12-0	Reserved.

ProDOS FST Note

Before it does any other processing, a ProDOS `JudgeName` call translates special characters into plain ASCII characters by calling the new Miscellaneous Tool Set routine `StringToText`. When making this call, the FST forces the target language to English.

Here are the `StringToText` translations that are important for ProDOS. Other translations occur, but only the ones listed below result in characters that the ProDOS FST keeps. (Other characters get turned into periods, and then groups of periods are shortened.)

All letters with diacritical marks (accent, grave accent, umlaut, tilde, degree symbol) become the corresponding normal letters.

Character	Becomes
$CA (option-space)	normal space ($20)
“®”	“(R)”
“©”	“(C)”
“™”	“(TM)”
“µ”	“u”
“ƒ”	“f”
“ß”	“B”
“Œ” and “œ”	“OE” and “oe”
“…” (option-;)	three periods (...)
“ç”	“c”
“ø” and “Ø”	“0” (zero).
“Æ” and “æ”	“AE” and “ae”

SetLevel $201A

`SetLevel` sets the current value of the system file level.

Whenever a file is opened, GS/OS assigns a file level equal to the current system file level. A `Close` call with a reference number of $0000 closes all files with the file level values at or above the current system file level. Similarly, a `Flush` call with a reference number of $0000 flushes all files with file level values at or above the current system file level.

See the description of `GetLevel` in this chapter.

Note This is not a new call. See the *Apple IIGS GS/OS Reference* for a complete description of this call.

Parameters

$00	*pCount*		**Word**—Input value (minimum 1)
$02	*level*	**1**	**Word**—Input
$04	*levelMode*	**2**	**Word**—Input

Errors $59 `invalidLevel` Invalid file level.

`pCount` Word input value. `pCount` is the number of parameters in this parameter block. The minimum is 1; the maximum is 2.

`level` Word input value. The value of the system file level.

`levelMode` Word input value. Internal range of the file level.

SetStdRefNum $203A

`SetStdRefNum` allows the caller to explicitly set the reference number associated with one of the standard I/O channels (standard input, standard output, standard error output). It provides the inverse functionality of the `GetStdRefNum` call.

Parameters

$00	*pCount*		**Word**—Input value (minimum 1)
$02	*prefixNum*	**1**	**Word**—Input
$04	*refNum*	**2**	**Word**—Input

Errors	$43	`invalidRefNum`	Invalid reference number.
	$53	`paramRangeError`	Parameter out of range.

`pCount` Word input value. Number of parameters in this parameter block. This parameter must be set to 2.

`prefixNum` Word input value. Decimal value of the prefix number associated with the standard I/O channel. Valid prefix numbers are 10 (standard input), 11 (standard output), and 12 (standard error output).

`refNum` Word input value. Reference number to use for the standard I/O channel. The reference number must refer to a currently open file.

Volume $2008

Given the name of a block device, `Volume` returns the name of the volume mounted in the device, along with other information about the device, volume, and FST which manages the file system on the volume.

Note This is not a new call. See the *Apple IIGS GS/OS Reference* for a complete description of this call.

Parameters

$00	*pCount*		**Word**—Input value (minimum 2)
$02	— *devName* —	**1**	**Long**—Input
$06	— *volName* —	**2**	**Long**—Input
$0A	— *totalBlocks* —	**3**	**Long**—Output
$0E	— *freeBlocks* —	**4**	**Long**—Output
$12	*fileSysID*	**5**	**Word**—Output
$14	*blockSize*	**6**	**Word**—Output
$16	*characteristics*	**7**	**Word**—Output
$18	*deviceID*	**8**	**Word**—Output

Errors See *Apple IIGS GS/OS Reference*.

`pCount` Word input value. `pCount` is the number of parameters in this parameter block. The minimum is 2; the maximum is 8.

`devName` Long word input pointer. Points to a GS/OS input string containing the name of a block device.

`volName` Long word input pointer. Points to a GS/OS output string where GS/OS returns the volume name of the volume mounted in the device.

`totalBlocks` Long word result value. Total number of blocks contained on the volume.

`freeBlocks` Long word result value. The number of free (unallocated) blocks on the volume.

`fileSysID` Word result value. Identifies the file system contained on the volume, as follows:

$0000	Reserved	$0008	Apple CP/M
$0001	ProDOS/SOS	$0009	Reserved
$0002	DOS 3.3	$000A	MS/DOS
$0003	DOS 3.2 or 3.1	$000B	High Sierra
$0004	Apple II Pascal	$000C	ISO 9660
$0005	Macintosh (MFS)	$000D	AppleShare
$0006	Macintosh (HFS)	$000E-$FFFF	Reserved
$0007	Lisa		

`blockSize` Word result value. The size, in bytes, of a block.

`characteristics` Word result value. Device and FST characteristics, as follows:

Bit 15	Set if the device is a RAM or ROM disk.
Bit 14	Set if the device is a generated device.
Bit 13	Set if the device is a linked device.
Bit 12	Set if the device is busy.
Bit 11	Set if the device is restartable.
Bit 10	Set if the device is not renameable.
Bits 9-8	These bits indicate the maximum CPU speed at which the device can function. For 1MHz operation, these bits will be 00; for normal Apple IIGS speed, these bits are 01; for devices that support accelerator cards, these bits are 10; and for devices that are not speed dependent, these bits are 11.
Bit 7	Set if the device is a block device.
Bit 6	Set if writing to the device is allowed.
Bit 5	Set if reading from the device is allowed.
Bit 4	Set if the volume mounted on the device contains files that are open.
Bit 3	Set if formatting the device is allowed.
Bit 2	Set if the device contains removeable media.
Bit 1	Set if the media in the device is write protected.
Bit 0	Set if the volume mounted on the device is formatted using a read-only FST.

`deviceID` Word result value. This is an identifying number associated with a particular type of device. This parameter may be useful for Finder-like applications when determining what type of icon to display for a particular device.

Current definitions of the device ID numbers include:

$0000	Apple 5.25 Drive (Includes UniDisk™, DuoDisk®, Disk IIc, and Disk II®)
$0001	ProFile™ 5 MB
$0002	ProFile 10 MB
$0003	Apple 3.5 Drive (includes UniDisk 3.5 Drive)
$0004	SCSI (generic)
$0005	SCSI hard disk
$0006	SCSI tape drive
$0007	SCSI CD-ROM
$0008	SCSI printer
$0009	Serial modem
$000A	Console driver
$000B	Serial printer
$000C	Serial LaserWriter®
$000D	AppleTalk® LaserWriter
$000E	RAM disk
$000F	ROM disk
$0010	File Server
$0011	Reserved.
$0012	Apple Desktop Bus®
$0013	Hard disk (generic)
$0014	Floppy disk (generic)
$0015	Tape drive (generic)
$0016	Character device driver (generic)
$0017	MFM-encoded disk drive

$0018	AppleTalk network (generic)
$0019	Sequential Access device
$001A	SCSI Scanner
$001B	Other scanner
$001C	LaserWriter SC
$001D	AppleTalk main driver
$001E	AppleShare file server driver
$001F	AppleTalk RPM driver
$0020	Apple SCSI Tape Driver

Comments

When a `dupVolume` error is returned, the pathname in conflict is returned in the buffer pointed to by the `volName` parameter.

The characteristics word is very similar to the same parameter to the `DInfo` call, with the addition of the “Media Write Protected”, “Open Files” and “Read-only File System” bit definitions. They are provided to return as much information as possible in a single GS/OS call about the mounted volume and the device in which it is mounted.

Chapter 34 **Apple IIGS Finder 6.0**

This chapter contains information about Finder 6.0. It discusses how to use the new Finder features, as well as how programmers can create programs that interact with the Finder.

What's New in Finder 6.0?

Better GS/OS Support

All File System Translators (FSTs) supported in System 6.0 are supported by Finder 6.0. These include: ProDOS, HFS, DOS 3.3, Pascal, AppleShare, and High Sierra/ISO 9660.

Finder 6.0 uses GS/OS `optionLists` to preserve as many file attributes as possible when copying files across file systems. For example, if you copy an HFS file to a ProDOS disk, its 4-byte Macintosh file type and creator type remain intact.

Resources

Resources are an integral part of Finder 6.0. All of the major components of the Finder are contained in resources. This includes all of the Finder's menus, its About Box, and its Tool Start-up table. All of the Finder's dialogs are contained in resources.

An `rRectList` resource (resource type $C001, ID = 1) now contains information that a user might wish to customize:

```
resource rRectList(1,locked,fixed,preload) {
  {
        { 39, 14,103,358},          /* first default window position                  */
        { 49, 24,113,368},          /* second default window position                 */
        { 59, 34,123,378},          /* third default window position                  */
        { 69, 44,133,388},          /* fourth default window position                 */
        { 79, 54,143,398},          /* fifth default window position                  */
        { 89, 64,153,408},          /* sixth default window position                  */
        { 99, 74,163,418},          /* seventh default window position                */
        {109, 84,173,428},          /* eighth default window position                 */

        {$e000,$002c,$FFFF,  0},    /* Y1 = default folder background & outline color */
                                    /* X1 = comma character                           */
                                    /* Y2 = default preference settings               */
                                    /* X2 = reserved for default quit setting         */

        {180,570,  0,  0},          /* default trashcan position (y1,x1 used only)    */
        {120, 20,180,500},          /* default trash window position                  */
        {120, 20,180,500},          /* default clipboard window position              */

        {$dddd,$dddd,$dddd,$dddd},  /* default desktop pattern                        */
        {$dddd,$dddd,$dddd,$dddd}
  }
};
```

`rComment(1)` and `rComment(2)` resources contained in any files the Finder deals with have special purposes. These resource contain unformatted text. The Finder recognizes two kinds of `rComment` resources (the only two that are currently defined).

An `rComment` resource with an ID of 1 is displayed in the Comment card in Icon Info. If this resource is not present, you can generally add it using the Comment card.

A file's `rComment` resource with an ID of 2, if present, is used when the user tries to open a document that can't be launched. The message in the resource usually explains how to properly use the file. (For one example, try launching the system resource file, Sys.Resources.) This happens only after Finder extensions decline to handle opening the document by refusing `finderSaysBeforeOpen` and `finderSaysOpenFailed` broadcasts. The message shows up in an alert window, with a Note icon.

Optimizations

Scrolling speed in windows has been dramatically improved. This was accomplished by checking the visible region (`visRgn`) of the window which was being updated. If an icon isn't inside the `visRgn`, it isn't drawn.

Finder 6.0 stores and searches the information in Finder.Data files in a more efficient order, so folders open much faster. The new Finder.Data format is also more compact than previous Finder.Data versions.

Because the format was also changed to allow long pathnames, previous Finder versions do not recognize Finder 6.0's Finder.Data files (windows open as if no Finder.Data file is present).

When a folder which does not contain a Finder.Data file or a network folder is opened, icons are placed in the window much faster, dramatically improving the time required to open a folder. Opening a folder on a local hard-drive containing 150 items took approximately 29 seconds with Finder 1.3. Finder 6.0 opens the same folder in under 3 seconds.

When cleaning up the normal way (that is, not using Option-Clean Up to clean up by name), the Finder now only animates the drawing of the icons which are visible or will become visible. This speeds up Clean Up dramatically.

Control-Opening (or Control-Double clicking) a folder prevents the Finder from reading any Finder.Data file in the folder. Folders containing many items open much faster this way.

When files or folders are dragged, the outlines which are dragged belong to only those items which have some portion of their icon or name inside the current `visRgn`. There is a limit of dragging 100 item outlines, but because more than 100 items are rarely visible in a window, this is rarely an issue.

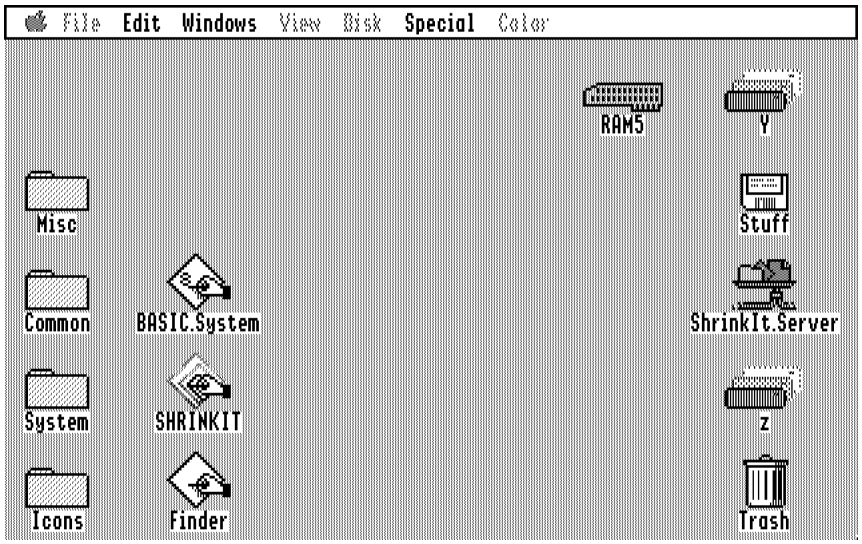
Enhancements

Finder's About Box

The modeless About box has useful information in it which is updated automatically every 15 seconds. The About box shows the total physical memory in the system and the total available memory. This is different from **free memory** in that all purgeable memory is included, whereas **free memory** in the control panel is just a list of the free memory blocks in the system Thus, **available memory** is a more accurate portrayal of the state of the user's system. The amount of memory which the Finder, System, Desk Accessories, and Setup Files are using is also shown. The amount of memory included in the **System** calculation includes the memory allocated to the system RAM disk. The version of the Finder and the version of the System software is also shown. The version and version string information is obtained from the Finder from its `rVersion(1)` resource and from the system resource file from its `rVersion($07FF0001)` resource.

User-Interface Changes

Folders are now allowed on the desktop.



Tweaks

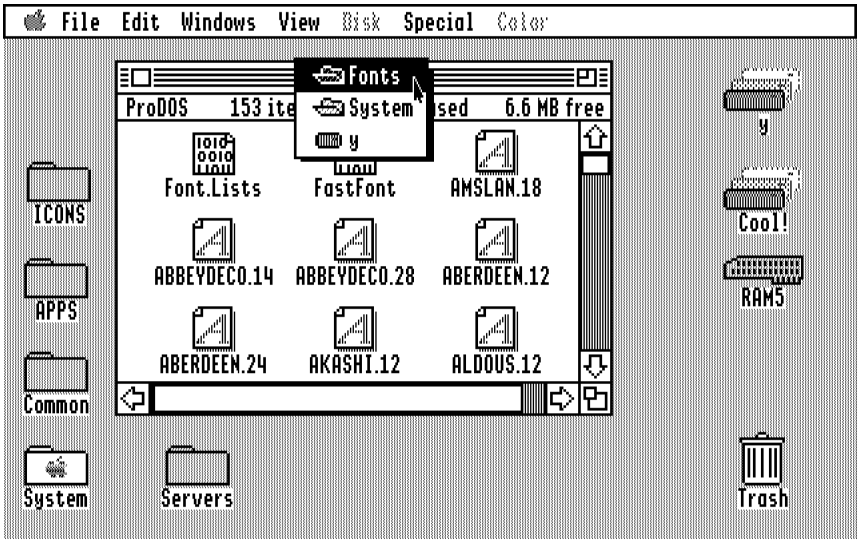
You can now double click the name of a volume or folder icon and have that icon open. Previously the Finder would think you wanted to rename the icon and create a LineEdit control. (This was very annoying.) This allows users more “slop” when they want to open folders that use a list view (so that the user doesn’t have to try to hit the tiny icon to the left of the name).

Files on the desktop can now be renamed. “Put Away” removes a disk from the desktop.

After closing a window, if there aren’t any selected icons in any other window or on the desktop, the Finder automatically selects the icon that the zoom effect zoomed into.

All icons are now deselected if the Esc key is pressed. If an icon has a rename field, that icon is deselected. Pressing Return while a single icon is highlighted gives that icon a rename field.

Holding down the Command key while clicking the title of the front window shows a Pop-up menu containing a list of folders indicating the hierarchy of the current window. Selecting a folder from the Pop-up opens the chosen folder’s window. Holding down the Option key while selecting a folder from the Pop-up closes the current window after opening the selected folder.



The Windows Menu

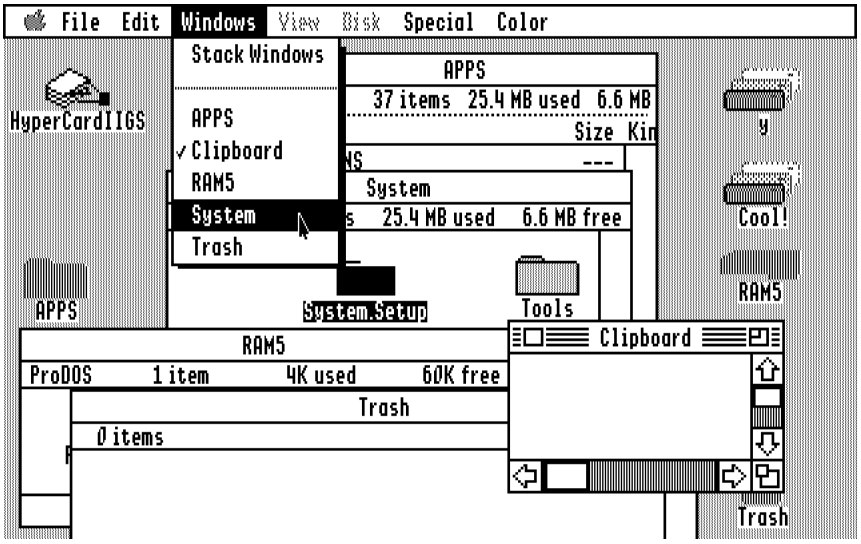
The Apple IIGS does not have an enormous desktop. This can cause a tremendous amount of clutter on the desktop as windows completely overlap one another. A problem frequently arises of how to quickly find a particular window. The Windows menu provides two solutions: Stacking the windows such that their titles are immediately visible, and showing the windows that the user has opened in alphabetical order, allowing any to be immediately shown. The windows menu also gives the user a quick reminder of which windows are open without having to manually dig through a pile of open windows.

Opening a window adds the window's title to the Windows menu, while closing a window removes the title from the Windows menu. The item for the front window always has a check mark by it.

Choosing a window from the Windows menu brings that window to the front. Option-choosing a window from the windows menu closes that window (without bringing it to the front).

The Windows menu does not apply to desk accessories. If a control panel window or desk accessory window is the frontmost window, no item in the Windows menu has a check mark. The Windows menu is dimmed when no Finder windows are open. Omitting the names of desk accessory windows from the Windows menu simplifies the user's perception of the desktop in that there is only a single reference to a desk accessory. Each desk accessory is listed once in the Apple menu—not twice, in both the Apple menu and Windows menu. Having the desk accessory available from a single point means the user does not wonder if there is a difference between choosing the it from the Apple menu and choosing it from the Windows menu.

Windows with duplicate names appear in the Windows menu, and the menu item which corresponds to a given window will not change as long as that window or any of the duplicates are not closed (or additional duplicates opened).



User-Interface Bugs Squashed

In previous versions of the Finder, when a user was dragging a folder to a destination window, the Finder wouldn't realize that it shouldn't track in the window if the mouse was over the info-bar, or over the scroll bars, or the grow box, or the title. As a result, a folder might exist under the info bar, and a file dropped on the info bar would vanish into the folder. This was very bad, and has been fixed.

Options: Tunneling, Reverse-Tunneling, and Close All

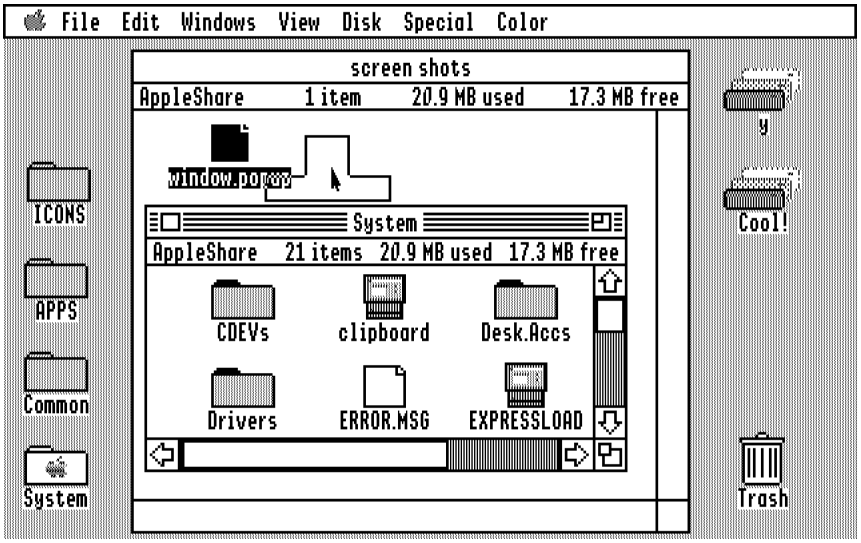
If the user double-clicks a folder while holding down the Option key, the Finder opens the folder and then closes the current window. This is called "Tunneling."

If the user Option-selects a folder from the pop-up in the title bar of a window, the Finder opens the selected folder and then closes the current window. This is called "Reverse Tunneling."

If the user Option-clicks the close box of a Finder window, the Finder closes all open windows.

The (Formerly) Impossible Drag

Below, the user wants to move the icon called “window.popup” into the window named “System.” The user intends to drag the icon from its current position into the visible, open window. However, as soon as the user presses the mouse button with the cursor over the icon, the window named “screen shots” comes to the front, and the window named “System” disappears behind “screen shots.”



Finder 6.0 solves this problem the same way Macintosh Finder 7.0 does. The arrangement of windows stays the same until the user releases the mouse button. If the icon is dragged into the “System” window, the file will be copied and “System” left as the front window. A click to just select the icon, select a group of icons (lasso them), or a drag to a position in the same window will bring “screen shots” to the front.

Command, Option, and Control Keys

Finder 6.0 has a number of shortcut keys. First, holding down the Control key while opening a window forces the Finder to **not** read a Finder.Data file if present in the folder being opened. This can save an enormous amount of time when opening a folder with a large number of items.

Holding down the Control key while coloring an icon reverses the Preference setting for “*Color selected icon’s background instead of its outline.”*

Holding down the Control key when closing a window reverses the Preference setting for “*Save Finder data to disk.”* Note that this can be used in combination with other command keys such as “Close All” (Option-close) which will close all the windows and also reverse the preference setting for all the windows. The Control key also works when Option-closing using the Windows menu.

Holding down the Option key when double-clicking a document forces the Finder to put up the Standard File dialog used in “Locating” an application for that document. Choosing “Open” from the Standard File dialog runs the chosen application only once and does not bind the document to the chosen application.

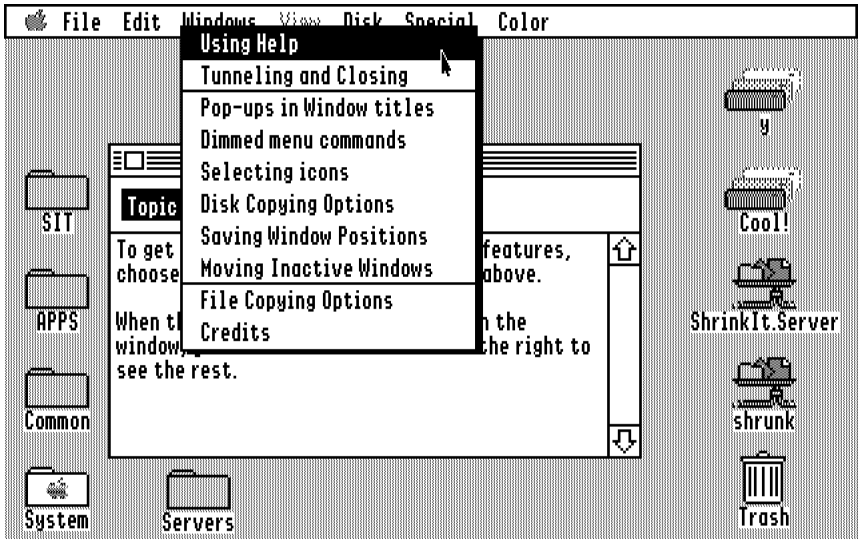
Holding down both the Option and Control keys while double-clicking a document forces the Finder to put up the Standard File dialog used for “Locating” an application for that document. Option-Control-double-click forces the Finder to permanently bind the document to the chosen

application. The document can be re-bound later by Option-Control-double-clicking the same type of document again.

The link from the document type to the application (the “binding”) is stored in the Finder’s “`Desktop`” file in the Icons folder of any on-line volume (usually either the volume containing the application or the start up volume).

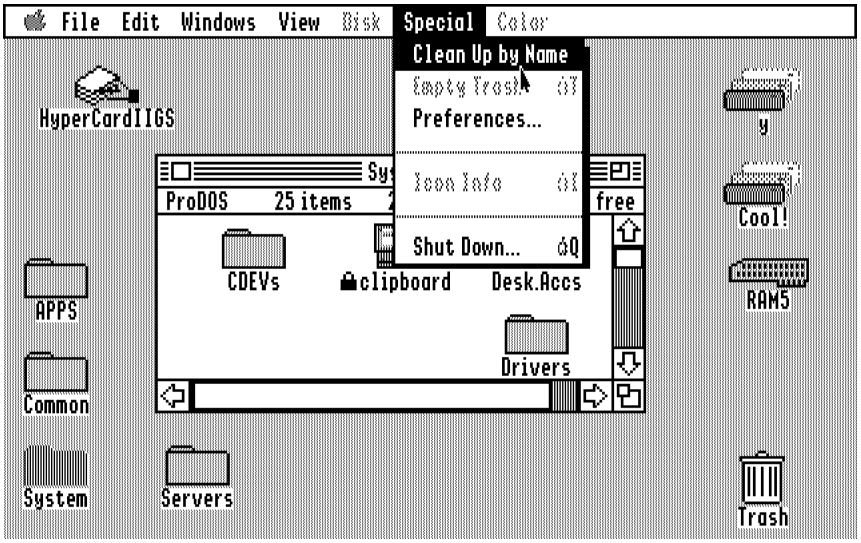
New Help System

A new help system is available in Finder 6.0. The help system uses a Pop-up menu and a TextEdit box. Choosing a topic from the pop-up menu displays text about that topic in the text box. The help system remembers the last chosen topic—if the help window is closed and later re-opened, the same topic will be shown. The text shown can be copied onto the clipboard, but it can’t be modified in the help window.



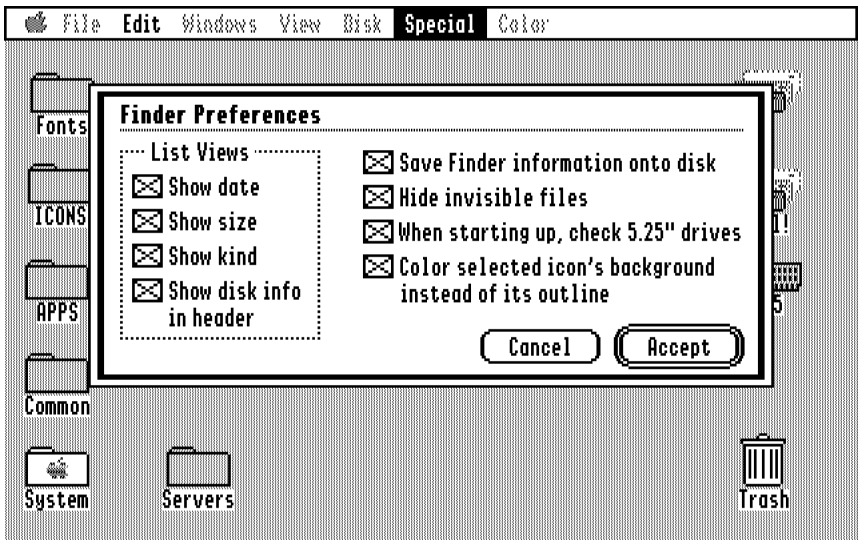
Clean Up (and Clean Up by Name)

Option-choosing *Clean Up* changes the menu item to *Clean Up by Name*. When chosen, *Clean Up by Name* alphabetizes the files in the window and aligns them to the grid used by *Clean Up* without animating the movement of each file. The width of the window determines how many icons will fit in one row. The wider the window, the more icons will fit. If the view is set to "by Small Icon" the icons will be placed into columns based on the height of the window. The taller the window, the more icons will be placed into each column.



Preferences

Finder 6.0 has a very different set of preferences, as shown below:



The "List Views" options are new to Finder 6.0. They allow the user to remove or add individual fields from their windows. The effect of "Show disk info..." is shown prominently in the screen shot under "Window Information Bars."

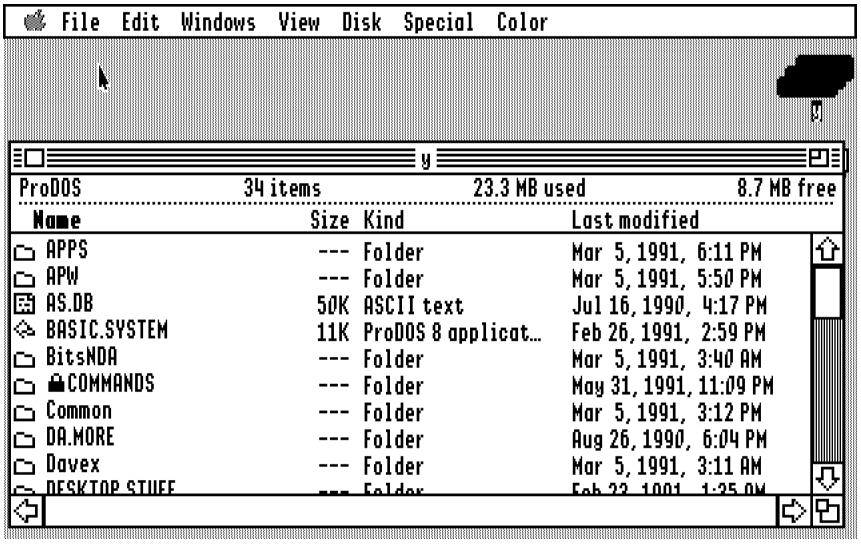
If a View that is not checked in the “List Views” is selected for a particular window, that view is shown anyway. The concept behind this is that the user should have more direct control over the individual views which are presented. “List Views” is a control which changes all fields in all windows. The Views menu affects only the front window.

When starting up, check 5.25" drives is a new option in Finder 6.0. If the user deselects the checkbox, the Finder only shows the drive icon for any connected 5.25" drives, instead of showing both the drive icon and polling the drive for a diskette. When entering the Finder, this avoids the incredibly annoying racket created when the 5.25" drive recalibrates its drive head. This option only applies if the user has 5.25" drives and has the 5.25" driver installed.

Window Information Bars

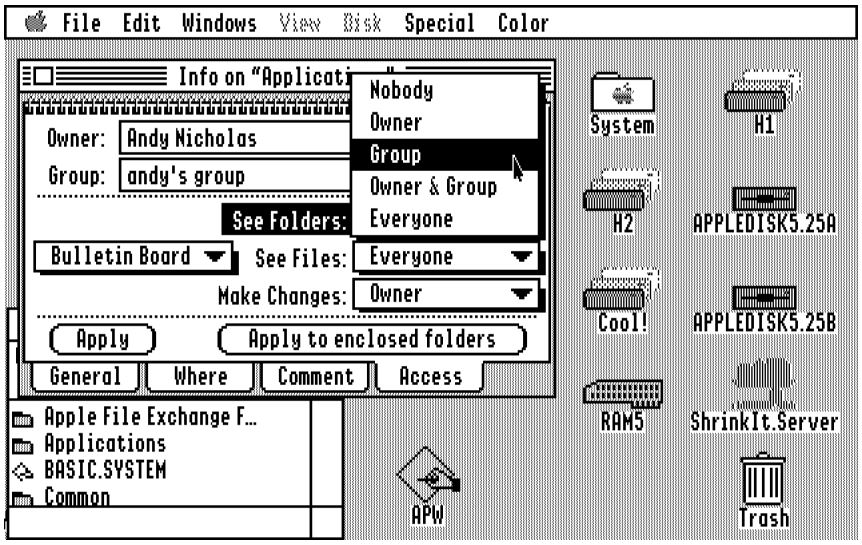
The window information bars in Finder 6.0 are much improved. First, the file system appears prominently, as does the amount of free and used space. The free and used space appears in appropriate units – K for values from zero K to 1023K, and megabytes (to the nearest 0.1M) for values of 1.0MB and up.

The “Size” of a file is calculated to the nearest K, not the nearest 1/2 K as before.



Icon Info

Icon info has not drastically changed for Finder 6.0, but it sports a few improvements. First, as shown below, the pop-up menus for network folder privileges have drop-triangles to match the latest human interface guidelines.



Comments are now allowed in all files, not just files on an AppleShare file server. Comments are not allowed on local folders or volumes. Comments can be edited and can be any length as long as they are saved back to the file's resource fork. (If a comment is only saved to a server's database it is truncated to 199 characters.) If a file on a local volume does not have a resource fork, an *Add Comment* button appears on the Comment card. Clicking *Add Comment* gives the user a warning that adding a resource fork to the file makes it unusable by ProDOS 8. Comments cannot be attached to $FF (ProDOS System) and $F9 (GS/OS System) files on local volumes.

File type $BC load files and older style icon files can now be inactivated. And, finally, if a file contains an `rVersion(1)` resource, the contents of that resource appear on the General card.

Launching

The Finder now allows smooth transitions when launching files which have their auxiliary type set to indicate that they are a desktop application. Instead of showing an inverse stripe along the top of the text screen indicating the application being launched, the Finder now leaves the Super Hi-Res screen enabled and shows the name of the application in the menu bar. Applications following the rules in the File Type Notes for file type $B3 or file type $B5 benefit from this enhancement.

New Icon Matching System

Finder 6.0 employs a radically different scheme for keeping track of application and document icons. Finder 6.0 employs the concept of a "desktop database" which keeps track of which applications "own" which document icons. In the absence of a definite document "owner," the existing applications are allowed to "vote" on which application can handle the document the best.

Note Voting is not implemented in Finder 6.0. Set the voting bits correctly anyway, in case voting is implemented later.

Simple pseudo-code follows of the Finder's actions after the user double clicks an icon.

```
On opening an icon:
  begin
      if icon is an application (file type $B3 or $FF)
          begin
            If the application's pathname is already in an existing
              rFinderPath resource, then just launch.

                  If the application's pathname isn't in any existing rFinderPath
                      resource, then
                      begin
                          If file doesn't have a resource fork, just launch.

                          If file has a resource fork and an rBundle resource,
                             then move the rBundle resource into the finder's
                             desktop database for that volume... then launch.

                             When the desktop database is updated, the rVersion
                             resource from the application is compared against
                             known rVersion resources so that duplicate entries
                             aren't made to the desktop.
                      end.
         end.
      if icon is a document
         begin
         From the icon object in memory, the Finder finds who owns this document.
         If no application owns the document,
             begin
                Presented the user with a Standard File dialog stating that an
                application couldn't be found for this document.  The user can either
                cancel the launch or choose an application.  If the user chooses
                an application, the Finder generates a "OneDoc" structure and saves
                it in the Finder's desktop database as an rBundle resource.
             end.

         If an application owns the document,
             begin
                 If the application still exists then launch.
                 If the application doesn't exist (meaning that it was moved or
                   renamed), then give the user a Standard File dialog asking what
                   application to launch.  If the user chooses an application, then
                   update the existing rFinderPath resource to contain the pathname
                   of the new application.
             end.
        end.
  end.
```

Application Notes

Note This section makes reference to several resource types. These resources are documented in Appendix B and in Apple IIGS Technical Note #76.

If you are writing an application and wish to take advantage of Finder 6.0's abilities, there are a few things to keep in mind.

For the Finder to keep track of your application, your application **must** have an `rVersion(1)` resource and an `rBundle(1)` resource. When the Finder first launches your application, it copies the `rBundle`, `rVersion`, and any needed `rIcon` resources into the `Desktop` file on the volume on which your application currently exists (or onto the boot volume if that's not possible). Every time thereafter when the Finder launches your application it checks the `rVersion` of your application against the `rVersion` which it copied.

Do not include any `rFinderPath` resources in your application. Any `rFinderPath` resources in your application are ignored when the Finder launches your application and copies your application's `rBundle`, `rVersion`, and `rIcon` into the `Desktop` file.

Because of a quirk in the Finder's design, it is better to group `OneDoc` structures together within an `rBundle` which use the same `rIcon` ID for the same icon. (That is, several `OneDoc` structures can share an icon image.) If the same icon is to be displayed for multiple sets of criteria and the `OneDoc` structures which refer to the shared `rIcon` are grouped together in the `rBundle`, the Finder loads each `rIcon` resource into memory only once, instead of once per reference. This can drastically reduce the amount of memory taken by `rIcon` resource and the amount of time the Finder spends updating and loading `Desktop` files.

All the `OneDoc` structures in your `rBundle`, once copied to a `Desktop` file, share the same `rFinderPath` resource. If the Finder notices that your application has moved, it changes only a single `rFinderPath`, thus linking all the `OneDoc` structures to the application's new location on disk.

△ **Important** When you create `OneDoc` structures for `rBundle` resources, always set the `NumResults` field to 4 or greater. The large icon and small icon fields of an `rBundle` must always exist because of a bug in Finder 6.0. If you want the Finder to use the generic icon for either the large or small icon, use `NIL` as the `rIcon` resource ID. △

Dependency Rules

Note This section makes reference to several resource types. These resources are documented in Appendix B and in Apple IIGS Technical Note #76.

The `Desktop` files in the `Icons` folder of a volume are special. `Desktop` files contain `rBundle`, `rVersion`, and `rIcon` resources copied from applications which the Finder has launched, and from files other than applications on which the user has used "Icon Info".

A useful third-party utility would be one which allowed editing, organizing, and updating users' `Desktop` files. In order to expedite this, there is one golden dependency rule to keep in mind:

"Any resource referenced directly or indirectly by an `rBundle` belongs to that `rBundle`."

An `rFinderPath` resource is only referenced by a single `rBundle` resource. There may be multiple references to an `rFinderPath` within a single `rBundle` (by multiple `OneDoc` structures), but multiple `rBundles` must not reference a single `rFinderPath`.

An `rVersion` resource may be referenced by multiple `rFinderPath` resources, but those `rFinderPath` resources must only be referenced by a single `rBundle` resource (that is, an `rVersion` must not be indirectly connected to two or more `rBundle` resources).

rIcon resources are referenced by a single rBundle. There may be (and should be, to save memory and disk space) multiple references to a single rIcon resource by a single rBundle.

When removing an rBundle from a Desktop file, it is safe to assume that all the rIcon resources referenced by the rBundle may be removed, that the rFinderPath resources referenced by the rBundle may be removed, and that the rVersion resources referenced by the rFinderPath resources referenced by the rBundle may be removed without causing dependency problems.

When Finder 6.0 removes an rBundle resource because a newer version of an application needs to have its (newer) rBundle resource copied into a Desktop file, Finder 6.0 only removes the rBundle resource. Any rIcon and rFinderPath (and rVersion referenced by the rFinderPath) resources referenced by the removed rBundle are left "dangling" (not connected).

Icon Loading and Searching Order

When starting up, the Finder loads icons from each device from lowest numbered on-line device (#1) through the highest numbered on-line device. On each device, the Finder first loads any Desktop file from the icons folder, then loads all non-inactive old-style (Pre-Finder 6.0) icon files in the Icons folder. When the user inserts a disk, the Finder first loads the Desktop file from the inserted disk, then converts all the old-style icon files.

The Finder searches for icons based on the order they were loaded. This means that the Finder will look through the icon from device #1 first before looking through all the icons from device #2, and so on.

Because of the order that icons are loaded, icons attached to rBundle resources in Desktop files will be matched first before an icon which is set to match the same criteria in an old-style icon file.

Within Desktop files, the Finder loads rBundle resources sequentially, from lowest ID through highest ID. Finder searches through the OneDoc structures from the first loaded rBundle to the last loaded rBundle.

The Finder always searches its built-in icons last.

Interprocess Communication

Anything that communicates with the Finder through the Tool Locator AcceptRequests/SendRequest mechanism is a "Finder Extension." Permanent Inits and desk accessories can be Finder extensions, as can control panels.

Communication between the Finder and extensions goes both ways. When the Finder needs to send information to extensions, it calls AcceptRequests with a finderSays code (these are documented in the next section), allowing any interested AcceptRequests clients to receive the information.

When an extension needs the Finder to do something, it calls SendRequest with a tellFinder code, directing the request to "Apple~Finder~", with a sendHow value of sendToName+stopAfterOne.

Finder extensions can be stored on disk as permanent initialization files. When executed, an extension should call `AcceptRequests` (in the Tool Locator) to install a "request procedure" that communicates with the Finder.

In the case of a permanent initialization files, the user ID value passed to `AcceptRequests` must be the file's unmodified memory ID, as returned from `MMStartUp` or as found in the A register when the initialization file was executed. (In exceptional cases, a Finder Extension may need to install more than one request procedure. This is OK, but they should all have the same user ID.)

Tip The user ID returned by the `UserID` function in ORCA/C and ORCA/Pascal has the auxiliary user ID set to $0100. To get the unmodified memory ID, use a bitwise and operation to mask out the auxiliary user ID, anding the value returned by the user ID function with $F0FF.

If an extension needs to access data files, it may use the value of the "@" prefix any time between `finderSaysHello` and `finderSaysGoodbye`. The "@" prefix is the directory the Finder is running from, or (if the user launched the Finder from a server) the user's User folder on the file server.

The FinderExtras Folder

Many Finder extensions are NDAs or Inits, which are kept in memory at all times (except for any dynamic segments). If an extension is useful only from the Finder, you may want to free up memory while in other applications by keeping the extension in the FinderExtras folder within the Finder's folder (that is, usually *:System:FinderExtras).

There is a special file type/auxiliary type combination for Finder extensions designed for FinderExtras: file type $BC, auxiliary type $0001.

Before `finderSaysHello`, the Finder loads and executes any Inits or Finder extensions in FinderExtras. After `finderSaysGoodbye`, it removes each FinderExtras extension from memory by sending the `srqGoAway` request, removing all Request Procedures with the extension's memory ID, and finally calling `UserShutDown` on its memory ID.

Extensions in the FinderExtras folder must **not** remain connected to the system in any way after receiving the `srqGoAway` request, or the system will crash.

`finderSays` Codes

Any entity that has called `AcceptRequests` can receive these requests; they are sent to all request procedures. The requests are sent by the Finder.

This is a list of the request codes the Finder sends. A description of each code follows.

reqCode	Name
$0100	`finderSaysHello`
$0101	`finderSaysGoodbye`
$0102	`finderSaysSelectionChanged`
$0103	`finderSaysMItemSelected`
$0104	`*finderSaysBeforeOpen`
$0105	`*finderSaysOpenFailed`
$0106	`*finderSaysBeforeCopy`
$0107	`finderSaysIdle`
$0108	`*finderSaysExtrasChosen`
$0109	`*finderSaysBeforeRename`
$010A	`*finderSaysKeyHit`

The Finder sends the requests which are marked with an asterisk (*) so they are received by only the first request procedure to handle the request. All other requests can be handled by any number of procedures.

`finderSaysBeforeCopy` $0106

Before the Finder does a file copy or a `ChangePath` call to move a volume on a volume, for each file, the Finder calls `finderSaysBeforeCopy`.

▲ **Warning** No not modify any of the parameters passed to `dataIn`. They are for examination only. ▲

When the Finder makes a `finderSaysBeforeCopy` call, the `dataIn` parameter is a pointer to a structure with the following format:

$00	*pCount*	**Word**—Parameter count (at least 2)
$02	— *srcPath* —	**Long**—Pointer to GS/OS source pathname
$06	— *destPath* —	**Long**—Pointer to GS/OS destination pathname

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*abortFlag*	**Word**—$0000 = continue; $0001 = abort; $0002 - $FFFF reserved

`finderSaysBeforeOpen` **`$0104`**
`finderSaysOpenFailed` **`$0105`**

When the user opens a document or application icon, the Finder sends `finderSaysBeforeOpen`. If the request does not get handled, the Finder tries to find an appropriate application to launch for the document. If that doesn't work, the Finder sends `finderSaysOpenFailed` to give extensions a chance to handle the request, knowing that no application was found.

▲ **Warning** When you receive `finderSaysBeforeOpen`, do not assume the Finder is necessarily present! See discussion below. ▲

(If the user opens several icons at once, the Finder makes one `finderSays` call for each icon, in turn. Also, if by opening a document the Finder is about to attempt to launch an application, the Finder sends a separate `finderSaysBeforeOpen` for the application after sending `finderSaysBeforeOpen` for the document.)

The Finder does not send `finderSaysBeforeOpen` or `finderSaysOpenFailed` when opening a folder.

The modifiers field contains a useful set of modifiers. Note that the Command key is masked out of the modifiers field if the user hit a Command-key combination to invoke `finderSaysBeforeOpen`. So, if the user has hit Command-O to call the Finder's "Open" function, the bit for the Command key will be clear in the modifiers field.

Note that using the Shift key to trigger something for the extension isn't necessarily a good thing. There is a very thin line between Shift-clicking to select an icon, and Shift-double-clicking to open and do something extension-based to it.

When the Finder makes a `finderSaysBeforeOpen` or a `finderSaysOpenFailed` call, the `dataIn` parameter is a pointer to a structure with the following format:

$00	*pCount*	**Word**—Parameter count (at least 6)
$02	— *pathPtr* —	**Long**—Pointer to GS/OS pathname
$06	— *rect* —	**Long**—Pointer to rectangle to zoom out from (or `NIL` if none)
$0A	*fileType*	**Word**—File type
$0C	— *auxType* —	**Long**—Auxiliary type
$10	*modifiers*	**Word**—Modifiers
$12	— *iconObj* —	**Long**—`theIconObj` (`NIL`, or handle to the icon object being opened)
$16	*printFlag*	**Word**—print flag (0 for Open; nonzero for Print)

All seven of these fields are always present, including `printFlag`, even though Finder 6.0 sets the parameter count to six.

Tip Be sure to check printFlag to distinguish an Open from a Print. Try choosing Print from the File menu to make sure your extension does not accidentally treat Print like Open.

`dataOut` buffer:
Reserved

Can something besides the Finder sends `finderSaysBeforeOpen`?

Yes. Sending `finderSaysBeforeOpen` is permitted in the desktop environment, as long as sufficient tools are started to support NDAs.

It may be useful for NDAs or other utilities to send `finderSaysBeforeOpen` to ask other utilities to behave as they would if a certain file were doube-clicked from the Finder. For example, the Control Panels NDA opens a control panel, and EasyMount mounts a server from an EasyMount document.

If you write any utility that accepts `finderSaysBeforeOpen`, be prepared to be in an environment other than the Finder. Do not assume that your `tellFinder` requests will succeed (you'll get error $0120 from `SendRequest` if the Finder is not present). Do not assume that all the tools the Finder starts, such as Text Edit, are started. Do not assume 640 mode.

If the environment your utility needs is not present, simply reject the `finderSaysBeforeOpen` request. This way other utilities still get a shot at handling it.

If you send `finderSaysBeforeOpen`, you must provide all seven parameters, and you should set the parameter count to seven (even thought Finder 6.0 sets it to six). If there is no rectangle for the accepting procedure to zoom out from, set the rectangle pointer to `NIL`. Always set `theIconObj` to `NIL`, since you have no icon object handle.

finderSaysBeforeRename $0109

When the user is about to rename an icon, the Finder sends `finderSaysRename`. If the request does not get accepted, the Finder tries to rename the icon. If the request is accepted, the Finder uses the return from `dataOut` to determine whether to perform the rename or not.

This call is intended as a simple stop-gap security measure to (for instance) keep someone from renaming something which really shouldn't be renamed (like the system folder on a server in a school).

▲ **Warning** Do not modify any of the parameters passed to `dataIn`. They are for examination only. ▲

When the Finder makes a `finderSaysBeforeRename` call, the `dataIn` parameter is a pointer to a structure with the following format:

$00	*pCount*	**Word**—Parameter count (at least 4)
$02	— *oldPathPtr* —	**Long**—Pointer to GS/OS old pathname
$06	— *newPathPtr* —	**Long**—Pointer to GS/OS new pathname
$0A	*fileType*	**Word**—File type
$0C	— *auxType* —	**Long**—Auxiliary type

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*abortFlag*	**Word**—Boolean; non-zero to abort the rename

finderSaysExtrasChosen $0108

`finderSaysExtrasChosen` notifies extensions that the user selected an item from the Extras menu. The menu item number of the selected item is in the low word of `dataIn`.

An extension that added an item to the Extras menu was assigned a menu item number when it called `tellFinderAddToExtras`; an extension must accept a `finderSaysExtrasChosen` request if and only if it owns the menu item number passed in the low word of `dataIn`.

The Extras menu title remains highlighted until the request processing is finished. If an extension puts up a modal dialog in response to `finderSaysExtrasChosen`, the Extras menu title remains highlighted the whole time, as it should. In this case, the extension's menu item should end with an ellipsis (for example, "Encrypt Files...").

If no extension accepts a `finderSaysExtrasChosen` request, the Finder calls `SysBeep2(sbOperationFailed)`.

`dataOut` buffer:

Reserved

finderSaysGoodbye $0101

The Finder sends `finderSaysGoodbye` early in its shutdown process to inform extensions that the Finder is going away (for whatever reason). Extensions must remove all of their Extras menu items at this time.

Note Extensions should be prepared to receive multiple `finderSaysGoodbye` requests, doing no harm on a redundant request. Calling `tellFinderRemoveFromExtras` with a stale item ID, or zero, is harmful. This is especially important if the extension ever refuses to go away (by rejecting an `srqGoAway` request, or by accepting it and returning a user ID of zero).

The Finder does not send redundant `finderSaysGoodbye` requests by itself, but redundant requests are unavoidable in some cases when a utility attempts to remove or replace an extension on the fly.

`dataIn` buffer:

Reserved

`dataOut` buffer:

Reserved

finderSaysHello $0100

The Finder sends `finderSaysHello` in its start-up process, every time the Finder is launched.

In response to `finderSaysHello`, an extension may make `tellFinderAddToExtras` requests to install menu items into the Finder's Extras menu. `tellFinderAddToExtras` is one of the few requests that can be made at `finderSaysHello` time. Most other requests should wait until the Finder has finished its start up sequence.

It is reasonable to watch for `finderSaysHello` and `finderSaysGoodbye` notifications to keep track of whether the Finder is present.

`dataIn` is a pointer to a buffer in the following format:

$00	*pCount*	**Word**—Parameter count (will always be at least 3)
$02	— *version* —	**Long**—Finder's version number (from its `rVersion` resource)
$06	*userID*	**Word**—Finder's user ID (for use with `SetCurResourceApp`)
$08	*iconSize*	**Word**—`iconObjSize`; for accessing fields of extended icons on the desktop

`dataOut` buffer:

Reserved

finderSaysIdle $0107

Finder broadcasts `finderSaysIdle` immediately before calling `TaskMaster` in its main loop. Finder extensions should be careful not to steal too much time at this point, or the Finder will become sluggish.

Tip The Finder can't accept many `tellFinder` codes in the middle of a `finderSays` call. (For example, you can't do `tellFinderOpenWindow` from within `finderSaysBeforeOpen`.) One way to handle this situation is to set a variable in your program rather than actually taking action, then wait for the next `finderSaysIdle` call to actually do the work.

`dataIn` buffer:

Reserved

`dataOut` buffer:

Reserved

finderSaysKeyHit $010A

If the Finder isn't able to deal with a keypress, it sends `finderSaysKeyHit` before returning to the Finder's event loop.

When the Finder makes a `finderSaysKeyHit` call, the `dataIn` parameter is a pointer to a structure with the following format:

$00	*pCount*	**Word**—Parameter count (at least 2)
$02	*key*	**Word**—Message from TaskMaster (character in low byte)
$04	*modifiers*	**Word**—Modifiers

`dataOut` buffer:

Reserved

finderSaysMItemSelected $0103

The Finder sends this whenever the user chooses any normal menu item (that is, not an $F000 range "Windows" item, not an $E000 range "Extras" item, and not a desk accessory item from the Apple menu). This request is sent to all Finder extensions. The Finder decides whether to continue with the menu selection based on whether any extension accepts the request and whether the `abortIt` flag is true.

Note Before acting on a `finderSaysMItemSelected` request, check the `abortIt` flag in the `dataOut` buffer. If the value is nonzero, some other Finder extension saw the request first and handled it before you. (Occasionally it is useful for an extension to see that particular menu items are being chosen, even if they've already been handled, so the Finder does not set the `stopAfterOne` flag to `SendRequest` when sending `finderSaysMItemSelected`.)

`dataIn` is a pointer to a buffer with the following format:

$00	*pCount*	**Word**—Parameter count (will be at least 3)
$02	*menuItemID*	**Word**—Menu item ID
$04	*menuID*	**Word**—Menu ID
$06	*modifiers*	**Word**—Modifiers (after menu was released; Command masked out)

`dataOut` is a pointer to a buffer with the following format:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*abortFlag*	**Word**—Boolean; non-zero means Finder won't continue menu selection

Here are the valid menu item ID values. Note that there is a separate code for `finderItemCleanUpByName`, even though the user sees them as the same item with a varying name.

```
$012D    finderItemAbout
$012E    finderItemHelp
$015F    finderItemNewFolder
$0160    finderItemOpen
$0161    finderItemPrint
$0162    finderItemClose
$0163    finderItemCloseAll
$0164    finderItemDuplicate
$0165    finderItemPutAway
$0166    finderItemValidate
$00FA    finderItemUndo
$00FB    finderItemCut
$00FC    finderItemCopy
$00FD    finderItemPaste
$00FE    finderItemClear
$0191    finderItemSelectAll
$0192    finderItemShowClipboard
$01C3    finderItemStackWindows
$01F5    finderItemByIcon
$01F6    finderItemBySmallIcon
$01F7    finderItemByName
```

```
$01F8    finderItemByDate
$01F9    finderItemBySize
$01FA    finderItemByKind
$0227    finderItemFormat
$0228    finderItemErase
$0229    finderItemVerify
$022A    finderItemEject
$0259    finderItemCleanup
$025A    finderItemEmptyTrash
$025B    finderItemPreferences
$025C    finderItemIconInfo
$025D    finderItemShutDown
$025E    finderItemCleanUpByName
$028B    finderItemColorBlack
$028C    finderItemColorBlue
$028D    finderItemColorYellowBrown
$028E    finderItemColorGray1
$028F    finderItemColorRed
$0290    finderItemColorPurple
$0291    finderItemColorOrange
$0292    finderItemColorPink
$0293    finderItemColorDarkGreen
$0294    finderItemColorAqua
$0295    finderItemColorBrightGreen
$0296    finderItemColorPaleGreen
$0298    finderItemColorPeriwinkleBlue
$0299    finderItemColorYellow
$029A    finderItemColorWhite
```

`finderSaysSelectionChanged` `$0102`

The Finder sends this whenever the set of selected icons **may** have changed. On receiving this notification, an extension can make the `tellFinderGetSelectedIcons` call to see what icons are now selected. Extensions that have menu items in the Extras menu may want to call `EnableMItem` or `DisableMItem` on those items at this point.

`dataIn` buffer:

Reserved

`dataOut` buffer:

Reserved

tellFinder Codes

Extensions use SendRequest in the Tool Locator to send these requests to "Apple~Finder~", using a sendHow value of sendToName+stopAfterOne.

reqCode	Name
$8000	*tellFinderGetDebugInfo
$8001	*askFinderAreYouThere
$8002	tellFinderOpenWindow
$8003	tellFinderCloseWindow
$8004	tellFinderGetSelectedIcons
$8005	tellFinderSetSelectedIcons
$8006	tellFinderLaunchThis
$8007	tellFinderShutDown
$8008	tellFinderMItemSelected
$800A	tellFinderMatchFileToIcon
$800B	tellFinderAddBundle
$800C	tellFinderAboutChange
$800D	tellFinderCheckDatabase
$800E	tellFinderColorSelection
$800F	tellFinderAddToExtras
$8011	*askFinderIdleHowLong
$8012	tellFinderGetWindowIcons
$8013	tellFinderGetWindowInfo
$8014	tellFinderRemoveFromExtras
$8015	tellFinderSpecialPreferences

tellFinder Environment

During a tellFinder request, the Finder automatically sets the CurResourceApp to the Finder and the current menu bar to the System menu bar. Before returning control to your program, the Finder restores your settings.

Request codes marked with an asterisk above can be made just about any time. The Finder accepts the other requests only at certain times: during finderSaysHello, finderSaysGoodbye, finderSaysExtrasChosen, finderSaysSelectionChanged, finderSaysKeyHit, finderSaysMItemSelected, finderSaysIdle, and while the Finder is in its TaskMaster call.

Typically, but not always, the Finder is prepared to handle requests when an NDA Action routine gets called, or when a run queue task gets called during SystemTask.

tellFinder requests attempted when the Finder is unprepared return Finder result fErrBusy.

dataOut buffer format

All dataOut buffers begin with a count word that tells the caller how many times a request was accepted. (SendRequest requires this.)

All tellFinder requests return a result code word in the second word of the buffer pointed to by dataOut. This is zero if the request was handled successfully; other values depend on the particular request being made, but usually nonzero values are error codes returned unchanged from the toolbox or GS/OS.

A `finderResult` code is:

$0000	No error.
$00xx	A GS/OS error.
$xxxx	A toolbox error.
$4201	`fErrBadInput`; bad input value.
$4202	`fErrFailed`; could not complete request.
$4203	`fErrCancel`; user cancelled operation.
$4204	`fErrDimmed`; menu item was dimmed.
$4205	`fErrBusy`; not now, the Finder has a headache.
$4206	`fErrNotPrudent`; can't add Finder's resources to desktop file.
$4207	`fErrBadBundle`; unknown `rBundle` version, or `rBundle` damaged.
$42FF	`fErrNotImp`; request not implemented.

askFinderAreYouThere $8001

If the Finder is present, it always accepts this request and returns no error.

`dataIn` is reserved and must be zero.

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—Boolean; non-zero means Finder is there

askFinderIdleHowLong $8011

The Finder returns the number of ticks since the last time there was any user activity.

`dataIn` is reserved and must be zero.

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—$0000 for normal completion or an error code
$04	— *ticks* —	**Long**—Tick count; number of ticks since last user activity in the Finder

tellFinderAboutChange $800C

Informs the Finder that the contents of a directory have changed. If there is an open window for this directory, the Finder re-reads the directory's contents and brings the window up to date.

This is useful when a desk accessory or Finder extension creates a new file, especially if that file is immediately useful to the user. (For example, the EasyMount Finder extension calls `tellFinderAboutChange` after creating a new server alias.)

`dataIn` is a pointer to the GS/OS pathname of a directory.

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—$0000 for normal completion or a GS/OS error code

`tellFinderAddBundle` $800B

Tells the Finder to examine the contents of an `rBundle` from the provided pathname and add the contents of the `rBundle` (including attached `rIcon`, `rFinderPath`, and `rVersion` resources) to the `Desktop` file. Because this call does not update the in-memory contents of the `Desktop` file, the Finder must be re-launched to incorporate any changes to any `Desktop` files.

`dataIn` is a pointer to a buffer which contains:

$00	*reserved*	**Word**—Reserved; must be zero
$02	— *bundlePath* —	**Long**—Pointer to GS/OS pathname
$06	— *deskPath* —	**Long**—Pointer to GS/OS pathname for desktop file
$0A	— *bundleID* —	**Long**—`rBundle` ID to put into desktop file

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—$0000 for normal completion or a GS/OS error code

tellFinderAddToExtras $800F

The Finder adds the specified menu item to the Extras menu, adding the Extras menu to the menu bar first if it is not already there. The item is added at the bottom of the menu unless otherwise specified.

Each time the Finder starts up, each extension receives the `finderSaysHello` message and makes zero or more `tellFinderAddToExtras` calls to add menu items to the Extras menu. Because no extension can determine how many other extensions will add items to the Extras menu, the Finder administrates dividing lines between groups of items. For the first item in every group extensions must set bit 15 of the menu item template's version field to tell the Finder that a dividing line should appear before that item, if necessary. (No dividing line is needed if there are no items above it in the menu; the Finder checks that.)

Always flag your first Extras item with a dividing line. Extensions should always set the dividing line bit of their first menu item; if you have a lot if items, you may want additional dividing lines.

Each dividing line created by the Finder is associated with the item immediately below it; removing that item automatically removes the dividing line.

If your extension already has one or more Extras items and needs to add more items later, adding the item to the bottom of the menu is not the right thing (other extensions may have added items below yours). In this case, set bit 14 of the menu item template's version field, and in the template's menu item ID field provide the item number of the item you wish to insert after. (This should be the ID of an item you previously added with `tellFinderAddToExtras`.)

When the user chooses the extra menu item from the Extras menu, the Finder broadcasts `finderSaysExtrasChosen`, passing the menu item ID that the Finder assigned to the extension during the `tellFinderAddToExtras` call. The extension that owns the menu item is expected to accept the `finderSaysExtrasChosen` request.

Note If you make any Menu Manager calls on your menu item, the Extras Menu, or the system menu bar, be sure to set the `CurResourceApp` to the Finder's value (as provided at `finderSaysHello` time), or the Menu Manager calls may fail. For example:

`GetCurResourceApp` (preserve present value).
`SetCurResourceApp` to the Finder.
`SetMItemName` on your item.
`CalcMenuSize` on the Extras menu.
`SetCurResourceApp` back to preserved value.

`dataIn` is a pointer to a menu item template, just like `InsertMItem2` requires (see *Apple IIGS Toolbox Reference: Volume 3*, page 37-15). The `itemID` field is normally ignored, since the Finder assigns an ID for the item. The `version` field has a special use.

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—$0000 for normal completion or a GS/OS error code
$04	*menuItemID*	**Word**—Menu item assigned by the Finder
$06	*menuID*	**Word**—Menu ID for the Extras menu

tellFinderCheckDataBase $800D

Given the data from an `rVersion` resource, `tellFinderCheckDatabase` lets you find out whether the information is already recorded in a Finder Desktop file. You can also ask the Finder to remove out-of-date information it finds and associates a new pathname with the matching `rVersion` information.

△ **Important** Two fields in the `dataIn` buffer are stored in a nonstandard format. The fields marked as SwapLong are four bytes long, but the first and second words are interchanged. (Instead of +0 +1 +2 +3, the order is +2 +3 +0 +1.) △

`dataIn` is a pointer to a buffer which contains:

$00	*flags*	**Word**—Update flags
$02	— *pathPtr* —	**SwapLong**—Input pathname
$06	— *rVersionPtr* —	**SwapLong**—`rVersion` resource

`flags` Set bit 15 to allow changes to the Desktop databases. The Finder makes no changes if this bit is clear. If it's set, the Finder removes the `rBundle` resources associated with matching `rVersion` resources, and it replaces the `rFinderPath` (if necessary) to refer to the `pathPtr`.

`pathPtr` When flags bit 15 is set, this is a pointer to new pathname to associate with the matching `rVersion` data. When flags bit 15 is clear, `pathPtr` is reserved and should be `NIL`.

`rVersionPtr` Pointer to image of an `rVersion` resource (must be locked).

Note A "matching" `rVersion` resource is one with the same application name string. The additional-information string is unimportant. An exact match is a matching `rVersion` with a matching version long word.

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—$0000 for normal completion or a GS/OS error code
$04	*versionResult*	**Word**—Result of comparison

`versionResult` is 0 if no match is found, $FFFF if an exact match is found, $8000 if a newer `rVersion` is found.

`tellFinderCloseWindow` $8003

Closes the specified Finder window.

`dataIn` is a pointer to a GS/OS input pathname. This can be a full pathname; one of "Trash", "Clip", or "About"; or a device name. The Finder calls `ExpandPath` on the pathname, so opening a window from a device name is possible.

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—$0000 for normal completion or an error code

`tellFinderColorSelection` $800E

Applies color selection to the selected icons. To revert an icon to its default colors, set the foreground and background colors to zero.

`dataIn` is formatted like this: $wx00yz00

Each icon's old foreground color is `and`ed with W and `or`ed with Y.
Each icon's old background color is `and`ed with X and `or`ed with Z.

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—$0000 for normal completion or an error code

`tellFinderGetDebugInfo` `$8000`

These are not useful except as described later in this chapter, or as defined by Apple.

`dataIn` is reserved and must be zero.

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—$0000
$04	*reserved1*	**Word**—Reserved
$06	*DP*	**Word**—Address of Finder's first direct page
$08	— *diskIcon* —	**Long**—Handle to first desktop `IconObj`
$0C	— *nameChainH* —	**Long**—Match-by-name chain reference
$10	— *filetypeBlock* —	**Long**—Pointer to list of match-by-file-type chain references
$14	— *deviceBlock* —	**Long**—Pointer to list of match-by-device-type chain references
$18	— *masterChainH* —	**Long**—Match-by-non-file-type chain reference
$1C	— *finderPathsH* —	**Long**—Handle of the list of `FinderPath` handle info records
$20	*finderPathsCount*	**Word**—Number of `FinderPath` records
$22	— *nameChainInsert* —	**Long**—Insertion point for the name chain
$26	— *reserved2* —	**Long**—Reserved
$2A	— *masterChain* —	**Long**—Insertion point for the master chain
$2E	— *reserved3* —	**Long**—Reserved
$32	— *chainTable* —	**Long**—Handle to chain table
$36	— *iconOffsetArray* —	**Long**—Handle to array of containing offsets within handles containing icons
$3A	— *iconHandleArray* —	**Long**—Handle to array of containing handles containing icons
$3E	*iconArrayUsed*	**Word**—Number of icons currently being used by Finder
$40	*iconArraySize*	**Word**—Number of icons that can be stuffed into current icon arrays above
$42	— *reserved4* —	**Long**—Reserved
$44	*reserved5*	**60 bytes**—Reserved for future parameters

`tellFinderGetSelectedIcons` $8004

Returns a handle containing information on each icon that is currently selected. The returned `stringList` handle is yours to deal with. When you're done with it, you must call `DisposeHandle` on it.

Each string is either a fully expanded pathname or the name of some other Finder icon, such as "Trash" or the device name of an AppleDisk 5.25 device.

In `dataIn`, bit 31 is set if you would like the return value to be in extended `stringList` format. (See "Extended `stringList` Warning," later in this section.) All other bits in `dataIn` are reserved and should be zero.

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—$0000 for normal completion or an error code
$04	— *wPtr* —	**Long**—Pointer to window containing the selected icons; `NIL` for the desktop
$08	— *stringHandle* —	**Long**—New `stringList` handle

A `stringList` handle's contents has the following format:

$00	*count*	**Word**—Number of pathnames following
$02	*wStrings*	**GS/OS input strings**—`count` GS/OS input strings

If bit 31 is set in `dataIn`, the extended `stringList` handle has the following format:

$00	*count*	**Word**—Number of variable length records following
$02	*records*	**variable**—records

Each record consists of:

$00	*offset*	**Word**—Offset to `WString1` (offset is from the start of the record)
$02	*iconBottom*	**Word**—Y coordinate of icon, icon view (bottom of icon)
$04	*iconMiddle*	**Word**—Y coordinate of icon, icon view (middle of icon)
$06	*iconText*	**Word**—Y coordinate of icon's text, list view
$08	*iconHeight*	**Word**—Reserved for the height of the icon for current view (not useful in 6.0)
$0A	*iconWidth*	**Word**—Reserved for the width of the icon for current view (not useful in 6.0)
$0C	— *iconHandle* —	**Long**—`iconObj` handle for this icon
$10	*wString*	**variable**—This field is at the offset specified in the first field of the record

Don't depend on the offset to the `WString` in each record in an extended `stringList` handle always being the same. Apple reserves the right to change the amount of information returned before each pathname.

Extended `stringList` Warning

Because of a problem in Finder 6.0, it is unsafe to use the extended `stringList` versions of the `tellFinderGetSelectedIcons` or `tellFinderGetWindowIcons` calls unless you first follow the instructions below, making a five-byte patch to the Finder in memory.

Accept the `finderSaysHello` request. Compare the Finder version long word to $0600A000 (version 6.0). If it matches, pass the `dataIn` value (the pointer to the `finderSaysHello` structure) to `FindHandle`. At offset +$2E5A in the block `FindHandle` locates for you, store the following bytes: $A0 $0A $00 $80 $3F $00.

```
;
; PatchFinder60
;
; It is unsafe to call tellFinderGetSelectedIcons (extended) or
; tellFinderGetWindowIcons (extended) in Finder 6.0.
;
; This patch makes it safe, but disables the iconHeight and
; iconWidth fields in the extended stringList data.
;
; The patch checks for Finder version 6.0 ($0600A000) and
; then locates the main segment by doing FindHandle on the
; finderSaysHello dataIn value.
;
; Then we copy a "LDY #$000A, BRA +$3F" to offset $2E5A.
;
PatchFinder60  start

               ldy #2                  offset to rVersion vers, low word
               lda [dataIn],y
               cmp #$A000
               bne notFinder60
               iny
               iny
               lda [dataIn],y          rVersion vers, high word
               cmp #$0600
               bne notFinder60

               pha
               pha                     space for FindHandle
               pei dataIn+2
               pei dataIn
               _FindHandle
               phd
               tsc
               tcd                     handle is at 3 (temp DP on stack)
               ldy #2
               lda [3],y
               tax
               lda [3]
               sta 3
               stx 5                   pointer to main segment is at 3
```

```
                ldy #$2E5A              offset into main Finder segment
                lda #$0AA0              first word of patch (LDY #$xx0A)
                sta [3],y
                iny
                iny
                lda #$8000              second word of patch
!                                       (high byte of LDY, BRA)
                sta [3],y
                iny
                iny
                lda #$003f              offset for BRA, and a superfluous $00
                sta [3],y

                pld
                pla
                pla                     discard handle

notFinder60     rts
                end
```

`tellFinderGetWindowIcons` $8012

Returns the icons in a particular window.

`dataIn` is a pointer to a Finder window (any directory window or the Trash window) or `NIL` for the desktop. The resulting `stringList` handle lists all the icons in the specified window (or on the desktop). Bit 31 controls the type of the `stringList` handle (extended or regular).

> **Note** If you set bit 31 to get an extended `stringList`, read the warning under `tellFinderGetSelectedIcons`.

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—$0000 for normal completion or an error code
$04	— *theHandle* —	**Long**—New `stringList` handle

The `stringList` value uses the same format as the one in `tellFinderGetSelectedIcons`.

`tellFinderGetWindowInfo` $8013

Returns information about a Finder window.

`dataIn` is a pointer to any Finder window. This call does not make sense for the `Desktop`, so `NIL` is not allowed.

▲ **Warning** Do not modify the strings you get from `tellFinderGetWindowInfo`; the Finder does not provide a copy that can be modified. ▲

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—$0000 for normal completion or an error code
$04	*windType*	**Word**—Window type (see table below)
$06	*windView*	**Word**—Window view (see table below)
$08	*windFST*	**Word**—File system ID for a directory window
$0A	— *windTitle* —	**Long**—Pointer to window's Pascal String title
$0E	— *windPath* —	**Long**—Pointer to window's GS/OS string pathname
$12	— *reserved1* —	**Long**—Reserved for future use
$16	— *reserved2* —	**Long**—Reserved for future use

The window types defined in Finder 6.0 are:

`windType`	Window Type
$0002	directory window
$0004	trash
$0008	clipboard
$0010	Icon Info window
$0020	Verify window
$0040	About window
$0080	Help window

The window views defined in Finder 6.0 are:

`windView`	Window View
0	by Icon
1	by Small Icon
2	by Name
3	by Date
4	by Size
5	by Kind

`tellFinderLaunchThis` $8006

The Finder performs an `ExpandPath` call on the input pathname and verifies that the file exists and has an application file type ($FF, $B3, or $B5). If all is well, the Finder launches the specified application the next time the main event loop gets control.

`dataIn` is a pointer to the following structure:

$00	*reserved*	**Word**—Reserved (use $0000)
$02	— *pathPtr* —	**Long**—Pointer to a GS/OS pathname

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—$0000 for normal completion or an error code

`tellFinderMatchFileToIcon` $800A

Asks the Finder to search its internal data structures from all the volumes which have been mounted so far to look for an icon which matches the specified search criteria.

△ **Important** Several fields in the `dataIn` buffer are stored in a nonstandard format. The fields marked as SwapLong are four bytes long, but the first and second words are interchanged. (Instead of +0 +1 +2 +3, the order is +2 +3 +0 +1.) △

`dataIn` is a pointer to a buffer which contains:

$00	*pCount*	**Word**—Parameter count; 10 or 11
$02	*vote*	**Word**—Voting bits; use $8000 (don't care)
$04	*match*	**Word**—Which match we want
$06	*fileType*	**Word**—File type
$08	— *auxType* —	**SwapLong**—Auxiliary type
$0C	— *fileNamePtr* —	**SwapLong**—Pointer to file name to match against
$10	— *createDateTime* —	**SwapLong**—Pointer to create date/time in GS/OS format
$14	— *modDateTime* —	**SwapLong**—Pointer to mod date/time in GS/OS format
$18	*access*	**Word**—Local access word
$1A	*flags*	**Word**—Set bit 15 for extended file; all other bits are reserved and must be 0
$1C	— *optionList* —	**SwapLong**—Pointer to `optionList` or `NIL`
$20	— *EOF* —	**SwapLong**—Combined (resource and data fork) EOF

See the description of the `rBundle` resource in Appendix B for a description of the voting bits used in the `vote` parameter.

The match parameter specifies the match number we want to use – 1 for the first match, 2 for the second match, and so on.

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—$0000 for normal completion or an error code
$04	— *offset* —	**Long**—Offset to matching `OneDoc` structure in `rBundle`, or `NIL` if no match
$08	— *bundleHandle* —	**Long**—Handle to `rBundle` structure which matches, or `NIL` if no match
$0C	— *smallIcon* —	**Long**—Offset to small icon to use (never `NIL`)
$10	— *largeIcon* —	**Long**—Offset to large icon to use (never `NIL`)
$14	— *pathHandle* —	**Long**—Handle to `rFinderPath`, or `NIL` if no one owns the icon

tellFinderMItemSelected $8008

Tells the Finder to execute a menu-based action, just as if the user had chosen the item from the menus. The allowed menu item values are listed under `finderSaysMItemSelected`.

You can only simulate the "normal" Finder menu items. You can't use this call to simulate menu items from Extras, to select windows from Windows menu, or to open desk accessories. (You can bring windows to the front with `SelectWindow` in the Window Manager, and you can open Desk Accessories with `OpenNDA` in the Desk Manager.)

Actions that cause the Finder to shut down do not occur until the Finder's main event loop regains control. This includes Open (if application or document icons are selected) and Shut Down.

You specify the modifiers in the same format as in the modifiers field of an Event record. For some menu commands, the Option key or other modifiers are important.

`finderResult` will be `fErrDimmed` if the specified menu item is disabled.

`dataIn` is a pointer to a buffer which contains:

$00	*menuItemID*	**Word**—Menu item ID
$02	*modifiers*	**Word**—Modifiers
$04	*flags*	**Word**—bit 15 = highlight menu title, all other bits reserved

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—$0000 for normal completion or an error code

tellFinderOpenWindow $8002

`tellFinderOpenWindow` opens the specified window.

`dataIn` is a pointer to a GS/OS input pathname. This can be a full pathname; one of "Trash" (the trashcan window), "Clip" (the clipboard), or "About" (the Finder's about window); or a device name. The Finder calls `ExpandPath` on the pathname, so opening a window from a device name is possible.

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—$0000 for normal completion or an error code
$04	— *wPtr* —	**Long**—Resulting window pointer; `NIL` if there was an error

`tellFinderRemoveFromExtras` $8014

The Finder removes a menu item from the Extras menu, and removes the menu if there are no items left.

If the menu item was flagged as the first item in a group when it was added, removing it automatically removes the dividing line, if one was needed.

The low word of `dataIn` is a menu item number previously assigned by `tellFinderAddToExtras`. The high word is reserved and must be zero.

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—$0000 for normal completion or an error code

`tellFinderSetSelectedIcons` $8005

The Finder selects the icons specified in `stringList`. If other icons are selected in the same window (or on the desktop, if the specified icons are on the desktop) then previously-selected icons remain selected.

To deselect all icons, pass a `stringList` handle with a pathname count of zero.

No errors are returned for pathnames in the `stringList` handle for which no icon can be selected.

`dataIn` is a regular (not extended) `stringList` handle. Do not set bit 31, or any other bits in the highest byte. The `stringList` structure is described under `tellFinderGetSelectedIcons`.

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—$0000 for normal completion or an error code

tellFinderShutDown $8007

`tellFinderShutDown` shuts down the Finder. The requested action does not happen right away. Instead, it happens the next time the Finder's main event loop gets control.

The high word of `dataIn` is reserved and must be zero. The low word of `dataIn` contains one of the following values:

0	Turn off power.
1	Restart system.
2	Quit from the Finder.

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—$0000 for normal completion or an error code

tellFinderSpecialPreferences $8015

Tells the Finder to set some special preferences for this execution of the Finder only. The preferences are not saved; the Finder must be re-told about the preferences each time it is run. In the future this may allow certain special behavior modifications to the Finder by third-party utilities.

It's OK to call `tellFinderSpecialPreferences` at `finderSaysHello` time.

`dataIn` is a pointer to a buffer which contains:

$00	*pCount*	**Word**—Parameter count; must be 1
$02	*trashHardDisks*	**Word**—Allow trashing hard disks?

The `trashHardDisks` parameter is a boolean flag that tells the Finder if the user is allowed to drag a hard disk into the trash. Code 1 if the user can drag the hard disk into the trash, and 0 if not.

`dataOut` buffer:

$00	*recvCount*	**Word**—Number of times the request was received
$02	*finderResult*	**Word**—Reserved. Finder 6.0 does not return a useful value here.

Internal Finder Data Structures

Finder uses two data structures for many of its operations: `iconObj` handles for icons, and `windBlk` structures for windows. Each window which belongs to the Finder has at least a portion of the `windBlk`. You can find the pointer to a window's `windBlk` by calling `GetWRefCon` (in the Window Manager) with the window's pointer.

Use this information with care, making conservative assumptions when possible. For example, be prepared to deal with unexpected values, such as a `windView` greater than five.

These data structures are read only! You should **not** change the values of any fields.

When possible, use `tellFinderGetWindowInfo` instead of examining the `windBlk` structure directly.

Block pointed to by `RefCon` (a window's `windBlk`)

`windIcons`	Long	Handle of first `iconObj` in window (`NIL` = none).
`windID`	Word	Window's ID number (see below).
`windView`	Word	View kind: 0 icon 1 small icon 2 name 3 date 4 size 5 kind
`windIc`	Long	Handle of `iconObj` that opened into this window.
`windDiskIc`	Long	Handle of disk icon that originally owns this.
`windItems`	Word	Number of items in window.
`windUsed`	Long	Number of bytes used in window.
`windFree`	Long	Number of bytes free on disk.
`windFST`	Word	FST ID of this window.
`windAccess`	Word	AppleShare access bits.
`windDirty`	Word	Flags word: bit 15 Used by Finder. bits 14-0 Reserved; ignore these bits.
`windTitle`	Block 54	Window's title (padded with spaces).
`windMenuItem`	Word	Window's menu item number.
`windMenuText`	Block 52	Window's title for the menu item.
`windDate`	Block 8	Window's date.
`windPath`	Block 991	Complete pathname; GS/OS input string.

Window ID numbers (`windID` field of window's info block)

name	value	use
`sysWindID`	$0001	System window
`dirWindID`	$0002	Directory window, all fields present
`trashWindID`	$0004	Trash window, fields end at `windMenuText`
`clipWindID`	$0008	Clipboard window, fields end at `windMenuText`
`infoWindID`	$0010	Info window, all fields present
`verifyWindID`	$0020	Verify or Validate window, fields end at `windMenuText`
`aboutWindID`	$0040	About window, fields end at `windMenuText`
`helpWindID`	$0080	Help window, fields end at `windMenuText`

`iconObj` record offsets

`icNext`	Long	Handle of next icon in list (`NIL` = no more).
`icLast`	Long	Handle of previous icon in list (`NIL` = no more).
`icMom`	Long	Window that icon is currently in.
`icWind`	Long	Window that icon is opened into.
`icDisk`	Long	Disk `IconObj` which owns this icon.
`icFlag`	Long	Defined below.
`icFType`	Word	Icon's file type.
`icFileInfo`	Long	File's Auxiliary type or Device's File System.
`icKind`	Long	Pointer to kind string; can be `NIL`.
`icy`	Word	Position of icon's bottom.
`icx`	Word	X coordinate of icon's middle.
`icTextY`	Word	Icon's Y coordinate when viewed by text.
`icTitleLen`	Word	Half the length of the icon's title.
`icName`	Block 34	Name of icon; a Pascal string.
`icLocalAccess`	Word	Icon's current local access.
`icForked`	Word	High bit set if file is extended.
`icFBlocks`	Long	Icon's file block size; device's used blocks.
`icFBytes`	Long	Icon's file byte size; device's total blocks.
`icCDate`	Block 8	Icon's file creation date.
`icMDate`	Block 8	Icon's file modification date.
`icIcon`	Long	Index into Handle and Offset arrays for this icon.
`icSmallIcon`	Long	Index into Handle and Offset arrays for this icon.
`icRBundle`	Long	Handle of `rBundle` which matched; `NIL` for no match.
`icOneDocOffset`	Long	Offset to `OneDoc` within `rBundle` handle.
`icInfo`	Long	Pointer to Info window; `NIL` if none.
`icDevNum`	Word	Icon's device number (device icons).
`icDevInfo`	Word	If icon is a device, the return from `DInfo`.
`icOptionList`	Word	Start of the `optionList` for this file.
`icFST`	Word	The file system for this file (in `optionList`).
	Block 36	Remainder of `optionList`.
`icNetworkAccess`	Long	The network access for this file if file is on a network.

The size of an `iconObj` structure is subject to change. Fields may be added to the end in the future. Use the `iconObjSize` field of `finderSaysHello` `dataIn` to locate the three extended `iconObj` fields.

Extended `iconObj` Record

`icOldPosY`	Word	Y coordinate in real mom window.
`icOldPosX`	Word	X coordinate in real mom window.
`icOwner`	Block 993	Pathname of icon's real owner.

`icFlag` Flags

ICSELECTED	$00000001	1 = selected; 0=normal.
ICOPENED	$00000002	1 = icon is opened; 0 = icon closed.
ICOFFLINE	$00000004	1 = off line; 0 = on line.
ICEXTENDED	$00000008	1 = extended ICONOBJ record.
ICLOCKED	$00000080	1 = icon is locked.
ICFORECLR	$00000F00	Icon's foreground (outline) color.
ICBACKCLR	$0000F000	Icon's background (fill) color.
ICNETACCESS	$000F0000	Network access rights (4 bits).
ICNETWORK	$01000000	1 = Network; 0 = local.
ICREADABLE	$02000000	1 = read access; 0 = no read access.

▲ **Warning** All bits are used by `icFlag`. **Do not** commandeer any `icFlag` bits for any reason. **Do not** change the setting of any of the `icFlag` bits. Not all the used bits are publicly documented. The Finder relies heavily on the `icFlag` field and changes to it by anything other than the Finder are sure to cause disaster! ▲

Finding an Icon Image from an Icon Index

This section tells how to find an icon image in the Finder after you get an index from the `icIcon` field, `icSmallIcon` field, or from the small and large icon indexes provided by `tellFinderMatchFileToIcon`.

After receiving `finderSaysHello`, call `tellFinderGetDebugInfo`. This returns 128 bytes of data in `dataOut`. Extract the following two handles from `dataOut`:

```
+054   Handle iconOffsetArray
+058   Handle iconHandleArray
```

Each array contains a four-byte entry for each indexed icon, starting with zero. Multiply the icon index by four and use it to index into the pair of arrays. The entry from `iconHandleArray` tells you what handle the icon is stored in, and the entry from `iconOffsetArray` tells you the offset within that handle. Dereference the handle and add the offset, and you have the address of a QuickDraw II Auxiliary style icon, suitable for `DrawIcon`.

Things I'd Pay Cash for if I Were a User (Ideas for Extensions)

(Editor's note: This section contains a wish list compiled by the Finder's program and documentation authors, Andy Nicholas and Dave Lyons. While it isn't really Finder documentation, it would be a shame to cut it!)

Dave and Andy think that the following would be really cool. No doubt people have come up with some similar ideas on their own and are already working on them. You might want to ask around and avoid duplicating somebody else's efforts.

- A Finder Extension which would allow me to set an interval at which the extension would automatically scan my on-line volumes for folders which had changed, and within the folders which had changed, call `tellFinderAddBundle` for every application with a resource fork. Or, a Finder Extension which would scan through all my on-line volumes after removing my existing `Desktop` files to keep my `Desktop` databases up-to-date.

- A scheduling utility that launches a selected application at a selected time or interval (such as a backup program). Of course, it shouldn't launch while the user is in the middle of something, so it should watch for `finderSaysIdle` and (from time to time) call `askFinderIdleHowLong` to see if the Finder has been idle for several minutes.

- A "power user" Icon Info window, letting the user view and edit the file type and auxiliary type of a selected file, show the sizes of the data and resource forks, etc. For example, a "Power User Info" item in the Extras menu could open a modeless System window for each selected icon. (To create a modeless System window, use `SetSysWindow` in the Window Manager, and `GetAuxWindInfo` in the Window Manager; see also the Desk Manager toolbox documentation in *Apple IIGS Toolbox Reference: Volume 1* and in this book.)

- A utility to re-mount Apple SCSI partitions that have been unmounted. (This utility could also use `tellFinderSpecialPreferences` to let the user drag hard-disk partitions to the trash to unmount them.)

- Extensions to view or edit various sorts of files, such as text/Teach files, Sounds, Fonts, and various kinds of graphics files.

- A Find File utility with options to automatically open the window containing the found item or items, or even to launch those items.

- A pair of Extras menu commands called "Preserve Selection" and "Restore Selection", which would remember and restore your selected Finder icons (idea by Ron Lichty).

- Extension to encrypt and decrypt files (select document icons and choose "Encrypt..." from the Extras menu).

- A way to select icons by wild card (pop up a dialog where the user enters "*.asm", and all icons with names ending in ".asm" in the front window are instantly selected). (Use `tellFinderGetWindowIcons` and `tellFinderSetSelectedIcons`. Maybe have a checkbox for "select on desktop.")

- A text-based Finder interface, in a System window, maybe supporting scripting. (Dragging icons around would be the hardest part—you might be able to simulate it using `FakeMouse` in the Event Manager.)

- Keyboard navigation (similar to Macintosh Finder 7.0) is an obvious choice. We're already working on one and hope to release it soon. If you're thinking of trying this yourself, note that there are some big problems we have to get around by cheating, using hard-coded offsets into the Finder 6.0 code. One problem is the speed hit of calling `tellFinderSetSelectedIcons` twice (once to deselect icons, once to select the new one...fast enough for most purposes, but not for keyboard navigation). Another problem is that icons tend to be renameable far too often for navigation to work well; even if you click on the icon itself (not its name), you still get a rename field, and keystrokes get used in the icon's name instead of going to `finderSaysKeyHit`.

- Not quite a Finder extension... An NDA to be used outside the Finder (maybe inside, too), which sends `finderSaysBeforeOpen` to anybody willing to receive it. The user would choose files using Standard File (or by choosing a folder from a customizable list, to start Standard File out in a useful directory; or by choosing a file from a customizable list). This would be useful for control panels (the Control Panel NDA will open them) and for EasyMount

documents. (It will mount the server volume, asking for password info if needed. The results in 320 mode are poor in EasyMount 1.0.)

Chapter 35 **Sound Control Panel**

This chapter describes Sound Control Panel 2.0, part of Apple IIGS System Software 6.0.

Overview of the Sound Control Panel

- The Pitch scroll bar lets you adjust the pitch of the Standard Beep. (The setting is stored in Battery RAM.)
- The Volume scroll bar lets you adjust the system volume. (The setting is stored in Battery RAM.)
- The "Give visual indication of sounds" checkbox tells the system whether to blink the border along with sounds. (The setting is stored in Battery RAM.)
- Event and Sound pop-up menus let you choose what sound to play for each of twenty-two different events. (The various events occur when something calls `SysBeep2` in the Miscellaneous tools. Many parts of the system do this, as does third-party software that takes advantage of System Software 6.0.)
- Each event can be set to any sound in your `Sounds` folder, to the Standard Beep, to Silence, or to Not assigned.
- Sounds are stored as named `rSoundSample` resources in the *:System:Sounds folder.

Files related to the Sound Control Panel

The following files are used by the Sound control panel:

Sound in *:System:CDevs. This is the Sound control panel itself.

Sound.Settings in *:System:Sounds. This stores your assignments from events to sounds.

Sounds are read from all extended files in *:System:Sounds that contain named `rSoundSample` resources (type $8024; see Apple IIGS Technical Note #76, Miscellaneous Resource Formats). This is the same sound format HyperCard IIGS uses.

Human Interface Details

Opening the Sound window

To open the Sound control panel, first open the Control Panels NDA by choosing "Control Panels" from the Apple menu (or type Command-Shift-Esc). Scroll to the Sound icon and double-click it (or type "so" Return).

If you're in the Finder, you can double-click the Sound control panel file in the folder, instead. (You can open the CDevs folder by holding down Option while choosing Control Panels, or by typing Command-Option-Shift-Esc).

The Sound window contains two scroll bars, one checkbox, and two pop-up menus, as described below:

The Scroll bars

The pitch scroll bar adjusts the pitch of the Standard Beep. There are fifteen levels. When you adjust the pitch scroll bar, you hear the Standard Beep, regardless of the sound showing in the Sounds menu.

The volume scroll bar adjusts the system volume. There are fifteen levels. When you adjust the volume scroll bar, the system beep plays. This may or may not be the Standard Beep; it depends how you've set it with the Pop-up menus.

The Checkbox

There is a "Give visual indication of sounds" checkbox. When the box is checked, the system blinks the screen border when playing certain sounds.

You can toggle the checkbox by clicking in the box or in the text, or by typing G or Command-G.

The system beep and the following `SysBeep2` events can have a visual indication:

$0000..$000C	Impossible operations, etc.
$0030	Disk request.
$0033	Bad disk.
$0050..$0059	`AlertWindow`
$0Exx	`ErrorWindow`
$0Fxx	Reserved.

The Event Menu

The event menu lists a subset of the defined `SysBeep2` codes. See the description of `SysBeep2` in the Miscellaneous Tools chapter for details on all of the possible `SysBeep2` codes.

When you choose an event, the Sound menu automatically shows you what sound is currently assigned to that event. The various sound events listed are:

Code	Name
	System Beep
$0050	Attention
$0033	Bad disk
$0008	Bad keypress
$0009	Bad input value
$0004	Can't click there
$0054	Caution alert
$0014	Disk ejected
$0013	Disk inserted
$0030	Disk request
$0043	Empty trash
$0042	Fill trash
$000A	Input field full
$0053	Note alert
$0052	Stop alert
$0031	System start up
$0015	System shutdown
$0005	Task completed
$000C	Task failed
$000B	Task impossible
$0041	Whoosh closed
$0040	Whoosh open
$0100	You Have Mail

The Sound Menu

Choosing an item from the Sound menu associates the current event with that sound, and immediately plays the sound as a sample. (To hear the current sound again, just press Return or Enter.)

The Sound menu lists "Not assigned", "Silence", and "Standard Beep," plus any sounds found in the `Sounds` folder. (If no sounds are present, the dividing line under Standard Beep does not appear.)

Only the sounds that presently have events mapped to them are kept in RAM; others are loaded from disk as needed. Note that sounds that stop being mapped to events remain in RAM until you reboot.

Copy Sound to Clipboard

Choose Cut or Copy from the application's Edit menu to copy the current sound (as shown in the Sound menu) to the clipboard. (You can't copy Not assigned, Silence, or Standard Beep.) This puts both the sound and its name onto the clipboard as a `sampledSoundScrap`, $0002, and a `textScrap`, $0000.

Implementation Details

`BootCDEV`

When Sound receives the `BootCDEV` message, it installs a Request Procedure, scans the *:System:Sounds folder for sounds, and loads the Sound.Settings file, if present.

If the request procedure can't be installed (usually because you have two copies of the Sound control panel in your system), Sound instructs the Control Panel to "X" out the Sound icon on the boot screen.

Contents of the Event Menu

The "System Beep" item is hard-coded, but all other events come from the `rTaggedStrings` ID=1 resources in the Sound file. (If somebody writes an editor for this standard resource type, power users will be able to easily customize their Event menu.)

The `rTaggedStrings` ID=1 resource is locked and fixed and should remain that way.

Contents of the Sound Menu

The "Not assigned", "Silence", and "Standard Beep" items are hard-coded. All other items are the resource names of `rSoundSample` resources found in files in the *:System:Sounds folder.

Sound scans for sounds at boot time, and whenever the Sound window opens.

`rSoundSample` Format

Just like HyperCard IIGS, the Sound control panel plays sampled sounds at the pitch indicated by the `relPitch` field, ignoring the sample frequency field.

See the HyperCard IIGS Technical Notes for more information.

Sound's Request Procedure

Here are the request codes the Sound control panel's request procedure can accept:

- $0001 `systemSaysBeep`

  This is the one `SysBeep2` uses. If Sound successfully plays a sound mapped to the given event, it accepts the request. The low word of `dataIn` is the event code; bit 31 of `dataIn` disables playing the sound (but it is still accepted normally).

- $0004 `srqGetrSoundSample`

  This is a request to return an `rSoundSample` handle corresponding to a given name. `dataIn` is a pointer to the Pascal string name. If Sound can provide such a handle, it accepts the request and fills in the `dataOut` buffer, which is formatted as follows:

$00	*recvCount*	**Word**—Required by `SendRequest`
$02	— *soundHandle* —	**Long**—Sound handle; not guaranteed to be locked
$06	*flags*	**Word**—Flags word

  If bit 15 of the `flags` word is set, the caller must dispose of the handle when done; bits 14-0 are reserved, and should be 0.

- $0005 `srqSynchronize`

  `dataIn` and `dataOut` are reserved and should be zero. Sound waits for any pending sound to finish playing; then accepts the request, whether it had to wait for anything or not.

- $0006 `srqPlayrSoundSample`

  `dataIn` is the handle of an `rSoundSample` resource. If Sound can play the sound, it begins playing and accepts the request; otherwise it rejects the request. `dataOut` is reserved.

  There are special values for `dataIn`: If the value is $0000xxxx, it's not a sound handle. $00000001 is a request to play Silence, and $00000002 is a request to play the Standard Beep.

- $8000 (sent by name to "Apple~SoundCP~")
- $8001 (sent by name to "Apple~SoundCP~")
- $8003 (sent by name to "Apple~SoundCP~")
- $8100 (sent by name to "Apple~SoundCP~")

  These are used internally for the Sound control panel to communicate with its request procedure; nobody else should call them. The request procedure stays in RAM all the time, but the main Control Panel code does not, so they can't just communicate with direct calls or link-time shared globals.

Sound Chains into `BELLVECTOR`

For remapping the System Beep, Sound chains into `BELLVECTOR`, using `GetVector($1B)` and `SetVector($1B)`. Entry and exit is in 8-bit native mode. On exit, B, D, and X are always preserved, and Y is always zero (as required by the *Apple IIGS Firmware Reference*).

If the System Beep has been reassigned with Sound, then it plays the remapped sound and returns with the carry clear.

If System Beep has not been reassigned, it passes control to the previous value of `BELLVECTOR`.

Battery RAM locations

- Pitch: location $1F (range 0 to 14).
- Volume: location $1E (range 0 to 14).
- Visual indication of sounds: bit 0 of location $5E (the bit is set when visual indication of sounds is **off**).

Limitations

- Sound Control Panel 2.0 accepts `srqPlayrSoundSample` only if the Sound Tools are available and a page of bank zero space is allocatable.

  However, separate utilities can accept `srqPlayrSoundSample` to play sounds under difficult conditions.

Appendix A **Battery RAM Use Update**

This appendix contains new information about the use of Battery RAM. The complete information about battery RAM can be found in Volume 1, Chapter 14 of the *Apple IIGS Toolbox Reference*

New Uses

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.
Bits 4-3 is the menu item blink rate setting (0..3).

$5F Miscellaneous Toolbox Settings

Bits 7-6: Byte validity check (%10 if the byte has been initialized).
Bits 5-3: Reserved, should be zero.
Bit 2 is set for no QuickDraw scan line interrupts (`QDStartUp` does `SetIntUse(0)`).
Bit 1 is set for no `ShowBootInfo` icons.
Bit 0 is set to alphabetize desk accessory lists.

$60 Toolbox: `WaitUntil` scaling (see Miscellaneous Tools).

$61 Reserved for network medium selection.

$62 Specifies OS for network boot (1 for GS/OS, 2 for ProDOS 8).

Appendix B **Resource Types Update**

This appendix describes new resource types defined for System Software 6.0 and HyperCard IIGS 1.1. 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*

Additions to the System Resource File

This section describes many of the additions to the system resource file.

- These `rIcon` resources (type = $8001) have been added. 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.

- These `rCursor` resources (type = $8027) have been added.

  $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.

- These `rErrorString` resources (type = $8020) have been added for use by `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!

- One `rWindColor` (type = $8010) resource has been added:

  $07FF0001 Black and white lined pattern title bar.

- One `rVersion` (type = $8029) resource has been added:

  $00000001 Resource for the System Software version. The product name is "System"; string2 is "Copyright 1983-1992, Apple Computer, Inc."

New Resource Types

$802B rBundle

`rBundle` resources are used by Finder 6.0 to keep track of applications and their icons. See Chapter 34 for details on how this resource is used.

$00	*version*	**Word**—Version ($0000)
$02	*offset*	**Word**—Offset from beginning of `rBundle` resource to `DocList` structure
$04	— *iconID* —	**Long**—`rIcon` ID for the application
$08	— *bundleID* —	**Long**—`rBundle` ID for this `rBundle` resource
$0C	— *forFinder* —	**Long**—Used internally by the Finder; must be zero
$10	*docList*	**variable**—`DocList` structures

DocList structure

$00	*count*	**Word**—Number of `OneDoc` structures
$02	*OneDocs*	**variable**—Array of `OneDoc` structures

The `docList` structure is a count followed by that number of `OneDoc` structures. Each `OneDoc` structure defines a group of files that may be associated with the application containing this `rBundle` resource. For example, a text editor might have a particular file type and auxiliary type that it prefers for storing the text files, so there would be one `OneDoc` structure for that preferred file type. The word processor might also be able to read, and perhaps write, generic text files and program source files, so two more `OneDoc` structures would be added to tell the Finder the application can deal with those files. Finally, the editor might save a preferences file, and use a fourth `OneDoc` structure to tell the Finder what icon to use for the preferences file. In this case, `count` would be 4, and the four `OneDoc` structures would follow `count`.

OneDoc structure

$00	*OneDocSize*	**Word**—Size in bytes of this structure (varies)
$02	*offsetToFlags*	**Word**—Offset from beginning of `OneDoc` structure to `MatchFlags` field
$04	*numResults*	**Word**—Number of result field groups following; must be 4 or 5
$06	*result1*	**Word**—Bitmapped result field; see description
$08	— *pathID* —	**Long**—`rFinderPath` ID of the path needed to run the application
$0C	— *internal* —	**Long**—Used internally by the Finder; must be zero (`NIL`)
$10	— *iconID* —	**Long**—`rIcon` ID for the large icon for this document
$14	— *internal2* —	**Long**—Used internally by the Finder; must be zero
$18	— *smallIconID* —	**Long**—`rIcon` ID for the small icon for this document
$1C	— *internal3* —	**Long**—Used internally by the Finder; must be zero
$20	*docString*	**pString**—Document description
$xx	— *matchFlags* —	**Long**—Match field selector flags
$xx	…	**xx bytes**—Various match fields (see following descriptions)

Each `OneDoc` structure uses a series of tests that the Finder can apply to a file to see if the `OneDoc` structure applies to the file. The various tests are contained in the match field structures at the end of the `OneDoc` structure. If the file passes all of the tests, the Finder may use the information stored in the first part of the `OneDoc` structure, perhaps to pick out the small icon to use when displaying the file in a window.

Since more than one application could have a `OneDoc` structure that matches a particular file, there is no guarantee that the `OneDoc` structure you create will apply to a particular file.

`numResults` tells the Finder how many pieces of information you are defining for the files that match the `OneDoc` structure. When used with Finder 6.0, this value would be 4 or 5, but more values may be defined in the future. The four or five fields are called result fields, and referred to as result 1, result 2, and so forth.

Result 1 is the voting priority for the `OneDoc` structure. The Finder can use this field to decide which `OneDoc` structure to use when two or more applications have OneDoc structures that match a particular file. (In Finder 6.0, this field is ignored.) See Chapter 34 of this book for a discussion of voting. The various voting bits are:

Bit	Meaning
0	Launch this (that is if bit 0 = 0, then only display this icon).
4	(Voting priority 0.) Application for this `OneDoc` can open this file.
5	(Voting priority 1.) Application for this `OneDoc` can write this file (possibly in a different format than it was opened)
6	(Voting priority 2.) Application for this `OneDoc` can write to this file in the same format it was opened.
7	(Voting priority 3.) Application for this `OneDoc` is the owner of this type of file (for example, the application AppleWorks and the `OneDoc` for AppleWorks documents.)

All bits other than the ones shown are reserved and must be 0.

The `pathID` and `internal1` fields make up result 2. This result is the path for the application, and is filled in by the Finder.

The `iconID` and `internal2` fields make up result 3. This result defines the large icon for files that match the `OneDoc` structure. Assuming the Finder uses this `OneDoc` structure for a particular file (that is, this is the only matching `OneDoc` structure, or the Finder decides this `OneDoc` structure has priority over any others) it will use this large icon when displaying the file on the desktop or in a window that is displaying files by large icon. If you don't want to define a large icon, code `NIL` for `iconID`.

The `smallIconID` and `internal3` fields make up result 4. This result defines the small icon for files that match the `OneDoc` structure. Assuming the Finder uses this `OneDoc` structure for a particular file (that is, this is the only matching `OneDoc` structure, or the Finder decides this `OneDoc` structure has priority over any others) it will use this icon when displaying the file in a window that is displaying files by small icon. If you don't want to define a large icon, code `NIL` for `smallIconID`.

Result 5 (`docString`) is optional. It defines a string that will be used by the Finder — and other applications that use `OneDoc` structures — when the file type is described with a text string. If this string is not present, the best match available from the file type descriptor files is used. This string should only be present if it can do a better job describing this document than the file type descriptor files.

Note Use this string as a last resort! Many utilities display the file types from the file type descriptor files, and if you code a string here, the user will see at least two different file type strings, depending on the utility used to look at the file type.

A good use of this string would be to create a description of a preferences file that included the name of the application that uses the file. A bad use would be to substitute your own description for a file type because you don't happen to like the default description.

The result fields are followed by matchFlags, which is a bitmap telling the Finder which of the various possible match fields should be used. Each match field represents a distinct test that is applied to the file. For example, one match field can be used to check the file type of a file. If the file passes each of the tests, the Finder may apply this `OneDoc` structure to the file. If the file fails any one of the tests, the `OneDoc` structure would never be applied to the file.

Tip All fields have to match for the comparison to succeed. If you need to do "`or`" matching, you need more than one `OneDoc` structure (possibly with the same icon ID). If you do this, place any `OneDoc` next to each other sequentially in the `rBundle` so that the Finder spends less time when it needs to update the `Desktop` file.

MatchFlags bits:

Bit	Match Field
0	GS/OS file type
1	auxiliary type
2	file name
3	creation date/time
4	modification date/time
5	local access
6	network access
7	extended files
8	HFS-style file type
9	HFS-style creator type
10	`optionList` contents
11	total EOF (resource and data fork)
12-31	reserved for future use (use 0)

Example: $00000003 means match by file type and auxiliary type. Only the `matchFileType` and `matchAuxType` structures follow.

Example: $00000005 means match by file type and file name. The `matchFileType` and `matchFileName` structures follow, optionally separated by a `matchType` ID of zero ($0000).

Note This structure is upward compatible—if any bits are set that you don't recognize, you just ignore the fields after that point and skip to the next `OneDoc` structure. The size field lets you do that without knowing the size of any unknown fields.

A `MatchType` ID of zero ($0000) is reserved and is ignored by the Finder.

matchFileType structure:

$00	*matchTypeID*	**Word**—Match type ID for this structure; use 1
$02	*fileType*	**Word**—GS/OS file type

This structure tells which file types should be matched. Only one file type can be listed per structure.

Example: $00B3 means match GS/OS Application files
Example: $0000 means match file type $0000 **only**

matchAuxType structure:

$00	*matchTypeID*	**Word**—Match type ID for this structure; use 2
$02	— *auxMask* —	**Long**—Auxiliary type mask
$06	— *auxType* —	**Long**—Auxiliary type value

This structure selects the auxiliary file types that will be treated as a match. The `auxMask` parameter and the auxiliary type for a file are anded. If the result matches the `auxType` parameter, the file passes this test.

Example: $FFFFFFFF $00001234 means match auxiliary type $00001234 only
Example: $00008000 $00000000 means match auxiliary types with bit 15 OFF

matchFilename structure:

$00	*matchTypeID*	**Word**—Match type ID for this structure; use 3
$02	*fileName*	**pString**—File name

This parameter is used to match specific file names. The case of the names is not significant, but the characters are full eight bit characters, not normal seven bit ASCII characters. This allows for the possibility of extended characters (like ö) in file names.

The "*" character can be used as a wildcard. To match all file names, use a single wild card character of "*".

Example: $09 "Installer"
Example: $05 "*.TXT"

matchCreateDateTime structure:

$00	*matchTypeID*	**Word**—Match type ID for this structure; use 4
$02	*compareSpec*	**Word**—Bits used to allow or disallow specific comparisons
$04	*timeRec*	**8 bytes**—Date/time value (see discussion for details)

Bits 15-11 of `compareSpec` are reserved; use 0 for these bits.

Bits 10-8 define the type of comparison to perform. In each case, the time in the structure is being compared to the time for the disk file, so less than would give a result of `TRUE` if the time in the structure is less than the time for the disk file.

Bits 10-8	Type of Comparison
000	less than
001	equal
010	greater than
011	reserved
100	greater than or equal
101	not equal
110	less than or equal
111	reserved

For the purpose of this comparison, the date and time are split into eight different quantities. Each of the eight least significant eight bits of `compareSpec` tells whether one of these quantities should be selected. The date and time fields themselves make up the `TimeRec` parameter, with one of the date and time fields in each byte. The bits from `compareSpec`, and the bytes each bit matches in `TimeRec`, are:

compareSpec Bit	Date/Time Byte	Use
bit 0	byte 0	seconds (0..59)
bit 1	byte 1	minutes (0..59)
bit 2	byte 2	hour (0..23)
bit 3	byte 3	year (0..255; these values are mapped to 1900..2155)
bit 4	byte 4	day (0..30)
bit 5	byte 5	month (0..11; 0 mapping to January..December)
bit 6	byte 6	unused; set both the bit and byte to 0
bit 7	byte 7	weekday (1..7, mapping to Sunday..Saturday)

The date and time field uses the same format as the Miscellaneous tool call `ReadTimeHex`.

Example: $0188, $00 $00 $00 $5A $00 $00 $00 $03 matches any Tuesday in 1990 (`compareSpec` $0188 means check bytes 3 and 7 for equality).

matchModDateTime structure:

$00	*matchTypeID*	**Word**—Match type ID for this structure; use 5
$02	*compareSpec*	**Word**—Bits used to allow or disallow specific comparisons
$04	*timeRec*	**8 bytes**—Date/time value

See `matchCreateDateTime`, above, for a detailed description of the use and format of the `compareSpec` and `timeRec` parameters.

Example: $0188, $00 $00 $00 $5A $00 $00 $00 $03 matches any Tuesday in 1990 (`compareSpec` $0188 means check bytes 3 and 7 for equality).

matchLocalAccess structure:

$00	*matchTypeID*	**Word**—Match type ID for this structure; use 6
$02	*mask*	**Word**—Local access word mask
$04	*value*	**Word**—Local access word value

This structure allows selection of local files with particular access bits set. `mask` is anded with the access bits for a file, then the result is compared to `value`. If the values are equal, the match succeeds.

There is a separate structure for checking access bits for a file on a network volume.

Example: $0004 ($0004 selects files with their invisible bit set)

Example: $0080 ($0000 selects files with their destroy bit clear)

matchNetworkAccess structure:

$00	*matchTypeID*	**Word**—Match type ID for this structure; use 7
$02	— *mask* —	**Long**—Network access long mask
$06	— *value* —	**Long**—Network access long value

This structure allows selection of network files with particular access bits set. `mask` is anded with the access bits for a file, then the result is compared to `value`. If the values are equal, the file is selected.

There is a separate structure for checking access bits for a file on a local volume.

Example:	$00000404 (mask) $00000404 (value)	Selects folders to which both you and everyone can make changes.
Example:	$02020206 (mask) $02020006 (value)	Selects folders to which you can make changes, and folders in which the owner or the group (but not everyone) can make changes.
Example:	$00000084 (mask) $00000000 (value)	Selects folders which you do not own and to which you cannot make changes.

matchExtended structure:

$00	*matchTypeID*	**Word**—Match type ID for this structure; use 8
$02	*mask*	**Word**—Mask
$04	*compareValue*	**Word**—Comparison value

This structure determines whether a file is stored using storage type 5 – in other words, whether the file has a resource fork. If `compareValue` is $8000, the test succeeds if the file has storage type 5. If `compareValue` is $0000, the test succeeds if the file has a storage type other than 5.

The Finder creates the word value being tested. Only bit 15 is defined in Finder 6.0 (it is set when the file's storage type is 5). Bits 14-0 are reserved and must be masked out of the comparison, so `mask` should always be $8000.

Example: $8000
$8000 extended file (files with non-zero length resource forks)

matchHFSFileType structure:

$00	*matchTypeID*	**Word**—Match type ID for this structure; use 9
$02	*fileType*	**Long**—4-character type to match, in Mac order

All files on HFS and AppleShare volumes have four-character HFS-style file types. Extended ProDOS files can also have four-character file types. This structure lets you mask out any files that don't have a specific HFS file type.

All four characters are eight-bit characters, not seven-bit ASCII characters. Case is significant. Also, note that the characters are ordered for the native byte order for 68000 numbers, not the 65816 byte order normally used on the Apple IIGS. That means the string 'ABCD' is stored with the 'A' in memory first, 'B' next, and so on.

Example: $4D $41 $43 $41 matches file type ‘MACA’

matchHFSCreator structure:

$00	*matchTypeID*	**Word**—Match type ID for this structure; use 10
$02	— *fileType* —	**Long**—4-character creator type to match, in Macintosh order

All files on HFS and AppleShare volumes have four-character HFS-style creator types. Extended ProDOS files can also have four-character creator types.

All four characters are eight-bit characters, not seven-bit ASCII characters. Case is significant. Also, note that the characters are ordered for the native byte order for 68000 numbers, not the 65816 byte order normally used on the Apple IIGS. That means the string ‘ABCD’ is stored with the ‘A’ in memory first, ‘B’ next, and so on.

Example: $4D $41 $43 $41 matches creator type ‘MACA’

matchOptionList structure:

$00	*matchTypeID*	**Word**—Match type ID for this structure; use 11
$02	*count*	**Word**—Number of `FSTGroupOption` structures following
$04	*groupOptions*	**variable**—`FSTGroupOption` structures

FSTGroupOption structure:

$00	*count*	**Word**—Number of fileSystemID entries
$02	*fileSystemID*	**Word(s)**—`count` file system ID entries
$xx	*offset*	**Word**—Offset into `optionList`
$xx	— *mask* —	**Long**—Field mask
$xx	— *value* —	**Long**—Field value
$xx	*compareSpec*	**Word**—Comparison specification

`compareSpec` is broken down into several fields. Bits 15-11 and bits 3-0 are reserved and should be 0. Each of bits 3-0 enables one of four possible bytes of the `mask` and `value` parameters to be included in the comparison. Each 0 bit here forces a byte in the value to be 0 before the comparison. Bits 10-8 tell what type of comparison to do. In each case, the value for the disk file is being compared to the value for the structure, so less than would give a result of `TRUE` if the value for the file is less than the value in this structure.

Bits 10-8	Type of Comparison
000	less than
001	equal
010	greater than
011	reserved
100	greater than or equal
101	not equal
110	less than or equal
111	reserved

`matchOptionList` example:

$0002	Two `FSTGroupOption` structures follow (both must match for the comparison to succeed)

first `FSTGroupOption` structure:

$0003	Three file systems:
$1234	first `FileSysID`
$3456	second `FileSysID`
$4567	third `FileSysID`
$0010	offset into `optionList` buffer
$0000FFFF	mask for Long fetched from `optionList`
$00004321	comparison value for the masked Long
$050F	compareSpec (“not equal to”, low 4 bytes)

second `FSTGroupOption` structure:

$0001	Only one file system for this check
$1234	first `FileSysID`
$0020	offset into `optionList` buffer
$FFFFFFFF	mask for Long fetched from `optionList`
$12345678	comparison value for the masked Long
$040F	compareSpec (“not less than”, low 4 bytes)

matchEOF structure:

$00	*matchTypeID*	**Word**—Match type ID for this structure; use 12
$02	*compareSpec*	**Word**—Comparison specification
$04	— *EOF* —	**Long**—EOF of compare against; total of data and resource forks

`compareSpec` is broken down into several fields. Bits 15-11 and bits 3-0 are reserved and should be 0. Each of bits 3-0 enables one of four possible bytes of the `EOF` parameter to be included in the comparison. Each 0 bit here forces a byte in the value to be 0 before the comparison. Bits 10-8 tell what type of comparison to do. In each case, the value in the structure is being compared to the value for the disk file, so less than would give a result of `TRUE` if the EOF in the structure is less than the EOF for the disk file.

Bits 10-8	Type of Comparison
000	less than
001	equal
010	greater than
011	reserved
100	greater than or equal
101	not equal
110	less than or equal
111	reserved

$802A rComment

An `rComment` resource consists of unformatted text. Any file with an Apple IIGS format resource fork can have an `rComment` resource with a resource ID of 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. The various uses for this resource are detailed in Apple IIGS Technical Note #76.

$801B rFileType

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

$802C rFinderPath

$00	*version*	**Word**—Version ($0000)
$02	*offset*	**Word**—Offset to pathname
$04	*pCount*	**Word**—Parameter count; the minimum value is 0
$06	— *resourceID* —	***Long**—`rVersion` resource ID of the version resource
$xx	— *internal* —	**Long**—Used internally by the Finder; must be $00000000
$xx	*pathName*	**GS/OS String**—application pathname

`resourceID` is an optional parameter; it is missing if `pCount` is 0.

The `rFinderPath` resource is used by the Finder to locate an application. This resource is normally created and maintained by the Finder, and should not appear in an application. For details on how the `rFinderPath` resource is used, see Chapter 34.

$8028 rItemStruct

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

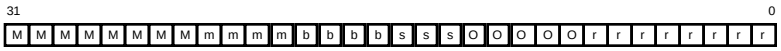
$8029 rVersion

The `rVersion` resource is used to record information about the application. An `rVersion` resource with an ID of 1 is accessed by the Finder, which can display the information contained in this resource.

The format for the `rVersion` resource is:

$00	*version*	**Long**—Version number
$04	*country*	**Word**—Country code; use 0 for U.S.
$06	*name*	**pString**—Name of the product (may be empty)
$xx	*moreInfo*	**pString**—Additional information, like copyrights (may be empty)

`version` The version number is an encoded long word, broken down into six fields.



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)

An important property of the version number is that you can compare two version numbers as unsigned long values. The version number for the more recent release is always greater than the version number for earlier releases.

See Apple IIGS Technical Note #100 for a complete description of version numbers, how the fields are used, and the various other formats that are used for version numbers throughout the system.

Examples:

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

`country` This field is reserved for a country code. Use 0 for the U.S.

`name` This string gives the name of the product. It can be blank.

`moreInfo` This string gives more information about a product, such as copyright information. It can be blank.

Note Applications are differentiated primarily by their Country code and Application name, and secondarily by their version number.

To keep its `Desktop` database up to date, the Finder checks the `rVersion` of an application it is launching against the `rVersion`

resources of the applications on which the Finder has gathered information.

If the Apple IIGS four-byte version supersedes the last known version of an application (matched by the "Application name" field) then the old `rBundle` resource from the old Application and its associated icons are removed from the `Desktop` database and the new `rBundle` resource and icons from the newer version of the application are incorporated into the `Desktop` database.

If the version of the application is below that of an application that the Finder already "knows," then any `rBundle` and `rIcon` resources in the application are **not** added to the Finder's `Desktop` database.

For the above reasons, if you want the Finder to launch the latest version of your application, the product name should not change throughout the lifetime of the product. Also, the version number of your product should steadily increase and never decrease.

Appendix C **Extended Character Set**

The table below shows the extended character set used by the Apple IIGS toolbox. The same character set is used by the Macintosh.

Not all characters appear in all fonts, and some additional or different characters may appear in some fonts. In fact, some special purpose fonts do not include the standard alphabet. In other words, this standard character set supplies a template, but there are many good reasons to deviate from this standard, and many character sets will do so.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			space	0	@	P	`	p	Ä	ê	†	∞	¿	–		
1			!	1	A	Q	a	q	Å	ë	°	±	¡	—		
2			"	2	B	R	b	r	Ç	í	¢	≤	¬	“		
3			#	3	C	S	c	s	É	ì	£	≥	√	”		
4			$	4	D	T	d	t	Ñ	î	§	¥	ƒ	‘		
5			%	5	E	U	e	u	Ö	ï	•	µ	≈	’		
6			&	6	F	V	f	v	Ü	ñ	¶	∂	Δ	÷		
7			'	7	G	W	g	w	á	ó	ß	∑	«	◊		
8			(	8	H	X	h	x	à	ò	®	∏	»	ÿ		
9			)	9	I	Y	i	y	â	ô	©	π	…			
A			*	:	J	Z	j	z	ä	ö	™	∫	space			
B			+	;	K	[	k	{	ã	õ	´	ª	À			
C			,	<	L	\	l	|	å	ú	¨	º	Ã			
D			-	=	M	]	m	}	ç	ù	≠	Ω	Õ			
E			.	>	N	^	n	~	é	û	Æ	æ	Œ			
F			/	?	O	_	o		è	ü	Ø	ø	œ			

- The characters from the space ($20) to the tilde ($7E) are all standard printing ASCII characters.
- While they have standard definitions, the characters $11..$14, $AD, $B0..$B3, $B5..$BA, $BD, $C2..$C6 and $D6 tend to be rare in most fonts.
- Character $CA is the non-breaking space.

Appendix D **Writing Your Own Tool Set Update**

This appendix contains new information about writing your own tool set. The complete information about writing your own tool set is in Volume 2, Appendix A of the *Apple IIGS Toolbox Reference*. Several technical notes also deal with this topic. See, for example, Apple IIGS Technical Note #73, "Using User Tool Sets".

Clarifications

- There are two vectors a system tool or user tool can jump to when it exits.

  ```
  ToBusyStrip $E10180
  ToStrip     $E10184
  ```

  In both cases, the `X` register should be set to the error code (0 for no error) and `Y` should be the number of bytes of parameter to strip from the stack.

  `ToStrip` shifts the 6 bytes of the `RTL` addresses up `Y` bytes, sets up `A` and the carry flag appropriately, and returns to the tool's caller.

  `ToBusyStrip` does the same things as `ToStrip`, but also decrements the system busy flag.

Appendix E **Toolbox Concordance**

Alphabetic Listing of Tool Calls by Tool

This section lists all of the tool and GS/OS calls documented in in *Apple IIGS Toolbox Reference*, volumes 1 through 3; *Apple IIGS GS/OS Reference*; and this book, *Programmer's Reference for System 6.0*. The calls are broken down first by tool, then listed alphabetically for the tool.

Some tool calls have been documented in more than one place. This section lists the different places where you can find information about each tool call. This includes the original description of the call; updates to the description that were major enough to require correcting or adding to the description in *Apple IIGS Toolbox Reference: Volume 3* or *Programmer's Reference for System 6.0*; and any mention of the call in the errors and corrections sections in *Apple IIGS Toolbox Reference: Volume 3* or *Programmer's Reference for System 6.0*. This section does not index the various other places where a tool call might be mentioned, such as in the description of another tool call or in the front matter at the start of the various chapters.

The references are listed by volume and page number. These abbreviations are used for the volume names:

I	*Apple IIGS Toolbox Reference: Volume 1*
II	*Apple IIGS Toolbox Reference: Volume 2*
III	*Apple IIGS Toolbox Reference: Volume 3*
6.0	*Programmer's Reference for System 6.0*

Apple Desktop Bus Tool Set

`AbsOn`	I-3-13
`AbsOff`	I-3-13
`ADBBootInit`	I-3-10
`ADBReset`	I-3-12
`ADBShutDown`	I-3-11
`ADBStartUp`	I-3-10
`ADBStatus`	I-3-12
`ADBVersion`	I-3-11, 6.0-1
`AsyncADBReceive`	I-3-14, III-26-3
`ClearSQRTable`	I-3-15
`GetAbsScale`	I-3-15
`ReadAbs`	I-3-16
`ReadKeyMicroData`	I-3-17, III-26-2
`ReadKeyMicroMemory`	I-3-18
`SendInfo`	I-3-19
`SetAbsScale`	I-3-23
`SRQPoll`	I-3-25
`SRQRemove`	I-3-26
`SyncADBReceive`	I-3-27

Audio Compression and Expansion Tool Set

`ACEBootInit`	III-27-6
`ACECompBegin`	III-27-13
`ACECompress`	III-27-14
`ACEExpand`	III-27-16
`ACEExpBegin`	III-27-18
`ACEInfo`	III-27-12
`ACEReset`	III-27-10
`ACEShutDown`	III-27-8
`ACEStartUp`	III-27-7
`ACEStatus`	III-27-11
`ACEVersion`	III-27-9
`GetACEExpState`	6.0-4
`SetACEExpState`	6.0-5

Control Manager

`CallCtlDefProc`	III-28-22
`CMLoadResource`	III-28-24
`CMReleaseResource`	III-28-25
`CtlBootInit`	I-4-41
`CtlNewRes`	I-4-45, 6.0-7
`CtlReset`	I-4-44
`CtlShutDown`	I-4-43
`CtlStartUp`	I-4-42
`CtlStatus`	I-4-44
`CtlVersion`	I-4-43
`DisposeControl`	I-4-45

Desk Manager

Dialog Manager

Event Manager

Font Manager

Integer Math Tool Set

Memory Manager

Menu Manager

MIDI Synth Tool Set

MIDI Tool Set

Miscellaneous Tool Set

Note Sequencer

Note Synthesizer

Print Manager

QuickDraw II

QuickDraw II Auxiliary

Resource Manager

SANE Tool Set

Scheduler

Text Tool Set

Tool Locator

Video Overlay Tool Set

Window Manager

Tool and GS/OS Error Codes

This section is a collection of all of the tool errors documented in *Apple IIGS Toolbox Reference*, volumes 1 to 3; *Apple IIGS GS/OS Reference*; and the System 6.0 ERSs that provided the source material for this book. The errors are listed in numerical order, which also happens to break the errors up into groups by tool number, since the most significant byte of each error code is the tool number for the tool that flags the error.

GS/OS errors are listed here as if GS/OS is a tool. Technically it isn't, but GS/OS errors are sometimes reported by tools that make calls to GS/OS.

System Failure Errors

$0001	pdosUnClmdIntErr	Unclaimed interrupt
$0004	divByZeroErr	Division by zero
$000A	pdosVCBErr	Volume control block is not useable
$000B	pdosFCBErr	File control block is not useable
$000C	pdosBlk0Err	Block zero allocated illegally
$000D	pdosIntShdwErr	Interrupt with I/O shadowing off
$0017	sPackage0Err	Can't load a package
$0018	package1Err	Can't load a package
$0019	package2Err	Can't load a package
$001A	package3Err	Can't load a package
$001B	package4Err	Can't load a package
$001C	package5Err	Can't load a package
$001D	package6Err	Can't load a package
$001E	package7Err	Can't load a package
$0020	package8Err	Can't load a package
$0021	package9Err	Can't load a package
$0022	package10Err	Can't load a package
$0023	package11Err	Can't load a package
$0024	package12Err	Can't load a package
$0025	putOfMemErr	Out of memory
$0026	segLoader2Err	Segment loader error
$0027	fMapTrshdErr	File map destroyed
$0028	stkOvrFlwErr	Stack overflow
$0030	psInstDiskErr	Insert disk alert
$0032-$0053		Memory Manager errors

GS/OS

$01	badSystemCall	Bad GS/OS call number
$04	invalidPcount	The parameter count is out of range
$07	gsosActive	GS/OS is busy
$10	devNotFound	Device not found
$11	invalidDevNum	Invalid device number
$20	drvrBadReq	Invalid request
$21	drvrBadCode	Invalid control or status code
$22	drvrBadParm	Bad call parameter
$23	drvrNotOpen	Character device not open
$24	drvrPriorOpen	Character device is already open
$25	irqTableFull	Interrupt table full
$26	drvrNoResrc	Resources are not available

$27	drvrIOError	I/O error
$28	drvrNoDevice	No device connected
$29	drvrBusy	Driver is busy
$2B	drvrWrtProt	Device is write protected
$2C	drvrBadCount	Invalid byte count
$2D	drvrBadBlock	Invalid block address
$2E	drvrDiskSwitch	The disk has been switched
$2F	drvrOffLine	Device off line or no media present
$40	badPathSyntax	Invalid pathname syntax
$42	tooManyFilesOpen	The AppleShare file server limit of open files has been reached
$43	invalidRefNum	Invalid reference number
$44	pathNotFound	Subdirectory does not exist
$45	volNotFound	Volume not found
$46	fileNotFound	File not found
$47	dupPathname	Create or rename attempted with a name that already exists
$48	volumeFull	The volume is full
$49	volDirFull	The volume directory is full
$4A	badFileFormat	Version error (incompatible file type)
$4B	badStoreType	Unsupported or incorrect storage type
$4C	eofEncountered	End of file encountered
$4D	outOfRange	Position out of range
$4E	invalidAccess	Access not allowed
$4F	buffTooSmall	Buffer too small
$50	fileBusy	File is already open
$51	dirError	Directory error
$52	unknownVol	Unknown volume type
$53	paramRangeError	Parameter out of range
$54	outOfMem	Out of memory
$57	dupVolume	Duplicate volume name
$58	notBlockDev	Not a block device
$59	invalidLevel	Invalid file level
$5A	damagedBitMap	Block number too large
$5B	badPathNames	Invalid pathnames for ChangePath
$5C	notSystemFile	Not an executable file
$5D	osUnsupported	Operating system not supported
$5F	stackOverflow	Too many applications on stack
$60	dataUnavail	Data unavailable
$61	endOfDir	End of directory has been reached
$62	invalidClass	Invalid FST call class
$63	resForkNotFound	The file does not contain a required resource
$64	invalidFSTID	Invalid FST number
$65	invalidFSTop	FST does not handle this type of call
$66	fstCaution	FST handled call, but result is weird
$67	devNameErr	A device exists with the same name as the replacement name
$68	devListFull	Device list is full
$69	supListFull	Supervisor list is full
$6A	fstError	Generic FST error
$70	resExistsErr	Cannot expand file; resource already exists
$71	resAddErr	Cannot add a resource fork to this kind of file
$7E	unknownUser	Unknown file server user
$7F	unknownGroup	Unknown file server group
$88	networkError	Generic network error

Tool Locator

$0001	toolNotFoundErr	The tool was not found
$0002	funcNotFoundErr	The tool function was not found
$0103	TLBadRecFlag	The StartStop record is invalid
$0104	TLCantLoad	A tool cannot be loaded
$0110	toolVersionErr	The requested minimum tool version was not available
$0111	messNotFoundErr	The specified message was not found
$0112	messageOvfl	No message numbers are available
$0113	nameTooLong	The message name is too long
$0120	reqNotAccepted	Nobody accepted the request
$0121	srqDuplicateName	The name has already been used
$0122	invalidSendRequest	Bad combination of reqCode and target

Memory Manager

$0201	memErr	Unable to allocate memory
$0202	emptyErr	Illegal operation on an empty handle
$0203	notEmptyErr	Illegal operation on a handle that is not empty
$0204	lockErr	Illegal operation on a locked or immovable block
$0205	purgeErr	Attempt to purge an unpurgeable block
$0206	handleErr	Invalid handle
$0207	idErr	Invalid user ID
$0208	attrErr	Illegal operation for the specified attributes

Miscellaneous Tool Set

$0301	badInputErr	Bad input parameter
$0302	noDevParamErr	No device for the input parameter
$0303	taskInstlErr	Specified task is already in the heartbeat queue
$0304	noSigTaskErr	No signature detected in the task header
$0305	queueDmgdErr	Damaged heartbeat queue
$0306	taskNtFdErr	Specified task is not in the queue
$0307	firmTaskErr	Unsuccessful firmware task
$0308	hbQueueBadErr	Damaged heartbeat queue
$0309	unCnctDevErr	Dispatch attempted to an unconnected device
$030B	idTagNtAvlErr	No ID tag is available
$0380	notInList	The specified routine was not found in the queue
$0381	invalidTag	The correct signature value was not found in the header
$0382	alreadyInQueue	Specified element already in queue
$0390	badTimeVerb	Invalid convVerb value
$0391	badTimeData	Invalid date or time to be converted
$034F	mtBufferTooSmall	The buffer is too small

QuickDraw II

$0401	alreadyInitiallized	QuickDraw II is already initialized
$0402	cannotReset	Never used
$0403	notInitialized	QuickDraw II is not initialized
$0410	screenReserved	The screen memory is reserved
$0411	badRect	Invalid rectangle
$0420	notEqualChunkiness	Chunkiness is not equal
$0430	rgnAlreadyOpen	Region is already open
$0431	rgnNotOpen	No region is open

$0432	rgnScanOverflow	Region scan overflow
$0433	rgnFull	Region is full
$0440	polyAlreadyOpen	Polygon is already open
$0441	polyNotOpen	No polygon is open
$0442	polyTooBig	The polygon is too big
$0450	badTableNum	Invalid color table number
$0451	badColorNum	Invalid color number
$0452	badScanLine	Invalid scan line number
$04FF		Not implemented

Desk Manager

$0510	daNotFound	Specified desk accessory is not available
$0511	notSysWindow	The window parameter is not a pointer to a system window owned by an NDA
$0520	deskBadSelector	Selector out of range

Event Manager

$0601	emDupStrtUpErr	The Event Manager has already been started
$0602	emResetErr	Can't reset the Event Manager
$0603	emNotActErr	The Event Manager is not active
$0604	emBadEvtCodeErr	The event code is greater than 15
$0605	emBadBttnNoErr	The button number given was not 0 or 1
$0606	emQSiz2LrgErr	The size of the event queue is larger than 3639
$0607	emNoMemQueueErr	Insufficient memory for the event queue
$0681	emBadEvtQErr	The event queue is damaged
$0682	emBadQHndlErr	Queue handle damaged

Sound Tool Set

$0810	noDOCFndErr	The DOC or RAM was not found
$0811	docAddrRngErr	DOC address range error
$0812	noSAddrInitErr	The Sound Tool Set is not active
$0813	invalGenNumErr	Invalid generator number
$0814	synthModeErr	Synthesizer mode error
$0815	genBusyErr	The generator is already in use
$0817	mstrIRQNotAssgnErr	Master IRQ not assigned
$0818	sndAlreadyStrtErr	The Sound Tool Set is already started
$08FF	unclaimedSndIntErr	Unclaimed sound interrupt error

Apple Desktop Bus Tool Set

$0910	cmndIncomplete	Command not completed
$0911	cantSync	Can't synchronize with the system
$0982	adbBusy	ADB busy (command pending)
$0983	devNotAtAddr	Device not at present address
$0984	sqrListFull	SQR list is full

Integer Math Tool Set

$0B01	imBadInptParam	Bad input parameter
$0B02	imIllegalChar	Illegal character in the string
$0B03	imOverflow	Integer or longint overflow

$0B04	imStrOverflow	String overflow

Text Tool Set

$0C01	badDevType	Illegal device type
$0C02	badDevNum	Illegal device number
$0C03	badMode	Illegal operation
$0C04	unDefHW	Undefined hardware error
$0C05	lostDev	Lost device; device is no longer on line
$0C06	lostFile	File is not longer available
$0C07	badTitle	Illegal file name
$0C08	noRoom	Insufficient space on the specified diskette
$0C09	noDevice	Specified volume is not on line
$0C0A	noFile	Specified file is not in the directory given
$0C0B	dupFile	Duplicate file
$0C0C	notClosed	Attempt to open a file that is already open
$0C0D	notOpen	Attempt to access a closed file
$0C0E	badFormat	Error reading real or integer number
$0C0F	ringBuffOFlo	Ring buffer overflow
$0C10	writeProtected	The specified disk is write protected
$0C40	devErr	The device did not complete a read or write

Window Manager

$0E01	paramLenErr	The first word of the parameter list is the wrong size
$0E02	allocateErr	Unable to allocate the window record
$0E03	taskMaskErr	Some reserved bits were not clear in the wmTaskMask field of the wmTaskRec record
$0E04	compileTooLarge	Compiled text is larger than 64K

Menu Manager

$0F03	menuNoStruct	Returned if bit 10 of itemFlag is not set

Control Manager

$1001	wmNotStartedUp	The Window Manager is not initialized
$1002	cmNotInitialized	The Control Manager has not been started
$1003	noCtlInList	The control is not in the window list
$1004	noCtlError	No controls in the window
$1005	noExtendedCtlError	No extended controls in the window
$1006	noCtlTargetError	No extended control is currently the target control
$1007	notExtendedCtlError	The action is valid only for extended controls
$1008	canNotBeTargetError	The specified control cannot be made the target control
$1009	noSuchIDError	The specified control ID cannot be found
$100A	tooFewParmsError	Too few parameters were specified
$100B	noCtlToBeTargetError	No control could be made the target control
$100C	noFrontWindowError	There is no front window

Loader

$1101		Entry not found
$1102		OMF version error
$1103		Pathname error

$1104		The file is not a load file
$1107		File version error
$1108		User ID error
$1109		Segment number out of sequence
$110A		Illegal load record found
$110B		Load segment is foreign

QuickDraw II Auxiliary

$1210	picEmpty	Picture is empty
$1211	badRectSize	The height or width is negative, the destination rect is not the same size as the source rect, or the source or destination rect is not within its boundary
$1212	destModeError	The destMode portion of resMode is invalid
$121F	badPictureOpcode	Bad picture opcode
$1221	badRect	Bad rectangle
$1222	badMode	Bad mode
$1230	badGetSysIconInput	No icon is available for the given input

Print Manager

$1301	missingDriver	Specified driver is not in the drivers folder of the system folder
$1302	portNotOn	The specified port is not selected in the Control Panel
$1303	noPrintRecord	No print record was specified
$1304	badLaserPrep	The version of LaserPrep in the drivers folder is not compatible with this version of the Print Manager
$1305	badLPFile	The version of LaserPrep in the drivers folder is not compatible with this version of the Print Manager
$1306	papConnNotOpen	Connection can't be established with the LaserWriter
$1307	papReadWriteErr	Read-write error on the LaserWriter
$1308	ptrConnFailed	Connection can't be established with the ImageWriter
$1309	badLoadParam	The specified parameter is invalid
$130A	callNotSupported	Tool call is not supported by the current version of the driver
$1321	startUpAlreadyMode	LLDStartUp call already made
$1322	invalidCtlVal	Invalid control value specified

LineEdit Tool Set

$1401	leDupStrtUpErr	The LineEdit Tool Set has already been started
$1402	leResetError	Can't reset LineEdit
$1403	leNotActiveErr	The LineEdit Tool Set has not been started
$1404	leScrapErr	The desk scrap is too big to copy

Dialog Manager

$150A	badItemType	Inappropriate item type
$150B	newItemFailed	Item creation failed
$150C	itemNotFound	No such item
$150D	notModalDialog	The frontmost window is not a modal dialog

Scrap Manager

$1610	basCrapType	No scrap of this type

Standard File Operations Tool Set

$1701	badPromptDesc	Invalid promptRefDesc value
$1702	badOrigNameDesc	Invalid origNameRefDesc value
$1704	badReplyNameDesc	Invalid nameRefDesc value in the reply record
$1705	badReplyPathDesc	Invalid pathRefDesc value in the reply record
$1706	badCall	SFPGetFile, SFPGetFile2 and SFPMultiGet2 are not active
$17FF	sfNotStarted	Standard File is not active

Note Synthesizer

$1901	nsAlreadyInit	The Note Synthesizer has already been started
$1902	nsSndNotInit	The Sound Tool Set has not been started
$1921	nsNotAvail	No generators are available
$1922	nsBadGenNum	Invalid generator number
$1923	nsNotInit	The Note Synthesizer has not been started
$1924	nsGenAlreadyOn	The specified note is already being played
$1925	soundWrongVer	The version of the Sound Tool Set is not compatible with this version of the Note Synthesizer

Note Sequencer

$1A00	noRoomMidiErr	The Note Sequencer is already tracking 32 notes; there is no room for a MIDI NoteOn
$1A01	noCommandErr	The current seqItem is not valid in this context
$1A02	noRoomErr	The sequence is nested more than 12 levels deep
$1A03	startedErr	The Note Sequencer is already started
$1A04	noNoteErr	Can't find the note for a NoteOff command
$1A05	noStartErr	The Note Sequencer was not started
$1A06	instBndsErr	The specified instrument is outside of the bounds of the current instrument table
$1A07	nsWrongVer	The version of the Note Synthesizer is incompatible with the Note Sequencer

Font Manager

$1B01	fmDupStartUpErr	The Font Manager has already been started
$1B02	fmResetErr	Can't reset the Font Manager
$1B03	fmNotActiveErr	The Font Manager has not been started
$1B04	fmFamNotFndErr	Family not found
$1B05	fmFontNtFndErr	Font not found
$1B06	fmFontMemErr	Font not in memory
$1B07	fmSysFontErr	System font cannot be purged
$1B08	fmBadFamNumErr	Illegal family number
$1B09	fmBadSizeErr	Illegal font size
$1B0A	fmBadNameErr	Illegal name length
$1B0B	fmMenuErr	FixFontMenu never called
$1B0C	fmScaleSizeErr	Scaled size font exceeds limits
$1B0D	fmBadParmErr	Bad parameter to FMStartUp

List Manager

$1C02	listRejectEvent	The list control did not handle the event

Audio Compression and Expansion Tool Set

$1D01	aceIsActive	The ACE Tool Set has already been started
$1D02	aceBadDP	Requested direct page location is not valid
$1D03	aceNotActive	The ACE Tool Set has not been started
$1D04	aceNoSuchParam	Requested information type not supported
$1D05	aceBadMethod	Specified compression method is not supported
$1D06	aceBadSrc	Specified source is invalid
$1D07	aceBadDest	Specified destination is invalid
$1D08	aceDataOverlap	Specified source and destination areas overlap
$1DFF	aceNotImplemented	The requested function has not been implemented

Resource Manager

$1E01	resForkUsed	The resource fork is not empty
$1E02	resBadFormat	The resource fork is not correctly formatted
$1E03	resNoConverter	No converter routine for the resource type
$1E04	resNoCurFile	No current resource file
$1E05	resDupID	The specified resource ID is already in use
$1E06	resNotFound	The specified resource was not found
$1E07	resFileNotFound	The specified ID does not match an open file
$1E08	resBadAppID	The user ID was not found; the calling program has not issued a ResourceStartUp call
$1E09	resNoUniqueID	No more resource IDs are available
$1E0A	resIndexRange	Index is out of range; no resource was found
$1E0B	resSysIsOpen	The system resource file is already open
$1E0C	resHasChanged	The resource has been changed and has not been updated
$1E0D	resDiffConverter	Another converter is already logged in for this resource type
$1E0E	resDiskFull	Volume full
$1E10	resNameNotFound	The named resource was not found
$1E11	resBadNameVers	Bad version in rResName resource
$1E12	resDupStartUp	Already started with this ID
$1E13	resInvalidTypeOrID	The resource type or ID was not valid

MIDI Tool Set

$2000	miStartUpErr	The MIDI Tool Set has not been started
$2001	miPacketErr	Incorrect packet length received
$2002	miArrayErr	Array was an invalid size
$2003	miFullBufErr	MIDI data discarded because of buffer overflow
$2004	miToolsErr	Required tools inactive or incorrect version
$2005	miOutOffErr	MIDI output disabled
$2007	miNoBufErr	No buffer allocated
$2008	miDriverErr	Specified device driver invalid
$2009	miBadFreqErr	Unable to set MIDI clock to the specified frequency
$200A	miClockErr	MIDI clock wrapped to zero
$200B	miConflictErr	Two processes are competing for the MIDI interrupt
$200C	miNoDevErr	No device driver loaded
$2080	miDevNotAvail	MIDI interface not available
$2081	miDevSlotBusy	Specified slot not available in Control Panel
$2082	miDevBusy	MIDI interface already in use
$2083	miDevOverrun	MIDI interface overrun by input data; the interface is not being served quickly enough
$2084	miDevNoConnect	No connection to MIDI interface

$2085	miDevReadErr	Error reading MIDI data
$2086	miDevVersion	ROM version or machine type incompatible with device driver
$2087	miDevIntHndlr	Conflicting interrupt handler installed

TextEdit Tool Set

$2201	teAlreadyStarted	The TextEdit Tool Set has already been started
$2202	teNotStarted	The TextEdit Tool Set has not been started
$2203	teInvalidHandle	The teH does not refer to a valid TERecord
$2204	teInvalidDescriptor	Invalid descriptor value specified
$2205	teInvalidFlag	The specified flag word is invalid
$2206	teInvalidPCount	The specified parameter count is not valid
$2208	teBufferOverflow	The output buffer was too small to accept all data
$2209	teInvalidLine	The starting line value is greater than the number of lines in the text (can be interpreted as an end of file indication in some cases)
$220B	teInvalidParameter	A passed parameter was not valid
$220C	teInvalidTextBox2	The LETextBox2 format codes were inconsistent
$220D	teNeedsTools	The Font Manager was not started

MIDI Synth Tool Set

$2301	msAlreadyStarted	MIDI Synth already started
$2302	msNotStarted	MIDI Synth never started
$2303	msNoDPMem	Can’t get direct page memory
$2304	msNoMemBlock	Can’t get memory block
$2305	msNoMiscTool	Miscellaneous Tool Set not started
$2306	msNoSoundTool	Sound Tool Set not started
$2307	msGenInUse	Ensoniq generator in use
$2308	msBadPortNum	Illegal port number
$2309	msPortBusy	Port is busy
$230A	msParamRangeErr	Parameter range error
$230B	msMsgQueueFull	Message queue full
$230C	msRecBufFull	Rec buffer is full
$230D	msOutputDisabled	MIDI output disabled
$230E	msMessageError	Message error
$230F	msOutputBufFull	MIDI output buffer is full
$2310	msDriverNotStarted	Driver not started
$2311	msDriverAlreadySet	Driver already set
$2380	msDevNotAvail	The requested device is not available
$2381	msDevSlotBusy	The requested slot is already in use
$2382	msDevBusy	The requested device is already in use
$2383	msDevOverrun	Device overrun by incoming MIDI data
$2384	msDevNoConnect	No connection to MIDI
$2385	msDevReadErr	Framing error in received MIDI data
$2386	msDevVersion	RIM version is incompatible with device driver
$2387	msDevIntHndlr	Conflicting interrupt handler is installed

Media Control Tool Set

$2601	mcUnImp	Unimplemented for this device
$2602	mcBadSpeed	Invalid speed specified
$2603	mcBadUnitType	Invalid unit type specified
$2604	mcTimeOutErr	Timed out during device read

$2605	mcNotLoaded	No driver is currently loaded
$2606	mcBadAudio	Invalid audio value
$2607	mcDevRtnError	Device returned error (cannot perform the command)
$2608	mcUnRecStatus	Unrecognized status from the device
$2609	mcBadSelector	Invalid selector value specified
$260A	mcFunnyData	Funny data received (try again)
$260B	mcInvalidPort	Invalid port specified
$260C	mcOnlyOnce	Scans only once
$260D	mcNoResMgr	Resource Manager not active (must be loaded and started)
$260E	mcInvalidPort	Invalid port specified
$260F	mcWasShutDown	The tool set was already shut down
$2610	mcWasStarted	The tool was already started
$2611	mcBadChannel	An invalid media channel was specified
$2612	mcInvalidParam	An invalid parameter was specified
$2613	mcCallNotSupported	An invalid media control tool call was attempted

Finder Errors

$4201	fErrBadInput	Bad input value
$4202	fErrFailed	Could not complete request
$4203	fErrCancel	User cancelled operation
$4204	fErrDimmed	Menu was dimmed
$4205	fErrBusy	Not now, the Finder has a headache
$4206	fErrNotPrudent	Can’t add Finder’s resources to desktop file
$4207	fErrBadBundle	Unknown rBundle version, or rBundle damaged
$42FF	fErrNotImp	Request not implemented

E

F

G

H

I

N

O

P

Q

R

S

T

U

V

W