APPLE
PROGRAMMER'S
AND DEVELOPER'S
ASSOCIATION

Apple IIGS Toolbox Reference Update, Beta Draft

™

APDA# K2B005

Apple® II

Apple IIGS Toolbox Reference Update

APDA Draft
30 August, 1988

This document contains preliminary information
does not include:

- final editorial corrections
- final artwork
- final indexes
- a glossary
- final technical information

ISBN 0-201-xxxxx-x
ABCDEFGHIJ-DO-898

Contents

Preface

What's in the Update

This update to the *Apple IIGS Toolbox Reference* contains new material describing numerous changes to the Toolbox. There are four previously undocumented tool sets, 65 new tool calls, and numerous corrections and additions.

Organization

In order to make this update easy to use for those who are accustomed to the *Apple IIGS Toolbox Reference*, it is organized in a similar manner. The material is arranged in chapters that are devoted to individual tool sets or managers. Chapters that update information about tool sets that were documented in the first edition appear in the same order as the corresponding chapters in the original reference. The chapters documenting the four new tool sets appear in alphabetical order among the other chapters. There are several tool sets that are essentially unchanged; these tool sets do not appear in the update.

At the beginning of each chapter is a brief note about its contents; tool sets that were previously documented are so noted, and the chapters where their documentation can be found are mentioned.

Typographical conventions

This update largely obeys the typographical conventions of the *Apple IIGS Toolbox Reference*. New terms appear in **boldface** when they are introduced. Parameter and field names are given in *italics*. Sample code listings appear in the `Courier` typeface.

Indexes

The update contains two indexes. First, there is an index of calls that are new in this Update, arranged alphabetically. Next is an index listing all tool calls, both those in the *Apple IIGS Toolbox Reference*, and those documented in this update. This is included to make it easier to find a particular call's description, whether it is a new call, or one that was previously documented.

Chapter 1

Apple Desktop Bus Tool Set

This chapter contains new information about the Apple Desktop Bus Tool Set. The complete reference to the Tool Set is in Volume 1, Chapter 3 of the *Apple IIGS Toolbox Reference.*

Clarification

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

Bit 5:	0-Caps-lock down
	1-Caps-lock up
Bit 3:	0-Control key down
	1-Control key up
Bit 2:	0-Shift key down
	1-Shift key up
Bit 1:	0-Option key down
	1-Option key up
Bit 0:	0-Open-Apple key down
	1-Open-Apple key up

Chapter 2

Audio Compression and Expansion Tool Set

This chapter documents the features of the new Audio Compression and Expansion Tools Set (ACE Tools). This is a new tool set, not previously documented in the *Apple IIGS Toolbox Reference.*

About Audio Compression and Expansion

The Audio Compression and Expansion (ACE) tools are a set of utility routines that compress and decompress digital audio data. The tool set is designed to support a variety of methods of audio signal compression, but at present only one model is implemented. A knowledgeable programmer could, however, create alternative methods and use them with the ACE tools.

With the present model of compression supported by the ACE tools, you can choose either of two compression ratios. You can compress a digital audio signal to half its original size or to three-eighths its original size. The ratio used is determined by a parameter of the ACE call that does the compression.

The obvious advantage of compressing an audio signal is that it takes up less space on the disk and less time to transfer the data. A digital sample that is compressed to half its original size occupies only half the space and takes only half as long to transfer. Such a sample can load from the disk twice as fast as the uncompressed version , and is much more economical to upload to or download from a commercial computer network.

Uses of the ACE Tool Set

Software often includes sound effects, music, or speech. The problem with digitized sound is that requires considerable storage space. A faithful monophonic digitization of thirty seconds of an FM radio signal occupies nearly a megabyte of disk space. A user might be somewhat reluctant to use a program that occupies so much space to achieve sound effects. The ACE Tool Set provides you with the means to compress digitized sound signals to minimize this problem.

ACE presently supports Adaptive Differential Pulse Code Modulation (ADPCM). This compression model assumes that audio signals tend to be relatively smooth and continuous. If frequency and amplitude (pitch and loudness) of a typical audio signal are plotted against time, the graph is relatively smooth compared to a spreadsheet, a text document, or other typical files that may contain arbitrarily distributed byte-values. As a result, it is possible to construct a computable model of what the next sample in the signal will probably look like. ADPCM constructs such a model by examining a signal and comparing its predictions against actual observed values. It then encodes the difference between its prediction and the actual value.

ADPCM relies on the relative predictability of audio signals. If the changes in an audio signal are too great or sudden, the value that ADPCM records will be erroneous. In general, there is a certain statistically predictable amount of error that appears in any signal that is compressed by this method. The errors appear, not as distortions of the quality of the sound, but as pink noise, or hiss, much like the hiss on ordinary cassette recordings. This makes ADPCM compression suitable for many sound compression tasks, particularly for sound effects or speech in games or business software, but not for very high-fidelity reproduction. A signal compressed by the ADPCM method will likely be too noisy for use in professional musical or film recording.

How ADPCM works

The ADPCM model assumes that any particular digital sample in a block of audio data has a value that is relatively close to those on either side of it. In fact, the noise in the reproduced signal arises from samples that vary more than the model assumes. ADPCM predicts what the next value should be, and compares it with the value that is actually there. The difference is encoded in a value that is some number of bits in size, that size being specified by the implementation code. With ADPCM the programmer can specify encoded values either 3 or 4 bits wide. Since the original data is stored in 8-bit samples, the compression rate is either 8 to 3 or 8 to 4, depending on which size a particular program specifies.

Errors result when the difference between the original signal and the value that ADPCM predicts is greater than can be encoded in the specified number of bits. The encoded value then effectively becomes a random value, and so becomes audio noise. If the target code is 3 bits wide, then the difference observed by the compression algorithm is more likely to be out of range than if the code size is 4 bits. Greater compression results in greater loss of fidelity.

As stated earlier, the fidelity loss sounds like hiss, not like a gross distortion of the audio signal. Even using inaccurate predictive models, ADPCM tends to produce hiss rather than harmonic distortion. The technique tracks the gross characteristics of audio signals well even when the rate of errors is high. A decompressed signal sounds faithful to the original, though muffled by noise.

Audio Compression and Expansion Tool calls

The Audio Compression and Expansion Tool calls are all new calls, added to the Toolbox since the first edition of the Reference was published.

ACEBootInit $011D

Applications must not make this call. ACEBootInit performs any initializations of the ACE tools that are necessary at boot time.

Parameters

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

C

```
extern pascal void ACEBootInit() inline(0x011D,dispatcher);
```

ACEStartUp $021D

Initializes the ACE tools for use by an application. ACEStartUp sets aside a region of bank $00, specified by ZeroPageLoc, for use as the ACE tools' direct-page. At present, ACE uses one 256-byte page of bank $00 memory as its direct-page. Future versions of the ACE tools may use a different amount of memory for the direct-page, so applications should determine the correct size for the direct-page with a call to ACEInfo. The tool set's direct-page should always begin on a page boundary.

Parameters

Stack before call

```
|_ previous contents _|
|_   zeroPageLoc     _|   Word—Where to allocate direct-page space
|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|
|_                   _|   <-SP
```

Errors

$1D01	ACEIsActive
$1D02	ACEBadDP

C

```
extern pascal void ACEStartUp() inline(0x021D,dispatcher);
```

ACEShutDown $031D

Performs any housekeeping that is required to shut down the ACE tool set. Applications that use the ACE tools should always make this call before quitting. The application must allocate and deallocate direct-page space in bank zero.

Parameters

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

Errors

$1D03	ACENotActive

C

```
extern pascal void ACEShutDown() inline(0x031D,dispatcher);
```

ACEVersion $041D

Returns the version number of the currently installed ACE tool set. This call can be made before a call to ACEStartUp. VersionInfo will contain the information in the standard format prescribed by Toolbox version-number protocol.

Parameters

Stack before call

```
|_ previous contents _|
|_      space       _|  Word—Space for result
|_                  _|  <-SP
```

Stack after call

```
|_ previous contents _|
|_   versionInfo    _|  Word—Version number of ACE tool set
|_                  _|  <-SP
```

C

```
extern pascal Word ACEVersion() inline(0x041D,dispatcher);
```

ACEReset $051D

Resets the ACE tool set. This call is made by a system reset. Applications should never make this call because tool set initializations appropriate to a machine reset are performed.

Parameters

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

C

```
extern pascal void ACEReset() inline(0x051D,dispatcher);
```

ACEStatus $061D

Returns a Boolean flag, which is TRUE (nonzero) if the tool set has been started up, and FALSE (zero) if it has not. This call can be made before a call to ACEStartUp.

Parameters

Stack before call

```
|_ previous contents _|
|_       space       _|   Word—Space for result
|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|
|_    activeFlag     _|   Word—Boolean indicating whether tool set is active
|_                   _|   <-SP
```

C

```
extern pascal Word ACEStatus() inline(0x061D,dispatcher);
```

ACECompBegin $0B1D

Prepares the ACE tools to compress a new audio sequence. After ACECompress completes the process of compression and returns, the ACE tools normally save certain relevant state information so that subsequent calls to ACECompress can be used on succeeding parts of the same audio sequence. It is often desirable to break a long audio signal into smaller parts for compression. The preservation of appropriate state variables allows a call to ACECompress to compress part of such a signal and then, for a subsequent call, to continue the compression process where the previous call left off.

When a program calls ACECompress to process a new Audio sample, it should call ACECompBegin to ensure that all saved state information is cleared and that ACECompress is starting with a "clean slate." When an application is compressing a long audio sequence as a number of smaller pieces, it should call ACECompBegin *only* before the *first* sub-sequence. Thereafter, the application should not make this call until all parts of the sequence have been processed. The state information that ACE preserves between calls allows ACECompress to process subsequent blocks using appropriate information from previous ones.

Call ACECompBegin only before compressing the first sequence of a series of sub-sequences, or before compressing a single sequence that is not part of a longer sequence.

Parameters

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

Errors

$1D03 ACENotActive

C

```
extern pascal void ACECompBegin() inline(0x0B1D,dispatcher);
```

ACECompress $091D

Compresses the equivalent of *NBlks* of blocks of digital audio data and stores them at the specified location. The data to be compressed are located the equivalent of *SrcOffset* bytes beyond the location specified by the *Src* handle. The resulting compressed data are stored the equivalent of *DestOffset* bytes beyond the location specified by the *Dest* handle. The size of the source data is the equivalent of *NBlks**512 bytes. The data have been compressed using the method specified by the *Method* parameter; for the supplied ACE ADPCM methods 1 and 2, the size of the resulting data is (*NBlks**64*(5-*Method*)).

◊ *Note:* Because ACECompress is guaranteed to reduce the size of every byte of source data, the resulting data can be stored in the same place as the source data. The source and destination locations in RAM can be the same.

Important

The noisier a sampled signal is, the noisier the sample compressed using ADPCM will be. Any noise that is introduced into the signal produces discontinuities in the audio data, and causes errors in the compression and expansion process. For this reason, any editing, equalization, or other sound-processing effects should be applied to the original signal before it is compressed. ADPCM compression should be the last process applied to an audio signal before it is stored on the final disk.

Parameters

Stack before call

previous contents	
Src	**Long**—Handle to the source data
SrcOffset	**Long**—Offset from *Src* to the actual storage location
Dest	**Long**—Handle to storage for the resulting data
DestOffset	**Long**—Offset from *Dest* to the actual storage location
NBlks	**Word**—Number of 512 KB blocks of data
Method	**Word**—Method of compression
	<-SP

Stack after call

```
|_ previous contents _|
|_                   _|  <-SP
```

Errors

$1D05	ACEBadMethod
$1D06	ACEBadSrc
$1D07	ACEBadDest
$1D08	ACEDataOverlap

C

```
extern pascal void ACECompress() inline(0x091D,dispatcher);
```

ACEExpand $0A1D

Decompresses a previously compressed Audio sample, using the method specified by the *Method* parameter, and stores it at the specified location. Unlike ACECompress, ACEExpand cannot store its results in the same location as its source since the resulting data is 2 to 2.67 times as large as the source.

Parameters

Stack before call

```
|_ previous contents _|

|_                   _|
|_      Src          _|   Long—Handle to the source data

|_                   _|
|_    SrcOffset      _|   Long—Offset from Src to the actual storage location

|_                   _|
|_      Dest         _|   Long—Handle to storage for the resulting data

|_                   _|
|_    DestOffset     _|   Long—Offset from Dest to the actual storage location

|_      NBlks        _|   Word—Number of 512 KB blocks of data

|_      Method       _|   Word—Method of compression

|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|

|_                   _|   <-SP
```

Errors

$1D05	ACEBadMethod
$1D06	ACEBadSrc
$1D07	ACEBadDest
$1D08	ACEDataOverlap

C

```
extern pascal void ACEExpand() inline(0x0A1D,dispatcher);
```

ACEExpBegin $0C1D

Prepares ACE to expand a new sequence. Like ACECompBegin, ACEExpBegin clears any stored state information from previous calls to expand compressed data. A large compressed sample can be decompressed by processing it as a series of sub-sequences with repeated calls to ACEExpand, because certain appropriate state variables are preserved from call to call. If you are calling ACEExpand to work on a new sequence which bears no relation to any other compressed sequence, or to expand a short sequence in just one call to ACEExpand, you should make this call first to clear these state variables. If, on the other hand, you are making a call to ACEExpand to decompress a sequence that is a part of a longer sequence and is not the first sub-sequence, you should *not* make this call first, because it will throw away all information that ACE has recorded about the previous sequences.

Parameters

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

Errors

$1D03 ACENotActive

C

```
extern pascal void ACEExpBegin() inline(0x0C1D,dispatcher);
```

ACEInfo $071D

Returns certain information about the currently installed version of the ACE tools. This call can be made before a call to ACEStartUp. The InfoItemCode parameter specifies what information the call is to return. At present, the only valid value is 0. This specifies that the call will return the size in bytes of the direct-page that ACE requires.

Parameters

Stack before call

```
|_ previous contents _|

|_                   _|
|_       space       _|   Long—Space for result

|_   infoItemCode    _|   Word—What type of info to return

|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|

|_                   _|
|_  InfoItemValue    _|   Long—Specified information

|_                   _|   <-SP
```

Errors

$1D04 ACENoSuchParam

C

```
extern pascal LongWord ACEInfo() inline(0x071D,dispatcher);
```

Chapter 3

Control Manager

This chapter documents new features and information about the Control Manager. The complete Control Manager documentaion is in Volume 1, Chapter 4 of the *Apple IIGS Toolbox Reference.*

New features in the Control Manager

The Control Manager has the following new features:

- Colors in control tables now use all four color bits in both modes; they formerly used only bits 0 and 1 in 640 mode. For any applications that use color controls in 640 mode, the effect is that controls will be a different color. This change was made so that dithered colors can be used with controls.
- The barArrowBack entry in the scroll bar table was never implemented as first intended, and is now no longer used.
- The Control Manager preserves the current port across Control Manager calls , including those that are passed through other tools, such as the Dialog Manager.
- The Control Manager no longer changes the following fields in the port of a window that contains controls:

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

- The Control Manager uses the state of the window port to compute the size of control bounds RECTs when creating a control.
- The Control Manager uses the new SpecialRect call if it is available, instead of making separate calls to FrameRect and FillRect.

Error corrections

This section explains corrections and improvements that have been made to Control Manager routines.

Drawing controls

The following list contains descriptions of changes the Control Manager's facilities for drawing controls:

- TestControl returns a zero if an invisible or inactive control is selected.
- MoveControl does not make invisible controls visible when moving them.
- SetCtlTitle redraws the titles of controls.
- The grow box control now has its own color table. Formerly this control shared the simple button default color table. Because of this a size box was drawn as a single black box. The highlighted grow box now appears as a white icon in a black box.

  The previous grow box default color table looked like this:

$0000	Black outline for box
$00F0	Not highlighted: black outline in white interior
$0000	Highlighted: black outline in black interior

  The new grow box default color table looks like this:

$0000	Black outline for box
$00F0	Not highlighted: black outline in white interior
$000*F*	Highlighted: white outline in black interior

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

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

Miscellaneous

The following additional changes have been made to the Control Manager:

- On page 4-76 of the Reference, in the section that covers the SetCtlParams call, it states that the call "Sets new parameters to the control's definition procedure..." This description is misleading; the call does not directly set the parameters. Rather, it *sends* the new parameters to the control's definition procedure, unlike SetCtlValue, which actually sets the appropriate value in the control record and then passes the value on to the definition procedure.
- The current version of the Control Manager maintains the required relationship among the *value*, *view*, and *size* fields of a scroll bar record. In earlier versions of the Control Manager it was the responsibility of the application to ensure that the *value* field never exceeded the quantity (*size-view*). The Control Manager now adjusts the *value* or *size* field if the other quantities are set to invalid values. For example, if *view* = 30 and *size* = 100, then the maximum *value* allowed is 70. If an application sets the control *value* field to 80, the Control Manager adjusts *size* to 110. If *value* = 70 and the application sets *size* to 90, the Control Manager adjusts *value* to 60. *view* can also be changed in a way that invalidates the three settings. In the example mentioned before, in which value = 70, view = 30, and size = 100), setting *view* to 40 will cause *value* to be set to 60.

Chapter 4

Desk Manager

This chapter documents new features of the Desk Manager. The complete reference to the Desk Manager is in Volume 1, Chapter 5 of the *Apple IIGS Toolbox Reference.*

New features in the Desk Manager

It is now possible for a New Desk Accessory(NDA) to be a modal dialog box. When an NDA is opened it returns a pointer to its window. The Desk Manager saves this pointer and marks the NDA open. Subsequent attempts to open the NDA simply select the open window until the NDA is closed. The current version of the Desk Manager checks the returned window pointer, and if its value is zero (if it is a null pointer) then the Desk Manager does not mark the NDA open. A programmer can therefore write an NDA that opens a modal dialog box when chosen. When the dialog box is dismissed, the NDA can be chosen again without having been explicitly closed.

Chapter 5

Dialog Manager

This chapter documents new features of the Dialog Manager. The complete reference to the Dialog Manager is in Volume 1, Chapter 6 of the *Apple IIGS Toolbox Reference.*

Error corrections

This section explains changes that have been made to correct problems with the Dialog Manager, and with its documentation in the *Apple IIGS Toolbox Reference.*

- The documentation for SetDItemType on page 6-82 of the Toolbox Reference says that the call is used to change a dialog item to a different type. In fact, SetDItemType should only be used to change the *state* of an item from enabled to disabled or vice-versa.
- GetItext formerly always stored at least 3 bytes of data into the resultPtr passed to it. This posed a problem if the editline item was only one character, meaning only two bytes should be stored: one for the length and one for the character itself. This has been fixed.
- GetNewModalDialog no longer crashes when passed a nonzero refcon value.
- IsDialogEvent now correctly claims all window control events.
- HideDITem, ShowDItem, GetDItemValue, EnableDItem and DisableDItem no longer crash with invalid item IDs.
- Several Dialog Manager calls failed if given invalid item IDs. This has been fixed. The calls affected are

    SetDItemValue

    GetDItemType

    SetDItemType
- When paramText characters (^0 through ^3) were used at the end of a line, garbage characters were appended. They now work correctly.
- DialogStatus formerly returned a value of 'active' after the Dialog Manager had been shut down. This has been fixed.
- Certain Dialog Manager and LineEdit calls assumed that foreground and background colors in the applicable grafPort were correctly set. This is actually only true if other color controls have previously been drawn. This problem has been fixed. The affected calls are

    StatText

    LongStatText

    LineEdit drawing routines.

Chapter 6

Event Manager

This chapter documents new features of the Event Manager. The complete reference to the Event Manager is in Volume 1, Chapter 7 of the *Apple IIGS Toolbox Reference.*

New call

SetAutoKeyLimit is a new call in the Event Manager.

SetAutoKeyLimit $1A06

Controls how repeated keystrokes are inserted into the event queue. The default value for the limit is zero, which specifies that autokey events are inserted only if no other events are already in the queue. *newLimit* determines how many autokey events must be in the event queue before PostEvent ceases to add them. If *newLimit* is zero, then the default condition is maintained: PostEvent will not add autokey events unless the queue is empty. If the *newLimit* is 5, then PostEvent will add 5 autokey events to the queue before it reverts to the rule that no more autokey events are to be posted.

Parameters

Stack before call

```
|_ previous contents _|
|_     newLimit      _|   Word—Limit for inserted autokey events
|_                   _|   <-SP
```

Stack after call

```
|_ Previous contents _|
|_                   _|   <-SP
```

C

```
extern pascal void SetAutoKeyLimit();
```

Chapter 7

Font Manager

This chapter documents new features of the Font Manager. The complete reference to the Font Manager is in Volume 1, Chapter 8 of the *Apple IIGS Toolbox Reference.*

New features in the Font Manager

The current version of the Font Manager incorporates several changes. In previous versions, FMStartUp opened each font file in the FONTS folder, and constructed lists of information for all available fonts. These lists contained font IDs, font names, and so forth for every font in every file in the FONTS folder. The present version of the Font Manager does this same work the first time it starts up, but caches all the information it compiles in a file called FONT.LISTS in the FONTS folder.

The next time the Font Manager starts up, it checks all the creation and modification dates and times in font files against the information in FONT.LISTS.It compile new FONT.LISTS information only if it finds new font files or other evidence of change. Otherwise, it simply starts up with the information stored in the LISTS file. In most cases, because it doesn't have to open every font file, the Font Manager can start up much more quickly.

New call

The new call InstallWithStats is provided to simplify the process of installing fonts. It allows an application to preserve certain information that is normally lost during font installation.

InstallWithStats $1C1B

Installs a font and returns information about that font. When an application requests the installation of a font, the Font Manager attempts to install the requested font, but it may not be available. In such cases, the Font Manager will install the closest match it can find to the requested font.

InstallWithStats installs a font just as if the application had called InstallFont, but it returns a FontStatRec in the buffer pointed to by ResultPtr. This record contains the ID of the installed font, which may be different from the font requested. It also contains the purge status that the font had before it was installed. Since purge status can be changed by installation, this information can make it easier to restore a font's purge status. If you need to know an installed font's purge status, use FindFontStats.

Parameters

Stack before call

```
|_ previous contents _|

|_                   _|
|_     desiredID     _|   Long—FontID of desired font

|_     scaleWord     _|   Word—Desired font-size

|_                   _|
|_     resultPtr     _|   Long—Pointer to result of call

|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|

|_                   _|   <-SP
```

C

```
extern pascal void InstallWithStats() inline(0x1C1B,dispatcher);
```

Chapter 8

List Manager

This chapter documents new features of the List Manager. The complete reference to the List Manager is in Volume 1, Chapter 11 of the *Apple IIGS Toolbox Reference.*

Clarification

The Reference states that a disabled item of a list cannot be selected. In fact, a disabled item can be selected, but it cannot be highlighted. The List Manager provides the ability to select disabled (unhighlighted) items so that it is possible, for instance, for a user to select a disabled menu choice as part of a help dialog.

Member text is now drawn in 16 colors in both 320 and 640 mode.

List Manager definitions

disabled	Bit 6 of the list-item's memFlag field is set. Disabled items appear dimmed and cannot be highlighted.
enabled	Bit 6 of the list-item's memFlagfield is clear. Enabled items appear normal and can be highlighted.
selected	Bit 7 of the list-item's memFlag field is set. This bit is set when a user clicks on the list-item, or the item is within a range of selected items. A selected item only appears highlighted if it is also enabled.
highlighted	A member of a list only appears highlighted when it is both selected and enabled. This means that bit 7 of the memFlag field is 1 and bit 6 is 0. A highlighted member is drawn using the highlight colors.

Chapter 9

Memory Manager

This chapter documents new features of the Memory Manager. The complete reference to the Memory Manager is in Volume 1, Chapter 12 of the *Apple IIGS Toolbox Reference.*

New call

RealFreeMem is a new Memory Manager call designed to provide accurate information about available memory.

RealFreeMem $2F02

Returns the number of bytes in memory that are free, plus the number that could be made free by purging. FreeMem only returns the number of bytes that are actually free, ignoring memory that is occupied by unlocked purgeable blocks. Since unlocked blocks of allocated memory can be freed by purging, FreeMem does not provide an accurate picture of the memory that is actually available. RealFreeMem provides a more accurate value.

Parameters

Stack before call

```
|_ previous contents _|

|_                   _|
|_       space       _|   Long—Space for result
|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|

|_                   _|
|_     freeBytes     _|   Long—Number of available bytes in memory
|_                   _|   <-SP
```

C

```
extern pascal Word RealFreeMem() inline(0x2F02,dispatcher);
```

Chapter 10

Menu Manager

This chapter documents new features of the Menu Manager. The complete reference to the Menu Manager is in Volume 1, Chapter 13 of the *Apple IIGS Toolbox Reference.*

New Information

This section lists several new features of the Menu Manager, and some information that was not previously clear.

- Menus in windows can now display the Apple character (ASCII $14).
- Menus now use their outline color for lines which separate menu items.
- The NewMenuBar call automatically sets bit 31 of the CtlOwner field in the menubar record, if the designated menubar is a window menubar, and the value passed for the window is not zero.
- The Menu Manager's justification procedures adjust for menubars in windows. Menus will be moved to the left if they would otherwise appear to the right of the menubar's right end.
- The default menubar has the following coordinates: top = 0; left = 0; height = 13; width = the width of the screen.
- MenuShutDown does not return an error if the Menu Manager has already been shut down.
- The CalcMenuSize call uses the newWidth and newHeight parameters to compute a menu's size. These parameters may contain the width and height of the menu, or may contain the values $0000 or $FFFF. A value of $0000 tells CalcMenuSize to calculate the parameter automatically. A value of $FFFF tells it to calculate the parameter only if the current setting is zero.

  The effect of all three uses:

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

Menu caching

The current version of the Menu Manager, on System Disks versions 3.2 and later, introduces new menu caching features. Menu caching is designed to provide faster display of menus under certain circumstances. When a menu is drawn on the screen, the area of the screen that it covers is copied into a buffer. When the menu goes away, the contents of the buffer are copied back to the screen.

With the menu caching feature, when the saved screen image is copied back to the screen, the menu that goes away is copied into the buffer. In other words, the Menu Manager swaps the menu image with the screen image. Therefore, the next time that menu is pulled down, the Menu Manager can copy it from the buffer instead of drawing a new image.

If the menu image changes, for example, an item is disabled or the items on the menu change, then the cached image is inaccurate, and the Menu Manager must redraw the menu. In those cases where a menu image does not change, the menubar can respond to the user more quickly.

Table 1 Calls that can change a menu image

CalcMenuSize
CheckMItem
DeleteMItem
DisableMItem
EnableMItem
FixMenuBar
InsertMItem
MenuNewRes
SetBarColors
SetMenuFlag
SetMItem
SetMItemFlag
SetMItemMark
SetMItemName
SetMItemStyle

Menu caching should not increase memory requirements since menu images are purgeable when not displayed on the screen.

This menu caching scheme should work properly with all existing standard menus. You will have to alter custom menus, however, so that they can take advantage of menu caching. Custom menus will still function normally, as long as they do not change the menu record directly, but they will not be able to take advantage of the menu caching scheme to speed up display.

Caching does not work with menus in windows, so the InsertMenu call automatically disables caching for such menus.

Caching with custom menus

Bit 3 of the MenuFlag field in a menu record indicates whether a menu's definition procedure knows about caching. A value of 1 indicates that the menu in question works correctly with caching. A custom menu that uses caching must define a menu record that sets this flag, and allocates an extra field, a handle to the cache in which the menu image will be stored. (See Table 2.

Table 2 Fields in a cachable menu record

MenuID	WORD	
MenuWidth	WORD	
MenuHeight	WORD	
MenuProc	LONG	
MenuFlag	BYTE	; Bit 3 = 1 to enable caching
MenuRes	BYTE	
FirstItem	BYTE	
NumOfItems	BYTE	
TitleWidth	WORD	
TitleName	LONG	
MenuCache	LONG	; New field in cachable menu records ; Handle to cache

The FixMenuBar call automatically allocates a cache for the defined menu if the caching flag is set.

Error corrections

This section explains some corrections that were made to the Menu Manager.

- CalcMenuSize has been modified so that it takes text styles into account when calculating the width of a menu.
- Display of menu titles has been corrected so they will be arranged properly relative to the left side of the menubar RECT.
- The CtlOwner flag of a menu inside a window must be negative (bit 31 must be set). This has always been true, but has been undocumented.

Chapter 11

MIDI Tool Set

This chapter documents the MIDI Tool Set. This is a new tool set and was not documented in the *Apple IIGS Toolbox Reference.*

About the MIDI Tool Set

One of the most interesting uses that has been invented for electronic circuitry is control of digital musical instruments. MIDI stands for Musical Instrument Digital Interface. It is the standard communications protocol that the electronic music industry uses to connect various digital instruments. A synthesizer or sequencer properly equipped with a MIDI bus can control or be controlled by another such device, so that a musician can connect complicated groups of MIDI instruments and control them easily.

If you add a suitable MIDI interface to your personal computer, you can connect it to a MIDI network. With appropriate software, the computer can control the network, and can record, edit, and play musical sequences. You can store musical data on disk and transfer them across telecommunications networks. The Apple IIGS has the capacity, with the MIDI tools, to control external MIDI instruments.

How the MIDI Tool Set works

The MIDI tools consist of a group of utilities that enable you to collect incoming MIDI data from the designated serial port, store the data in designated buffers, process them, and play them back at will. The tools are hardware independent. They use separately loaded device drivers, and so the MIDI tools themselves don't incorporate any assumptions about the nature of the MIDI interface that they might be driving. At present the Apple IIGS system software includes drivers for two MIDI interfaces: Apple.MIDI, for the Apple MIDI Interface, and Card6850.MIDI for plug-in ACIA cards, such as the Passport MIDI interface.

The MIDI tools provide fast response to MIDI data transfers and an accurate clock for time-stamping MIDI packets. The tools allocate one of the 14 general purpose generators on the Digital Oscillator Chip (DOC), and the first 256 bytes of DOC RAM, for use as a clock. This clock counts intervals of 76 microseconds, which allows fine timing of MIDI input and output data. It is sufficient to record and accurately reproduce large chords played on a MIDI keyboard. MIDI data may arrive as quickly as 1 byte every 320 microseconds, but the Apple IIGS can receive the data without loss as long as no running code disables interrupts for more than 300 microseconds. The tool set also provides a polling scheme so that it can receive MIDI data without loss even when interrupts must be disabled for longer times.

Interrupts drive the MIDI tool set's functions, so the computer's CPU can be occupied with other tasks as MIDI data are received or transmitted in the background. Because the interrupt scheme is a part of the MIDI tool set's design, there is no need for an application programmer to provide interrupt handlers. There are also error-checking mechanisms, and the tools will detect and report several different types of MIDI errors. The tool set can handle MIDI transfers as raw data, performing no interpretation, or it can assemble the data into MIDI packets, with packet-length and time stamps. The tool set can also return MIDI packets in the Standard MIDI File Format, established by the MIDI Manufacturers' Association in January 1988.

You can process MIDI information in real time, responding to keyboard control or directing immediate output from the computer to a MIDI instrument, or you can record MIDI data for batch-processing and later playback. This gives an application the power to act either as a sequencer and sequence editor, or a real-time controller for directing MIDI program and patch changes. The tool set is able to distinguish between MIDI notes that it has initiated and those it has not, and you can selectively switch off only the notes originated by the MIDI tools or all notes. You may also choose to switch off notes on any combination of MIDI channels.

Version requirements

If you use the Sound Tools, Note Synthesizer, and Note Sequencer with the MIDI tools, you will need certain minimum version numbers. The required minimum versions are

Sound Tools	v2.3
Note Synthesizer	v1.2
Note Sequencer	v1.2

Apple engineers adapted the above tools for use with the MIDI Tool Set, and previous versions cause MIDI data losses. The previous versions disabled interrupts for more than 300 microseconds at a time, which will cause problems because standard MIDI equipment transfers data at a rate of one byte every 320 microseconds.

You will only need the Note Synthesizer if you want to use the MIDI time-stamp clock. The clock is actually a DOC generator, and the Note Synthesizer is used to allocate it. If the MIDI clock is not in use, you may start up or shut down the Note Synthesizer at will. If, however, you are using the MIDI clock, then the Note Synthesizer must be loaded and started up. You do not need the Sound Tools or the Note Sequencer to use the MIDI tools, and you can load them or not as you choose.

Using the MIDI Tool Set

When you have successfully loaded and started the MIDI Tool Set, you will need to load a MIDI device driver. You must choose a driver and supply the MIDIDevice call with the slot number that the MIDI interface is using and the pathname of the driver . Once the device driver is loaded, you will need to allocate input and output buffers for MIDI data. Your application will need these buffers if it ever makes calls to MidiReadPacket or MidiWritePacket.

Input and output of MIDI data are directed as independent processes so that the Apple IIGS can perform other functions independently. When you have configured your system as described above, your application can perform whatever editing or other tasks you desire, and you can start or stop MIDI input or output any time you like. For example, your application could allow a user to start output, whereupon the tool set starts transmitting the contents of the output buffer. If the application periodically makes calls to MidiWritePacket to send the appropriate data to the output buffer, the sequence can play in the background while the user is using editing functions at the same time. Input and output processes can be active at the same time, so that a user could be both playing and recording simultaneously; this is useful for making multi-track sequences.

You can start and stop the clock provided with the MIDI tool set with calls to MidiClock. This allows you to accurately stamp MIDI packets for timing purposes. The clock increments every 76 milliseconds, and when a MIDI packet is received, the tool set stamps it with the current value of the clock. In this way, your application can keep accurate account of when a particular note, chord, or program change is to occur.

MIDI Tool calls

All the MIDI Tool Set calls are new calls, added to the Toolbox since publication of the *Apple IIGS Toolbox Reference.*

The routines you will use to work with the MIDI tool set are MidiControl, MidiDevice, MidiInfo, MidiReadPacket, and MidiWritePacket. Three of these calls are multifunction calls, which perform different actions depending on a control parameter passed to them. The workhorse of the group is MidiControl,which performs 18 different functions depending on the control function parameter. The other multipurpose calls are MidiDevice and MidiInfo.

The MIDI multipurpose tool calls are briefly described as follows. See the call descriptions for more complete information.

MidiControl Performs 18 different MIDI control functions as selected by *funcNum*, the first parameter to the call. The functions are selected by a numeric parameter passed to the MidiControl call.

funcnum	**Function**
0	Set real-time vector
1	Set real-time error vector
2	Allocate input buffer
3	Allocate output buffer
4	Start MIDI input
5	Start MIDI output
6	Stop MIDI input
7	Stop MIDI output
8	Flush input buffer
9	Flush output buffer
10	Flush input packet
11	Wait for output buffer to clear
12	Set input mode
13	Set output mode
14	Clear note pad
15	Set MIDI delay
16	Enable/disable running status output
17	Enable/disable receipt of system exclusive packets

MidiDevice Selects, loads, and unloads MIDI device drivers.

funcnum	**Function**
0	Not implemented in version 1.1
1	Load a device driver
2	Unload a device driver

MidiInfo Returns specified information about the state of the MIDI Tool Set.

funcnum	**Function**
0	Number of bytes in next input packet
1	Number of bytes waiting in input buffer
2	Number of bytes waiting in output buffer
3	Maximum number of bytes in input buffer
4	Maximum number of bytes in output buffer
5	Not implemented in version 1.1
6	Not implemented in version 1.1
7	Time stamp clock value
8	Time stamp clock frequency

MIDIBootInit $0120

Initializes the MIDI Tool Set; called only by the Tool Locator. An application must never make this call.

Parameters

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

C

```
extern pascal void MidiBootInit() inline(0x0120,dispatcher);
```

MIDIStartUp $0220

Starts up the MIDI Tools for use by an application. Applications should make this call before any other calls to the MIDI Tools. Normally an application must next call MidiDevice to load a MIDI device driver, and then MIDIControl to allocate an input buffer and an output buffer.

Parameters

Stack before call

|_ *previous contents* _|	
|_ *userID* _|	**Word**—User ID for the Memory Manager
|_ *zeroPageLoc* _|	**Word**—Beginning of three-page MIDI direct-page
|_ _|	**<-SP**

Stack after call

|_ *previous contents* _|	
|_ _|	**<-SP**

C

```
extern pascal void MidiStartUp() inline(0x0220,dispatcher);
```

MIDIShutDown $0320

Shuts down the MIDI Tool Set. An application that uses the MIDI Tools should make this call before it quits. MIDIShutDown deallocates the input and output buffers, stops the MIDI clock and deallocates its generator, and shuts down the hardware interface. The call's actions take place immediately, so the application should take any necessary steps to see that all recent MIDI output has been sent before shutting down the tools (see MIDIControl).

Parameters

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

C

```
extern pascal void MidiShutDown() inline(0x0320,dispatcher);
```

MIDIVersion $0420

Returns the version number of the currently loaded MIDI Tools according to standard Toolbox version number protocol.

Parameters

Stack before call

```
|_ previous contents _|
|_       space       _|   Word—Space for result
|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|
|_    versionNum     _|   Word—Version of currently installed MIDI tools
|_                   _|   <-SP
```

Errors

$2007 no output buffer allocated

C

```
extern pascal Word MidiVersion() inline(0x0420,dispatcher);
```

MIDIReset $0520

Resets the MIDI Tools; called by system reset. An application must not make this call.

Parameters

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

C

```
extern pascal void MidiReset() inline(0x0520,dispatcher);
```

MIDIStatus $0620

Returns a Boolean value of TRUE if the MIDI Tools are active, and FALSE if they are not.

Parameters

Stack before call

```
|_ previous contents _|
|_      space        _|  Word—Space for result
|_                   _|  <-SP
```

Stack after call

```
|_ previous contents _|
|_    activeFlag     _|  Word—Boolean; TRUE if the tool set is active
|_                   _|  <-SP
```

C

```
extern pascal Word MidiStatus() inline(0x0620,dispatcher);
```

MIDIClock $0B20

Controls operation of the optional time-stamp clock. The clock ticks once every 76 microseconds, and allows events to be scheduled for precise timing. The *funcNum* parameter specifies which clock function to perform, and the *arg* parameter provides the argument to the selected function.

Parameters

Stack before call

```
|_ previous contents _|
|_     funcNum       _|   Word—MIDIClock function number
|_                   _|
|_       arg         _|   Long—Argument to MIDIClock function
|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|
|_                   _|   <-SP
```

Errors

See the MidiClock function descriptions below.

C

```
extern pascal void MidiClock() inline(0x0B20,dispatcher);
```

MIDIClock functions

MIDIClock controls the operation of the MIDI clock provided by the tool set. Four different functions are provided for clock control. They are as follows:

0 miSetClock

The value of *arg* becomes the new value of the time stamp clock. The most significant bit of the *arg* parameter must be zero. There is a limit to the accuracy with which the clock can be set. The least significant byte of the time stamp clock will always be zero if the clock is stopped. If the clock is running, the value of the least significant byte will be undefined for the purposes of this call. The result is that an application can set the clock only to within 20 milliseconds of a particular value.

1 miStartClock

Allocates a DOC generator, writes consecutive values from $00 through $FF into the first page of the DOC RAM, and starts the clock. By default, the clock starts counting at zero. If the application stops the clock and restarts it, it starts with the same value it had when it stopped, unless the value is changed with an miSetClock call. miStartClock should be called before miStartInput. The process of starting the clock is time-consuming and disables interrupts, so it could cause MIDI data to be lost if it is done while the application is receiving a MIDI transmission. The Sound Tools and the Note Synthesizer *must* be loaded and started up before this call is issued.

Errors:

$0810	no DOC or DOC RAM found
$1921	no DOC generator available
$1923	Note Synthesizer not started

2 miStopClock

The MIDI time stamp clock is stopped. The DOC generator and its associated RAM are released for use by the Note Synthesizer. MIDI data that is received while the clock is stopped is stamped with the value of the stopped clock. Any output packets with time stamps greater than the value of the stopped clock are not sent until the clock is restarted or reset.

3 miSetFreq

The frequency of the Apple IIGS MIDI clock cannot be changed, but this may not be true in future versions.

MIDIControl $0920

Performs 18 different control functions required by the MIDI Tool Set.

The *funcNum* parameter selects which function is to be performed, and the arg parameter passes any argument required by that function.

Parameters

Stack before call

```
|_ previous contents _|
|_      Space        _|  Word—Space for result
|_     funcNum       _|  Word—Function to perform
|_                   _|
|_       arg         _|  Long—Argument to MIDIControl function
|_                   _|  <-SP
```

Stack after call

```
|_ previous contents _|
|_                   _|  <-SP
```

C

```
extern pascal void MidiControl() inline(0x0920,dispatcher);
```

MidiControl functions

A MidiControl call has 18 legal values for the *funcNum* parameter. Each value invokes a different control function. The functions are as follows:

0 miSetRTVec The long parameter contains the address of a service routine in the application. When the application receives MIDI real-time commands, it calls this service routine. A value of zero in this parameter disables the service routine. The service routine must not disable interrupts, and if it runs for longer than 300 microseconds, it must call the MIDI polling vector at least every 300 microseconds. The only MIDI calls that the service routine should make are MidiReadPacket and MidiWritePacket.

1 miSetErrVec The long parameter contains the address of a service routine in the application. The MIDI tool set calls this routine in the event of a real-time error. A value of zero in the parameter disables the service routine. This service routine must not enable interrupts. If it executes for longer than 300 microseconds, it must call the MIDI polling vector at least every 300 microseconds.It can call MidiWritePacket and MidiReadPacket, but no other MIDI tool calls.

2 miSetInBuf The long parameter contains a pointer to a 6-byte record. The fields of this record are as follows:

WORD	Size of input buffer
LONG	Address of input buffer

If the address is zero, the MIDI Tool Set will allocate the input buffer. If the specified size is zero, the MIDI tools will allocate a buffer 8 KB in size. If the application allocates the buffer it must be non-purgeable, in a fixed location, and must not cross bank boundaries.

3 miSetOutBuf The long parameter contains a pointer to a six-byte record. The fields of this record are as follows:

WORD	Size of output buffer
LONG	Address of output buffer

If the address is zero, the MIDI Tool Set will allocate the output buffer. If the specified size is zero, the MIDI Tool Set will allocate a buffer 8KB in size. If the application allocates the buffer, it must be non-purgeable, in a fixed location, and must not cross bank boundaries.

4 **miStartInput**	Starts an interrupt-driven process that reads MIDI data into the MIDI Tools' input buffer. An application can retrieve these data with a MidiReadPacket call. The long parameter contains the address of a service routine called when the first packet is available in a previously empty input buffer. A value of zero disables this service routine. The service routine must not disable interrupts, and if it runs for longer than 300 microseconds, it must call the MIDI polling vector at least every 300 microseconds. The only MIDI calls that the service routine should make are MidiReadPacket and MidiWritePacket.
5 **miStartOutput**	Starts an interrupt-driven process that writes MIDI data to the MIDI Tools' output buffer. The routine places the data into the output buffer using calls to MidiWritePacket. The long parameter contains the address of a service routine called when the output buffer is completely empty. A value of zero disables this service routine. The service routine must not disable interrupts, and if it runs for longer than 300 microseconds, it must call the MIDI polling vector at least every 300 microseconds. The only MIDI calls that the service routine should make are MidiReadPacket and MidiWritePacket.
6 **miStopInput**	Causes the MIDI Tool Set to ignore MIDI data until the next miStartInput call.
7 **miStopOutput**	Halts MIDI output until the next miStartOutput call.
8 **miFlushInput**	Discards the contents of the current input buffer.
9 **miFlushOutput**	Discards the contents of the current output buffer. The long parameter selects the method:

Long parameter **Action:**

$0000 00XX	Wait for the current packet to finish transmission, then turn off all notes that have not been turned off in channel XX. If XX = $10, turn off notes in all channels.
$0001 00XX	Wait for current packet to finish transmission, then turn off all possible notes (pitch $00 through $7F) in channel XX. If XX = $10, turn off notes in all channels.
$FFFF XXXX	Discard the contents of the output buffer immediately without turning off any notes.

10 miFlushPacket Discards the next input packet.

11 miWaitOutput Ceases execution until the output buffer becomes empty. This function may never return if output is disabled.

12 miSetInMode The long parameter selects the input mode. The legal values are as follows:

Long parameter	**Input mode:**
0	Raw mode. MIDI data is transferred unchanged.
1	Packet mode. MIDI data is converted to packets, with length of packet and time stamp bytes added to the front of each packet.
2	Standard mode. MIDI data is returned in Standard MIDI FIle Format.

13 miSetOutMode The long parameter selects the output mode. The legal values are as follows:

Long parameter **Input mode:**

	0	Raw mode. This mode is very similar to packet mode, but no attempt is made to keep track of which notes are on. Running status optimization is still performed unless explicitly disabled by miOutputStat. Because no record is kept of which notes are on, all notes that are turned on, must be explicitly turned off.
	1	Packet mode. MIDI data is converted to packets. The MIDI Tools track note-on and note-off commands.
	2	Standard mode. MIDI data is returned in Standard MIDI FIle Format.
14 miClrNotePad	Erases the MIDI Tool Set's record of which notes are on and which are off. This call causes the tool set's record to show that all notes are off.	
15 miSetDelay	Sets a delay value for use with MIDI synthesizers that cannot process MIDI data at the full MIDI transfer rate. The low word of the long parameter specifies a minimum delay between packet sends in units of 76 microseconds.	
16 miOutputStat	Enables or disables standard MIDI running status mode. When running status is enabled, MIDI status bytes are sent only when they change, or are otherwise absolutely necessary. This optimization speeds transmission and reduces CPU overhead, but can cause malfunctions if the synthesizer and computer disagree on the current value of the status byte. The long parameter value is $0000 to disable running status, $0001 to enable it.	
17 miIgnoreSysEx	Specifies whether to ignore MIDI System Exclusive data. System exclusive packets begin with the value $F0. If the application ignores system exclusive packets, they will not be buffered, and the application will not receive them. The parameter must be $0000 to ignore system exclusive data, $0001 to accept it.	

MIDIDevice $0A20

Allows an application to select, load, and unload device drivers for use with the tools. The call interprets the *driverInfo* parameter as the address of the driver to be loaded. The funcNum parameter specifies whether the driver is to be

1: loaded

or

2: unloaded

Parameters

Stack before call

```
|_ previous contents _|
|_     funcNum       _|   Word—Function to perform
|_                   _|
|_     drvrPtr       _|   Long—Pointer to device driver information
|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|
|_                   _|   <-SP
```

Errors

See the function descriptions below.

C

```
extern pascal void MidiDevice() inline(0x0A20,dispatcher);
```

MIDIDevice functions

MIDIDevice loads and unloads MIDI device drivers, which allow the MIDI Tools to drive a particular MIDI interface. The present version of the MIDI Tool Set supports the Apple MIDI Interface and ACIA 6850 MIDI Interface cards, such as the Passport MIDI interface. The function of the call is selected by the value of the *funcNum* parameter. The legal values are as follows:

0 Not implemented in version 1.1

1 miLoadDrvr

The *drvrPtr* parameter points to a device driver record, which specifies a device driver to be loaded. The call loads the specified device driver into memory, after shutting down and unloading any previously loaded device drivers. It then initializes the newly loaded driver.

Errors:

$2008	miDriverError
$2080	miDevNotAvail
$2081	miDevSlotBusy
$2082	miDevBusy
$2084	miDevNoConnect
$2086	DevVersion
$2087	miDevIntHndlr

2 miUnloadDrvr

Shuts down and unloads the currently loaded device driver. Terminates MIDI transmission or reception if they are currently active. Releases memory occupied by the device driver.

MIDIInfo $0C20

Returns certain information about the state of the MIDI Tools. The *funcNum* parameter can specify nine different functions, whose results are returned in result.

funcNums:

0.	number of bytes in next input packet
1.	number of bytes of data in input buffer
2.	number of bytes of data in output buffer
3.	maximum number of bytes of data in input buffer
4.	maximum number of bytes of data in output buffer
5.	address of packet being recorded by MidiRecordSeq (not yet implemented)
6.	address of packet being played by midiPlaySeq (not yet implemented)
7.	time stamp clock value
8.	time stamp clock frequency

Parameters

Stack before call

```
|_ previous contents _|
|_                   _|
|_      space        _|   Long—Space for result
|_     funcNum       _|   Word—Desired information
|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|
|_                   _|
|_    infoResult     _|   Long—The requested information
|_                   _|   <-SP
```

Errors

$2007	No input buffer allocated

C

```
extern pascal LongWord MidiInfo() inline(0x0C20,dispatcher);
```

MIDIReadPacket $0D20

Returns the length of a packet of MIDI data that it has transferred from the input buffer to the indicated array. If no packet is available, the call returns a zero. The first two bytes of the packet contain a value specifying the length in bytes of the packet. The next four bytes contain the MIDI clock time stamp. All bytes after the time stamp are actual MIDI data.

Parameters

Stack before call

```
|_ previous contents _|
|_       space       _|  Word—Space for result
|_                   _|
|_       bufPtr      _|  Long—Pointer to buffer to hold returned packet
|_       bufSize     _|  Word—Size in bytes of the packet buffer
|_                   _|  <-SP
```

Stack after call

```
|_ previous contents _|
|_       Result      _|  Word—Number of bytes returned
|_                   _|  <-SP
```

C

```
extern pascal Word MidiReadPacket() inline(0x0D20,dispatcher);
```

MIDIWritePacket $0E20

Queues the specified MIDI packet into the output buffer. If the packet is successfully written to the output buffer, the number of bytes written is returned. Otherwise, MIDIWritePacket returns zero. The packet format is the same as that used by MIDIReadPacket. The first two bytes contain the number of bytes in the packet. The next four contain the packet's time stamp. The remaining bytes contain the MIDI data.

Parameters

Stack before call

```
|_ previous contents _|
|_       space       _|   Word—Space for result
|_                   _|
|_      bufPtr       _|   Long—Pointer to buffer containing MIDI packet
|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|
|_   bytesWritten    _|   Word—Number of bytes written to the output buffer
|_                   _|   <-SP
```

C

```
extern pascal Word MidiWritePacket() inline(0x0E20,dispatcher);
```

Chapter 12

Miscellaneous Tool Set

This chapter documents new features of the Miscellaneous Tool Set. The complete reference to the Miscellaneous Tools is in Volume 1, Chapter 14 of the *Apple IIGS Toolbox Reference.*

New information

- ClearHeartBeat and DeleteHeartBeat will turn off the interrupts that occur every 60th of a second if the following conditions are satisfied:

1) There are no remaining heartbeat tasks
2) The interrupt handler installed in IRQ.VBL is the standard system interrupt handler, i.e. no other interrupt handlers have been installed
3) The built standard mouse is not running in VBL interrupt mode.

Chapter 13

Note Sequencer

This chapter documents the Note Sequencer. This is new documentation , not previously presented in the *Apple IIGS Toolbox Reference*.

About the Note Sequencer

The Note Sequencer is a collection of routines that implement a sequencer in the Apple IIGS. This sequencer is designed to play melodies using data stored in a specific format. It does not provide the means to create these data structures, and so an application must provide its own tools for building new sequences.

The Note Sequencer works with the Note Synthesizer and the Sound Tool Set, and it can work with the MIDI Tools if you choose. It can also send control output directly to a MIDI interface device. Use of the Note Sequencer version 1.3 requires the Note Synthesizer version 1.3 or later, and the Sound Tools version 2.4 or later.

◊ *Note:* The Note Synthesizer and the Note Sequencer refer to the software tools provided with the Apple IIGS, not to any separate instrument or device. The MIDI Tools are software tools for use in controlling external instruments, which may be connected through a MIDI interface device.

The Note Sequencer runs during interrupts, and so can run in the background while other application tasks take place in the foreground. Because of this, interrupts must be enabled while a sequence is being played. Any activity that disables interrupts interferes with execution of a sequence. Disk access, for example, disables interrupts, so an application cannot simultaneously gain access to a disk and play a sequence using the Note Sequencer.

The Note Sequencer provides means a for synchronization, but an application that needs a very fine degree of control over the Note Sequencer's timing can directly control it by using the StepSeq call.

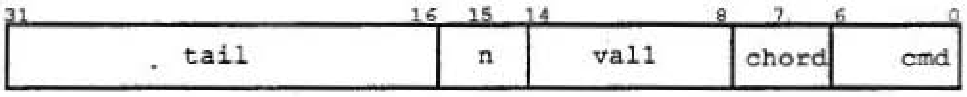
The Note Sequencer's command interpreter

The Note Sequencer is actually a command interpreter. The commands it interprets are 32-bit data structures called *seqItems*, or **Sequence items**. These 32-bit items contain information that the Note Sequencer needs to classify them as note commands, control commands, MIDI commands, or register commands, and to execute them properly.

The format of a seqItem is detailed in figure 13-1.

Figure 13-1 seqItem format

bit number:



cmd For all commands except Note commands, this is the command identifier, a 7-bit number which uniquely identifies the command.

ch The chord bit is a Boolean value. If set, it allows the Note Sequencer to play more than one note simultaneously.

val1 Val1 is a data field whose meaning depends on the command being issued.

n The note bit identifies Note commands. If bit n is set, the seqItem is a note command.

tail The format of the tail field depends upon the command type. It contains two or more fields with command-specific information in them.

Patterns and phrases

A **pattern** is any sequence of seqItems. The Note Sequencer plays melodies by carrying out the seqItem commands in specified patterns. A **phrase** is an ordered set of pointers to patterns or to other phrases. Since a phrase can contain pointers to other phrases, it is possible to nest phrases. The Note Sequencer supports up to 12 levels of such nesting.

When a program calls the Note Sequencer to play a sequence, it passes a parameter containing a handle to the first byte of the top-level phrase. This phrase consists of an ordered series of pointers to the patterns or phrases to be played, followed by *phraseDoneFlag* , a longword value (= $FFFFFFFF) that marks the end of a pattern or phrase. The last seqItem in any phrase or pattern is always *phraseDoneFlag*.

Each pattern consists of an ordered series of seqItems. They can be any valid seqItem, and describe the characteristics of each note to be played in the sequence. Control and Register Commands also allow the characteristics of the notes to be modified, and allow the programmer to build complex sequences by using conditional looping and branching. In this sense, the Note Sequencer is a simple programming language.

Both patterns and phrases are arrays of longwords that end with the flag value $FFFFFFFF. They are distinguished by a 4-byte header.

A phrase is identified by the header

```
dc  i2'0001'   ; low word
dc  i2'0000'   ; high word
```

The phrase body consists of a series of pointers. They can each point either to other phrases or to patterns, which are sequences of executable seqItems. For example:

```
dc  i4'phrase1'
dc  i4'pattern1'
dc  i4'phrase2'
```

A phrase always ends with *phraseDoneFlag*:

```
dc  i4'$FFFFFFFF'
```

A pattern is identified by the header:

```
dc  i2'0000'     ; low word
dc  i2'0000'     ; high word
```

The body of a pattern consists of seqItems, such as:

```
dc  i4'$880ABC74'     ; play C4, duration=10, volume=115
dc  i4'$880ABE74'     ; play D4, duration=10, volume=115
dc  i4'$880AB074'     ; play E4, duration=10, volume=115
```

Again, the pattern must end with *phraseDoneFlag*:

```
dc  i4'$FFFFFFFF'
```

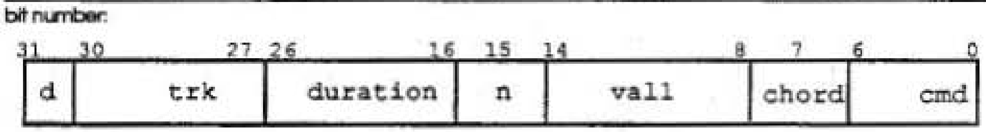
Sequence Items

There are four types of SeqItems: Note Commands, Control Commands, MIDI Commands, and Register commands. Each type is organized in the same way, but the values in each part of the data structure have different meanings in the different commands.

Note commands

Note commands switch notes on and off. You can use note commands in two ways. You can issue a pair of NoteOn and NoteOff commands, turning a specified note on at a certain point, and then explicitly turning it off, or you can issue a NoteOn command with a duration specified. In this case the Note Sequencer plays the note for a number of ticks equal to the value of the duration parameter, then turns the note off without the need for an explicit NoteOff command. Each tick occurs at an interval set by the Note Synthesizer's update rate (see Note Synthesizer).

Figure 13-2 Note command format



Bits 0-6 Note volume. Corresponds to MIDI velocity. A value of zero indicates a NoteOff command.

Bit 7 Chord bit. Indicates that the seqItem is to be played simultaneously with the next seqItem. Do not set both the chord bit and the delay bit in the same item.

Bits 8-14 Pitch. Selects the pitch to be played. Values may range from 0 to 127. A value of 60 selects middle C (261.6 Hz). Adjacent values are one semitone apart.

Bit 15 Note bit. Is always set for Note commands. If this bit is not set in a seqItem, then the seqItem is not a note command.

Bits 16-26 Duration. Specifies the length of time the Note Sequencer is to play the note. Values may range from 0 to 2047, and specify the number of ticks the note is to be played.

A duration of zero identifies the seqItem as a NoteOn command. A NoteOn seqItem is played continuously until the Note Sequencer finds a matching NoteOff.

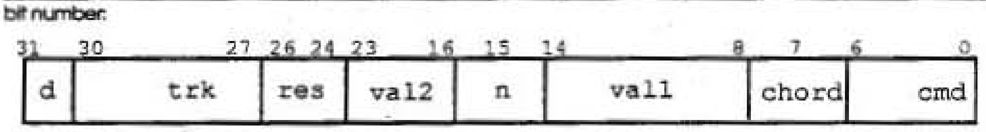
Bits 27-30 Track Number. assigns notes to synthesizer voices and to handlers by specifying their track numbers. Values from $00 to $0F are legal.

Bit 31 Delay Bit. If this bit is set, the Note Sequencer must finish playing this SeqItem before beginning to play the next one. The Note Sequencer cannot advance to the next seqItem until the duration specified by bits 16–26 is past.

Control Commands

Control commands are used to specify the characteristics of the Note Sequencer as it is playing the notes. They can control pitch bend, tempo, vibrato, and the sequence of patterns that is played.

Figure 13-3 Control command format



Bits 0–6 Command number.

Bit 7 Chord bit. The chord bit should be set in a control command. A cleared chord bit can sometimes cause unwanted delays in playing a sequence.

Bits 8–14 Val1. This field contains data specific to each command.

Bit 15 Note bit. Always clear this bit for Control commands. A set Note bit causes the seqItem to be processed as a Note command instead of a Control command.

Bit 16–23 Val2. This field contains data specific to each command.

Bits 24–26 Reserved for control field. These bits should always be clear unless otherwise specified.

Bits 27–30 Track number. Notes are assigned to synthesizer voices and to handlers by specifying their track numbers. Legal values are $00 to $0F.

Bit 31 Delay bit. The delay bit should always be cleared in Control commands, since they have no duration.

Pitch Bend command

cmd	0
chord	1
val1	Pitch wheel position. Values > 64 specify sharp pitch bend; values < 64 specify flat. Intervals are in semitones.
note	0
val2	No significance in the Pitch Bend command. Val2 should always be set to zero for Pitch Bend.
Reserved	Selects pitch bend assignment 0 selects both internal and MIDI pitch bend 1 selects internal pitch bend 2 selects MIDI pitch bend
trk	Track number
d	0

The Pitch Bend command creates a bend effect in a played note. A control command expresses pitch bend as a value from 0 to 127. A value of 64 indicates no pitch bend, and the note is played at the pitch specified in its Note command. The note is played at a pitch determined by its nominal pitch plus the pitch bend sharp or flat. The sequence must use a series of pitch bend commands to achieve the smooth portamento usually associated with a pitch bend.

Tempo command

cmd	1
chord	1
val1	New Increment. The value may vary from 0 to 127.
note	0
val2	0
Reserved	0
trk	0
d	0

This command sets the Note Synthesizer's increment value. The increment value determines the number of ticks between updates in the execution cycle, so larger increments translate to slower tempos.

All Notes Off command

cmd	2
chord	1
val1	0
note	0
val2	0
Reserved	0
Bit 24	0
Bit 25	0
Bit26	0
trk	0
d	0

This command turns off all notes currently being played, overriding any previous Note commands.

Jump command

cmd	3
chord	1
val1	Val1 is the high seven bits of the destination.
note	0
val2	Val2 is the low eight bits of the Jump destination.
Reserved	0
trk	not used.
d	0

The Jump command is the Note Sequencer's equivalent of a jump or goto command in a conventional programming language. Execution of seqItems will continue with the item specified by val1 and val2. The number given is a simple index into the series of seqItems.

Vibrato Depth command

cmd	4
chord	1
val1	The new value for vibrato depth; the value may vary from 0 to 127
note	0
val2	Control number if a MIDI command is generated
Reserved	0: internal and MIDI vibrato
	1: MIDI vibrato
trk	Track number
d	0

The Vibrato Depth command assigns a depth value to the vibrato effect used with the specified track. The vibrato effect is a modulation in the pitch of the voice assigned to the specified track. The Depth value can range from 0 to 127, with larger values resulting in greater vibrato.

Program Change command

cmd	5
chord	1
val1	Instrument number of the new instrument.
note	0
val2	New MIDI program number, if the sequence is using MIDI.
Reserved	Specifies MIDI usage; legal values are
	0 The Apple IIGS internal synthesizer and an external MIDI device.
	1 The Apple IIGS internal synthesizer only
	2 External MIDI device only.
trk	Track number; specifies which instrument program to change by specifying the track to which that instrument is assigned.
d	0

The Program Change command allows a sequence to change the instrument assigned to a track during play. The new instrument must be in the current instrument table for the new assignment to be possible.

Register Commands

Register commands provide the Note Sequencer with program control capabilities. The Note Sequencer maintains 8 psuedo-registers that can be used to implement looping and conditional branching structures. With register commands, and application can achieve the effect of control structures such as "if...then", "do...while" or "repeat...until" in sequences.

Bytes 2 through 9 of the Note Sequencer's direct-page contain the psuedo-registers; these psuedo-registers are number 0 through 7. Each register occupies eight bits of memory, but not all the commands use the full register. The IfGo and Set register commands treat each register as if it were only four bits in size, using only the least significant 4 bits of the byte.

Although the Increment and Decrement register commands act on the full eight bits of each psuedo-register, only the least significant four bits of each psuedo-register should be used. The most significant four bits should always be cleared.

Set Register command

cmd	6
chord	1
val1	low 3 bits contain the register number high 4 bits contain the value
note	0
val2	0
Reserved	0
Bit 24	0
Bit 25	0
Bit26	0
trk	0
d	0

Sets the specified psuedo-register to the specified value.

IfGo Register command

cmd	7
chord	1
val1	low 3 bits contain the register number high 4 bits contain the value
note	0
val2	offset: -128 to +127 seqItems
Reserved	0
Bit 24	0
Bit 25	0
Bit26	0
trk	0
d	0

Tests the specified psuedo-register for the specified value. If the psuedo-register contains the supplied value, then execution continues with the seqItem at the offset specified in val2, calculated from the current seqItem. If the values do not match, execution continues with the next seqItem in sequence. The IfGo command does not check the bounds of the offset provided, so the value must be a valid one, or the effects will be unpredictable.

Inc Register command

cmd	8
chord	1
val1	low 3 bits contain the register number
note	0
val2	0
Reserved	0
Bit 24	0
Bit 25	0
Bit26	0
trk	0
d	0

Increments the value of the specified psuedo-register.

Dec Register command

cmd	9
chord	1
val1	low 3 bits contain the register number
note	0
val2	0
Reserved	0
Bit 24	0
Bit 25	0
Bit26	0
trk	0
d	0

Decrements the value of the specified psuedo-register. If the value is zero when the command is executed, the psuedo-register's value will wrap to $FFFF.

MIDI Commands

MIDI commands allow an executing sequence to send data directly to MIDI devices that are connected to the Apple IIGS. All the standard MIDI commands are provided, except MIDI System Exclusive and MIDI System Common. These commands have been replaced by other commands that allow an executing sequence to send one or two bytes of raw data to the MIDI device.

These commands are based on the MIDI specification, version 1.0, which is not described in this documentation. For further information on the specification, see standard MIDI documentation.

MidiNoteOff command

cmd	10
chord	1
val1	bits 8 through 11 are the channel number
note	0
low	note number
high	velocity

Sends a MIDI NoteOff command on the channel number specified in *val1*. The note turned off is the note specified as a note number in the low byte of the high word, and a velocity in the high byte of the high word.

MidiNoteOn command

cmd	11
chord	1
val1	bits 8 through 11 are the channel number
note	0
low	note number
high	velocity

Sends a MIDI NoteOn command on the channel number specified in *val1*. The note turned on is the note specified as a note number in the low byte of the high word, and a velocity in the high byte of the high word.

MidiPolyphonicKeyPressure command

cmd	12
chord	1
val1	bits 8 through 11 are the channel number
note	0
low	note number
high	key pressure

Sends a MIDI polyphonic key pressure command on the channel number specified in *val1*. The note affected is the note specified as a note number in the low byte of the high word, and its new key pressure is in the high byte of the high word.

MidiControlChange command

cmd	13
chord	1
val1	bits 8 through 11 are the channel number
note	0
low	control number
high	control value

Sends a MIDI control change command to the channel specified in *val1*. The control number is specified in the low byte of the high word and the control's new value is in the hiogh byte of the high word.

MidiProgramChange command

cmd	14
chord	1
val1	0
note	0
low	program number
high	0

Sends a MIDI program change command to the channel specified in *val1*. The program number is specified in the low byte of the high word.

MidiChannelPressure command

cmd	15
chord	1
val1	bits 8 through 11 are the channel number
note	0
low	channel pressure
high	0

Sends a MIDI channel pressure command to the channel specified in *val1*. The new pressure value is specified by the low byte of the high word.

MidiPitchBend command

cmd	16
chord	1
val1	bits 8 through 11 are the channel number
note	0
low	pitch bend least significant byte
high	pitch bend most significant byte

Sends a MIDI pitch bend command to the channel specified by *val1*. The new pitch bend value is specified by the high word of the command, with the least significant byte of the value in the low byte of the high word, and the most significant byte in the high byte.

MidiSelectChannelMode command

cmd	17
chord	1
val1	bits 8 through 11 are the channel number
note	0
low	first data byte
high	second data byte

Sends a MIDI select channel mode command to the channel specified in *val1*. The new MIDI channel mode is specified by two data bytes, the first of which is passed in the low byte of the high word, and the second in the high byte of the high word.

MidiSystemExclusive command

cmd	18
chord	1
val1	0
note	0
low	least significant byte of low word of MIDI packet address
high	most significant byte of low word of MIDI packet address

Passes a pointer to a MIDI packet so that the Note Sequencer can send a MIDI system exclusive command. If MIDI was enabled in the StartSeq call, and if a valid MIDISetSysExlHighWord preceded this command, then the specified MIDI packet will be sent with a MIDI system exclusive command. The MIDISetSysExlHighWord command sends the high word of the MIDI packet's address, and this command sends the low word. The low word of the packet address is passed in the high word of the command.

The MIDI packet format is as follows:

length	word
time stamp	4 bytes
sys. excl.	$F001
data	2 bytes

MidiSystemCommon command

cmd	19
chord	1
val1	bits 8 through 10: low nibble of status byte value varies from 1 through 7 bits 11 and 12: number of data bytes bit 11 set: 1 data byte bit 12 set: 2 data bytes (bit 11 cleared)
note	0
low	first data byte
high	second data byte

Sends one or two bytes of MIDI data. The first data byte is passed in the low byte of the high word and the second data byte, if there is one, is passed in the high byte of the high word.

MidiSystemRealTime command

cmd	20
chord	1
val1	0
note	0
low	real-time number
high	0

Sends a MIDI system real-time command. The real-time number is specified in the low three bits of the low byte of the high word.

MidiSetSysExlHighWord command

cmd	21
chord	1
val1	0
note	0
low	low byte of high word
high	high byte of high word

Specifies the high word of a MIDI packet address to be sent with a MIDI system exclusive command. The MIDI system exclusive command and the low word of the packet address are sent by the MIDISytemExclusive command. The high word of the packet address is passed in the high word of the command.

Using the Note Sequencer

In order to use the Note Sequencer, you must have loaded and started up the following tool sets:

Tool Locator	
Memory Manager	
Sound Tools	version 2.4 or later
Note Synthesizer	version 1.3 or later
MIDI Tools	verson 1.2 or later (if MIDI is to be used)

The Note Sequencer's execution takes place during interrupts. This means that interrupts are disabled when *seqItems* are being executed. It also means that your error handlers and completion routines also run with interrupts disabled.

The Note Sequencer cannot play a semitone value of 0. The reason for this is that 0 is the value reserved for **filler notes**. Filler notes inserted to fill sequences out without affecting what is being played. For example, if an application is playing a complex sequence with several voices, a voice may need to rest for some time while other voices play. The application could use a delay to pause the voice, but during a delay the Note Sequencer does not check other notes to see if they should be turned off or otherwide serviced. This could be a problem if another note changed during the delay. An alternative solution is to use filler notes to fill out the needed time instead of using a delay command. The fillers take up space in the sequence, but do not change anything; they neither start nor stop any notes, and they do not prevent notes in other tracks from being serviced.

The Jump command does not check for bounds errors, it simply causes execution of a sequence to jump directly to the item specified. If the value is erroneous, then execution could jump to any arbitrary place. Needless to say, this would be undesirable.

You might try using allNotesOff and clearIncrement when you want to stop a sequence and be able to start it again easily. A sequence stopped in this way can easily be restarted with a call to set increment.

Sequence timing

Normally we think of a musical sequence as several independent tracks playing at the same time. For example, a musical passage might consist of a violin sound playing a melody accompanied by a viola and a flute. Musically, the three instruments will often play at once, sounding different notes. The Note Sequencer, however, always plays notes in sequence, one after another, however many instruments it is using to play them.

A chord, which is musically a group of different notes played at the same time, is executed by the Note Sequencer as a series of discrete notes played very quickly one after the other. For example, the Note Sequencer would play a chord consisting of F above middle C, A above middle C, and C one octave above middle C as a series of NoteOn commands:

```
NoteOn   F4    4 counts
NoteOn   A4    4 counts
NoteOn   C5    4 counts
```

If the Note Sequencer waited for each note to finish before beginning the next one, the resulting passage would be three distinct notes of equal length, which is not what was intended. The Note Sequencer therefore provides a way to play the three notes with very little delay between them; so little, in fact, that they sound as though they were being played all at once.

If the chord bit is set in a note command, it indicates that the next note should be played to chord with the current one. If, on the other hand, the delay bit is set, it indicates that the current note must be completed before the next one is played.

Using MIDI with the Note Sequencer

The appropriate calls must be made to the MIDI Tool Set to use MIDI with the Note Sequencer. Specifically, the MIDI Tools must be started up, a device driver must be selected, and a MIDI output buffer must be allocated.

You must specify whether MIDI is to be used when you start up the Note Sequencer. If the high bit of the *mode* parameter is set when the SeqStartUp call is made, then MIDI is enabled. In order for a particular track to use MIDI, it must be enabled for that track, using the SetTrkInfo call. Finally, the Note Sequencer checks tool call-specific and seqItem-specific flags for MIDI information, so that individual tool calls or commands can enable or disable MIDI.

If all the appropriate flags, the mode flag, the track flag, and the command or tool call flag are enabled, then MIDI commands are sent to external MIDI devices. This arrangement is designed to provide flexibility in execution. You could, for example, play only the drum parts of a sequence on external MIDI instruments by enabling MIDI output only on the appropriate tracks, or you could play all parts on external MIDI instruments. Switching between the two modes of play would not require any modification of the sequence itself.

A sample sequence

The following example is a sequence presented in 65816 assembly language:

```
Delay    equ  $80000000
T1       equ  $08000000
T2       equ  $18000000
qtr      equ  $40000
hlf      equ  $80000
Note     equ  $8000
C4       equ  $3000
D4       equ  $3E00
F4       equ  $4100
G4       equ  $4300
Chord    equ  $80

phrhndl  dc   i4'phr1'
phr1     dc   i4'01'                            ; it's a phrase
         dc   i4'phr2'
         dc   i4'pat1'
         dc   i4'phr2'
         dc   i4'pat1'
         dc   i4'pat2'
         dc   i4'$FFFFFFFF'                     ; end of phrase 1

phr2     dc   i4'01'                            ; it's a phrase
         dc   i4'pat2'
         dc   i4'pat1'
         dc   i4'$FFFFFFFF'                     ; end of phrase 2

pat1     dc   i4'00'                            ; it's a pattern
         dc   i4'Delay+T1+qtr+Note+C4+115'
         dc   i4'T1+qtr+Note+C4+Chord+115'
         dc   i4'Delay+T2+qtr+Note+G4+115'
         dc   i4'Delay+T1+hlf+Note+F4+115'
         dc   i4'                               ; end of pat1

pat2     dc   i4'00'                            ; it's a pattern
         dc   i4'T1+Note+G4+Chord+115'          ; NoteOn
         dc   i4'Note+hlf'                      ; filler note
         dc   i4'Delay+T2+qtr+Note+F4+115'
         dc   i4'Delay+T2+qtr+Note+D4+115'
         dc   i4'T1+Note+G4+Chord+115'          ; NoteOff
         dc   i4'$00000002'                     ; AllNotesOff
         dc   i4'$FFFFFFFF'                     ; end of pat2
```

Note Sequencer calls

All the tool call documentation for the Note Sequencer is new in this Update.

SeqBootInit $011A

Initializes the Note Sequencer. This call must not be made by an application.

Parameters

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

C

```
extern pascal void  SeqBootInit() inline(0x011A,dispatcher);
```

SeqStartUp $021A

SeqStartUp starts up the Note Sequencer and performs all the necessary initializations for the tool set. It also makes startup calls to the Sound Tools and the Note Synthesizer, so an application should not start up those tool sets before making this call.

Parameters

Stack before call

```
|- previous contents -|
|-    dPageAddr      -|  Word—Location of Note Sequencer direct page
|-      mode         -|  Word—MIDI flag
|-   updateRate      -|  Word—Rate of interrupt generation
|-   increment       -|  Word—Number of interrupts per tick-count
|-                   -|  <-SP
```

Stack after call

```
|- previous contents -|
|-                   -|  <-SP
```

Errors

$1A03 StartedErr; the Note Sequencer has already been started
$1A07 SeqNSWrngVer; the Note Synthesizer is the wrong version

C

```
extern pascal void  SeqStartUp() inline(0x021A,dispatcher);
```

SeqStartUp parameters

The *updateRate* parameter specifies how often the Note Sequencer will update its actions, using interrupts. For example, an *updateRate* value of 500 specifies that the Note Sequencer will receive interrupts at 200 Hz, or every 5 milliseconds. A value of 250 means that interrupts will be available at 100 Hz, or every 10 milliseconds. The same rate is used by the Note Synthesizer to update its instruments' envelopes.

The *increment* parameter specifies how many interrupts constitute one tick of the Note Sequencer counter. If *updateRate* is 500 and *increment* is 20, then one tick will take 100 milliseconds. The Note Sequencer gets interrupts every 5 milliseconds, and the counter is incremented every 20 interrupts. If a quarter note equals 5 ticks, then it lasts half a second, which corresponds to a tempo of 120 beats per minute. In general, the number of beats per minute can be computed using the following formula:

$$B = (24 * \textit{updateRate}) / (\textit{increment} * T)$$

Where B is beats per minute and T is the number of ticks in a beat.

Larger values for *updateRate* result in greater control of a sequence's tempo and smoother envelopes. On the other hand, a higher *updateRate* also requires more processor time to service.

One general method for choosing an appropriate *updateRate* value is to decide on the shortest note you will want to play. Suppose the shortest note that you want to play is a sixteenth note. Assign sixteenth notes a value of 1. Eighth notes are twice as long, so assign them a value of 2. Quarter notes then receive a value of 4, half notes 8, and whole notes 16. Now decide how long you want a whole note, or a quarter note, or a sixteenth note to be and compute the *updateRate* and *increment* so that the duration comes out the way you want it.

Once you have set the *updateRate* value, it remains in effect; you can't change it without shutting down and restarting the Note Sequencer. On the other hand, you can change the *increment* value, and the Note Sequencer provides tempo calls that vary the tempo for you.

SeqShutDown $031A

Shuts down the Note Sequencer tool set. It frees any buffers that the tools may have allocated. An application that uses the Note Sequencer should call SeqShutDown before quitting.

Parameters

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

Errors

$1A05 noStartErr; the Note Sequencer has not been started

C

```
extern pascal void  SeqShutDown() inline(0x031A,dispatcher);
```

SeqVersion $041A

Returns the version number of the Note Sequencer tools that is currently in use in the standard Toolbox version number format.

Parameters

Stack before call

```
|_ previous contents _|
|_      space        _|   Word—Space for result
|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|
|_    versionNum     _|   Word—Version number of the Note Sequencer
|_                   _|   <-SP
```

C

```
extern pascal Word  SeqVersion() inline(0x041A,dispatcher);
```

SeqReset $051A

Resets the Note Sequencer. SeqReset is called when the Apple IIGS System is reset. All internal notes presently being played are turned off. An application should not make this call.

Parameters

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

C

```
extern pascal void  SeqReset() inline(0x051A,dispatcher);
```

SeqStatus $061A

Returns a Boolean flag indicating whether or not the Note Sequencer is active. If the tool set is active, the flag is nonzero, otherwise it is zero.

Parameters

Stack before call

```
|_ previous contents _|
|_      space       _|  Word—Space for result
|_                  _|  <-SP
```

Stack after call

```
|_ previous contents _|
|_    activeFlag    _|  Word—Boolean; TRUE if the Note Sequencer is active
|_                  _|  <-SP
```

C

```
extern pascal Boolean  SeqStatus() inline(0x061A,dispatcher);
```

ClearIncr $0A1A

ClearIncr sets the Note Sequencer's increment value to zero, halting the current sequence, and returns the previous increment value. Setting the increment to zero does not disable the Note Sequencer's interrupts, so envelopes are still updated. This means that, while the sequence will not progress, notes being played when the increment was set to zero may hang. This call is only valid while a sequence is playing.

Parameters

Stack before call

```
|- previous contents -|
|-       space       -|   Word—Space for result
|-                   -|   <-SP
```

Stack after call

```
|- previous contents -|
|-       result      -|   Word—Old value of increment
|-                   -|   <-SP
```

C

```
extern pascal Word  ClearIncr() inline(0x0A1A,dispatcher);
```

GetLoc $0C1A

GetLoc returns certain information about the sequence that is playing. It provides an index to the seqItem that is executing, the current pattern, and the nesting level. The nesting level indicates how deeply control has passed into a structure with phrases nested within phrases. A nesting level value of 0 indicates that the Note Sequencer is playing the top-level phrase. GetLoc is valid only while a sequence is playing.

Parameters

Stack before call

```
|_ previous contents _|
|_      Space        _|   Word—Space for result
|_      Space        _|   Word—Space for result
|_      Space        _|   Word—Space for result
|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|
|_   curPhraseItem   _|   Word—Current item in phrase
|_    curPattItem    _|   Word—Current item in pattern
|_     curLevel      _|   Word—Current nesting level of phrase
|_                   _|   <-SP
```

C

```
extern LocRec GetLoc();
```

GetTimer $0B1A

GetTimer returns the value of the Note Sequencer's tick counter. While the counter is advancing, the value returned is necessarily somewhat inexact, since the value changes as the call is executed. The call is valid only while a sequence is playing.

Parameters

Stack before call

```
|- previous contents  -|
|-       space        -|   Word—Space for result
|-                    -|   <-SP
```

Stack after call

```
|- previous contents  -|
|-       result       -|   Word—Current timer value
|-                    -|   <-SP
```

C

```
extern pascal Word  GetTimer() inline(0x0B1A,dispatcher);
```

SeqAllNotesOff $0D1A

SeqAllNotesOff switches off all notes that are playing but does not stop the sequence. Thus, any notes that are held are turned off, but the sequence continues. Use thiscall to use to temporarily silence all instrument voices while a sequence is active. If the high bit of the *mode* parameter is set, then the Note Sequencer also turns off all external MIDI notes of which it is aware.

Parameters

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

C

```
extern pascal void  SeqAllNotesOff()
inline(0x0D1A,dispatcher);
```

SetIncr $091A

SetIncr sets the Note Sequencer's increment value. An application can use this facility to control the tempo of a sequence. The call only functions while a sequence is playing, but there is also a seqItem Control command, which allows the programmer to set the increment value, so that a sequence can be written with a specified tempo. If the increment value is set to zero, the sequence will halt.

Parameters

Stack before call

```
|_ previous contents _|
|_    increment      _|  Word—The desired increment value
|_                   _|  <-SP
```

Stack after call

```
|_ previous contents _|
|_                   _|  <-SP
```

C

```
extern pascal void  SetIncr() inline(0x091A,dispatcher);
```

SetInstTable $121A

Sets the current instrument table to the one specified in instTable.

Parameters

Stack before call

```
|_ previous contents _|

|_                  _|
|_    instTable     _|   Long—Handle to instrument table
|_                  _|   <-SP
```

Stack after call

```
|_ previous contents _|
|_                  _|   <-SP
```

C

```
extern pascal void  SetInstTable() inline(0x121A,dispatcher);
```

Instrument table

The *instTable* parameter is a pointer to an instrument table. The instrument table is a data structure in Apple IIGS memory that contains pointers to one or more instruments. The format of an instrument table is as follows:

```
|_   instNumber     _|   Word—Number of instruments

|_                  _|
|_      inst0       _|   Long—Pointer to instrument 0

|_                  _|
|_      inst1       _|   Long—Pointer to instrument 1
        . . .

|_                  _|
|_      instN       _|   Long—Pointer to instrument N
```

The *instNumber* parameter equals N+1. See Note Synthesizer for more information about instruments.

SetTrkInfo $0E1A

An application should call SetTrkInfo for each track it uses before starting to play a sequence. The call assigns instruments in the current instrument table to logical tracks, and determines the priorities of the instruments so that the Note Sequencer can correctly allocate generators to them.

The application may disable the internal voices of the IIGS for a specified track by issuing this call with the highest bit of the InstIndex parameter set.

You must use SetInstTable to assign instruments to their respective tracks before issuing this call.

Parameters

Stack before call

```
|- previous contents -|
|-     priority      -|   Word—Desired priority
|-     instIndex     -|   Word—Index number of instrument
|-     trackNum      -|   Word—Track number to assign specified instrument
|-                   -|   <-SP
```

Stack after call

```
|- previous contents -|
|-                   -|   <-SP
```

Errors

$1A01 InstBoundsError; instrument number out of range

C

```
extern pascal void  SetTrkInfo() inline(0x0E1A,dispatcher);
```

StartInts $131A

Enables interrupts. Use this call to restore normal functioning after a call to StopInts.

Parameters

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

C

```
extern pascal void  StartInts() inline(0x131A,dispatcher);
```

StartSeq $0F1A

Starts interpretation of a series of seqItems stored at the address specified by *sequence.*

Parameters

Stack before call

```
|_ previous contents _|

|_                   _|
|_ errHndlrRoutine   _|   Long—Pointer to error handler

|_                   _|
|_   compRoutine     _|   Long—Pointer to completion routine

|_                   _|
|_     sequence      _|   Long—Handle to buffer containing sequence

|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|

|_                   _|   <-SP
```

Errors

$2007	miNoBufErr; no MIDI output buffer is allocated
$1A05	NoStartErr; the Note Sequencer has not been started up

C

```
extern pascal void  StartSeq() inline(0x0F1A,dispatcher);
```

StepSeq $101A

Advances the Note Sequencer to the next seqItem in the current sequence, executing the current seqItems. A StepSeq call is the equivalent of one tick of the Note Sequencer counter.

Parameters

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

Errors

$1921 NoneAvailable; no seqItem available

$1924 AlreadyOn; note specified for NoteOn already on

$1A01 NoCommandErr ; invalid seqItem

$1A02 NoRoomErr; insufficient memory to continue

$1A04 NoNoteErr; note specified for NoteOff not available

C

```
extern pascal void  StepSeq() inline(0x101A,dispatcher);
```

StopInts $141A

Disables Note Synthesizer and Note Sequencer interrupts. If the Note Sequencer is started up, and interrupts are enabled, the Note Synthesizer calls the Note Sequencer interrupt handler whenever an interrupt occurs. When no notes are being played, the overhead involved in this processing is unnecessary, so StopInts provides a way to cause the Note Synthesizer not to service the interrupts. To restart interrupt processing, use the StartInts call.

The StartSeq call starts interrupt processing automatically, and the SeqShutDown automatically halts it. No other Note Sequencer call affects interrupt processing except StopInts and StartInts.

Parameters

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

C

```
extern pascal void  StopInts() inline(0x141A,dispatcher);
```

StopSeq $111A

Halts interpretation of a series of seqItems. The ***next*** parameter specifies whether there are more seqItems to be executed. If so, that is, if ***next*** is nonzero, the next phrase begins. Otherwise, the sequencer simply stops.

Parameters

Stack before call

```
|_ previous contents _|
|_       next        _|   Word—Boolean; TRUE if there are more seqItems
|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|
|_                   _|   <-SP
```

C

```
extern pascal void  StopSeq() inline(0x111A,dispatcher);
```

Chapter 14

Note Synthesizer

This chapter documents the Note Synthesizer. This is new documentation, not previously presented in the *Apple IIGS Toolbox Reference.*

About the Note Synthesizer

The Note Synthesizer is a tool set that controls operation of the Ensoniq Digital Oscillator Chip. With it, an application can turn the Apple IIGS into a digital synthesizer suitable for playing music and generating sound effects. The Note Synthesizer differs from the free-form synthesizer of the Sound Tool Set in that it is specifically designed to help you produce *musical* sounds. It includes utilities to help you to design **waveforms,** which determine the sound quality of synthesizer output, and **instruments**, which are data structures that specify certain things about the output sounds, so that you can store particular sound qualities for reuse.

Use of the Note Synthesizer version 1.3 requires the Sound Tools version 2.4 or later.

Instruments

The Note Synthesizer's basic functional unit is an **instrument**. This is a data structure stored somewhere in the memory of the Apple IIGS that defines the sound quality of a played note. When a program makes the NoteOn call it passes a parameter containing a pointer to an instrument, and that instrument is used to generate the note.

Table 14-1: Instrument data structure

Offset	**Field Name**	**Size**
0	Envelope	24 bytes
24	ReleaseSegment	1 byte
25	PriorityIncrement	1 byte
26	PitchBendRange	1 byte
27	VibratoDepth	1 byte
28	VibratoSpeed	1 byte
29	Spare	1 byte
30	AWaveCount	1 byte
31	BWaveCount	1 byte
32	AWaveList	AWaveCount*6 bytes
X*	BWaveCount	BWaveCount*6 bytes

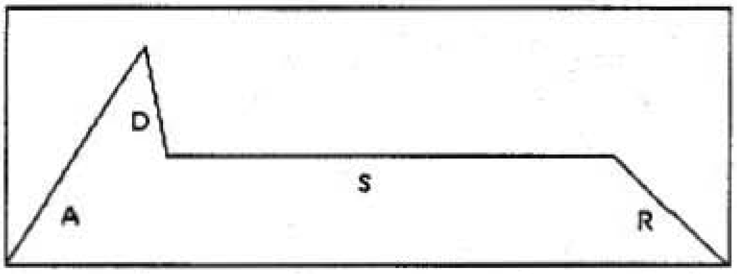
*X= (32+AWaveCount+6)

Envelope

The **envelope** describes the shape of the sound that the Note Synthesizer generates. A note's envelope is what gives it its dynamic quality. A short, sharp sound has a steep, short envelope, and a long, smooth sound has a flatter, longer envelope.

A synthesizer's envelope is traditionally described in terms of **attack**, **decay**, **sustain**, and **release**, or **ADSR**.

Figure 14-1: ADSR



The **Attack** portion of an envelope is the period when the sound is increasing from silence to its peak loudness. This part of the envelope determines the suddenness of a sound. A drumbeat or a plucked string has an extremely steep attack, whereas a bowed string or a softly blown wind instrument has a much flatter attack.

The **Decay** part of the envelope is the period when the sound falls off from its peak loudness to the level it stays at, which is its **sustain** portion. Attack and decay together can be used to control a sound's percussiveness. Sounds with a steep attack and decay tend to sound plucked or percussive. A steep attack followed by a flat decay, or by little or no decay, blare like a loud trumpet. A very flat attack and decay produce a sound with a soft, smooth quality.

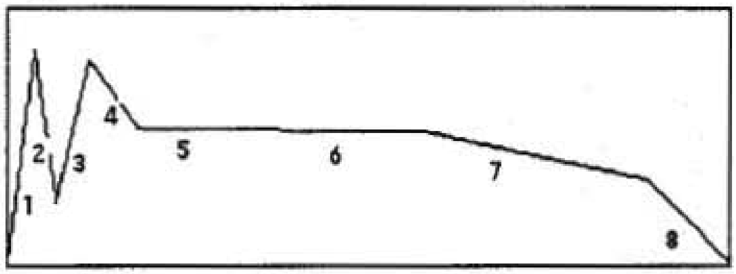
Sustain determines the note's overall perceived loudness and duration. A drumbeat has virtually no sustain or release; it consists almost entirely of attack and decay. A long, slow note on a violin, on the other hand, might have a very flat attack and decay, and a long, high sustain.

The **Release** is the portion of a note as it dies away. Long releases can produce a nice ringing quality, but can also be a problem if a note is still sounding when another note starts, and is dissonant with the first note .

Note Synthesizer envelopes

The envelope definition in the Note Synthesizer's instrument record is somewhat more complex than this simple four-part scheme. The instrument's envelope field can specify up to eight segments instead of just four, so more complex sequences of attack, decay, sustain, and release are possible. For example, the physical properties of pianos cause them to have a complex envelope with two attack segments. A simple ADSR is therefore limited in its ability to simulate a piano's envelope. The Note Synthesizer can do better, because its eight envelope segments allow a closer approximation of the piano's actual envelope.

Figure 14-2: Simulating a piano's envelope



Each segment of an instrument's envelope definition is composed of up to eight linear segments. During each segment, the note's loudness varies from its starting value toward its defined breakpoint value. The segments are defined as a series of breakpoints followed by increments. A **breakpoint** is a byte value between 0 and 127 representing a logarithmic loudness scale. A value difference of 16 represents a change of 6 decibels in loudness. The increments represent the amount the volume changes with each of the envelope's update interrupts. The value is a two-byte fixed point number.

Table 14-2: Envelope field

Segment		
1	breakpoint	increment
2	breakpoint	increment
3	breakpoint	increment
4	breakpoint	increment
5	breakpoint	increment
6	breakpoint	increment
7	breakpoint	increment
8	breakpoint	increment

The shape of the envelope is arbitrary; it can be any shape that can be specified in eight segments, so complex envelopes are possible. The last breakpoint, though, should always be zero, so that the note dies away at the end.

The length of time that a segment of the envelope lasts is given by the following formula:

$$T = \frac{(L-N)*256}{I*R}$$

where:

T =	segment's duration
L =	last breakpoint
N =	next breakpoint
I =	increment value
R =	update rate

As an example, for a segment that changes from 30 to 40 with an increment value of 25 and an update rate of 100 cycles per second, the formula becomes

$$T = \frac{(30-40)*256}{25*100} = \frac{2560}{2500} = 1.02 \text{ seconds}$$

Thus, with the given parameters, the specified segment will require 1.02 seconds.

releaseSegment

The *releaseSegment* parameter defines the segment at which release begins. Its value can be any number from 0 to 7, and simply identifies which segment in sequence is the beginning of the release phase of the envelope. The release phase may thus occupy several segments, but the last breakpoint should always be zero.

priorityIncrement

The *priorityIncrement* parameter is a value that is subtracted from the generator's priority value when the envelope reaches its sustain phase. This allows the Note Synthesizer to reallocate generators, giving higher priority to notes which are just starting. When the envelope reaches the release segment, the priority value assigned to its generator is again reduced, this time to half its current value. Thus, the highest priorities are assumed to go to notes that are just starting; notes being sustained are accorded lower priority, and notes in their release phase receive lowest priority. This is just a rule of thumb; the actual priority values depend on the priority that was specified when the generator was allocated.

pitchbendRange

The *pitchbendRange* parameter specifies the maximum pitch bend that is possible on the note. The maximium possible value for a pitch bend is 127; *pitchbendRange* specifies how much the pitch is raised by the pitch bend value of 127. The legal values are 1, 2 and 4.

vibratoDepth

The *vibratoDepth* parameter can be any number from 0 to 127. A depth of zero specifies that there is no vibrato effect on the note. Vibrato is produced by modulating the pitch of the two oscillators that make up a generator, using a triangle wave produced by a **Low Frequency Oscillator (LFO)**. When the *vibratoDepth* parameter specifies that there is to be no vibrato effect, the vibrato mechanism is switched off to save processing time.

vibratoSpeed

The *vibratoSpeed* parameter controls the rate of vibrato. Higher values produce faster vibrato. The actual speed of vibrato effect depends on the update rate.

AWaveCount and BWaveCount

The Note Synthesizer can use sampled or artificially created waveforms to produce its notes. The parameters *AWaveCount* and *BWaveCount* specify the number of waves in the wavelists that follow the wavecounts.

WaveLists

A **waveList** is an array of variable length. The elements of the array are 6-byte structures called **waveforms**. A waveList can contain up to 255 waveforms.

A **waveform** data structure specifies wave data that is intelligible to the Digital Oscillator Chip (DOC), and is stored somewhere in the memory of the DOC.

Table 14-3: A waveform

topKey	1 byte
waveAddress	1 byte
waveSize	1 byte
DOCMode	1 byte
relPitch	word

When the Note Synthesizer plays a note, it examines the topkey field of each waveform in the waveLists until it finds a value that is greater than or equal to the value of the note it is attempting to play. The first waveform it finds with an acceptable topkey value is the one it plays. For this reason, waveforms should be stored in increasing order of topkey value. The last waveform in a wavelist should have a value of 127, the maximum valid pitch value.

Using appropriate topkey values, the Note Synthesizer can be adjusted to custom tunings, such as **just temperment**, or non-Western scales. The topkey values specify the boundaries between neighboring notes.

The *waveAddress* parameter is the high byte of the waveform's address.

The *waveSize* parameter sets the size of the DOC's wave table, and the frequency resolution of the DOC.

The *DOCMode* parameter sets the mode of the Digital Oscillator Chip.

The *waveAddress*, *waveSize*, and *DOCMode* parameters are all DOC register values, and the Note Synthesizer loads them into the appropriate DOC registers when it accesses the waveform data structure.

The *relPitch* parameter is a word value that is used to tune the waveform. The high-byte value is the semitone, and the low byte is fractions of semitones. A value of 1 in the low byte corresponds to 1/256 of a semitone. A wavelist can specify a full range of notes for an instrument with entries for each note that differ only in the *relPitch* field. Such a wavelist would specify an instrument whose timbre is the same for every note; only the pitch is different.

For more information on DOC registers and waveforms, see the DOC specification, available from Ensoniq.

Using the Note Synthesizer

An application that uses the Note Synthesizer must first start it up, then allocate Digital Oscillator Chip generators for its use with AllocGen. It can play musical notes with individual calls to NoteOn and NoteOff for each note that it plays. NoteOn starts a generator, which automatically updates its envelope as it plays to produce the envelope specified by its assigned instrument. When the application calls NoteOff, the Note Synthesizer enters the release phase of the envelope for that voice, and the note begins to die away.

An application that uses the Note Synthesizer to play notes should disable the interrupts produced by any generators that are allocated for note production. The Note Synthesizer responds to interrupts from its oscillators as if they were timer interrupts.

Each generator is a pair of DOC oscillators. There are 32 such oscillators; two of them are reserved for Apple's use. The remaining 30 are paired into 15 generators, which produce the synthesized notes used by the Note Synthesizer. The Note Synthesizer uses one of these generators as a timer, leaving 14 generators for general use. If the MIDI Tool Set is started up and is using the MIDI clock function, another generator is allocated to serve as the MIDI clock, leaving 13 general-purpose generators for application use.

The Note Synthesizer requires that the Sound Tools be loaded and started up, and one page of bank zero memory must be allocated to it for use as direct-page. The Note Synthesizer shares this direct-page space with the Sound Tool Set. The direct-page area is divided into 15 blocks of 16 bytes, called Generator Control Blocks (GCB). The first byte of a GCB is the allocation signature, indicating which synthesizer is using which generator. The definition of the other 15 bytes of a GCB is determined by the requirements of the synthesizer to which it is allocated.

The GCB contains the values of any "knobs" or "controllers" defining the parameters of the voice that it is currently playing. These parameters are as follows:

SynthID	byte	Note Synthesizer ID = 2
GenNum	byte	Generator number ($00 -$0E)
Semitone	byte	Note being played as specified in call
Volume	byte	Output volume as specified in call
Pitchbend	byte	0-127; 64 specifies no pitch bend
VibratoDepth	byte	As specified in instrument definition

Note synthesizer internal variables: 10 bytes

The Note Synthesizer allocates generators to all the different sound tools that may need them. It therefore requires a piority scheme for allocating generators in the event that there is more than one request for the same generator.

A generator's priority may range from 0 through 128. A priority of 0 means the generator is not being used, and will be allocated to any use that requests it. A priority of 128 indicates that the generator is locked and cannot be reallocated. The remaining values in a generator's range are used by the Note Synthesizer to control allocation of generators.

When a generator is allocated, it receives a priority. The Note Synthesizer automatically lowers the priority of a generator that has reached the sustain portion of its envelope, and again when it reaches the release portion. When the note stops, the generator's priority becomes zero. An application specifies a priority when requesting allocation of a generator, so that allocation occurs when a generator is available with a priority lower than that requested.

DOC memory

An application that uses the Note Synthesizer must load any waveforms that it can use into DOC memory with the Sound Tools call WriteRAMBlock. If a zero is placed in the first 256 bytes of DOC memory, it causes the timer oscillator to halt. If the application uses the clock function of the MIDI Tools, then it must not write to the first 256 bytes of DOC memory.

Note Synthesizer calls

All the call descriptions for the Note Synthesizer are new. The tool calls were undocumented in the *Apple IIGS Toolbox Reference.*

NSBootInit $0119

Initializes the Note Synthesizer. An application must not make this call.

Parameters

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

C

```
extern pascal void  NSBootInit() inline(0x0119, dispatcher);
```

NSStartUp $0219

Starts up the Note Synthesizer for use by an application. An application must make this call before it makes any other Note Synthesizer calls except NSStatus. The update rate is the rate at which interrupts are generated to update envelopes and low-frequency oscillations. The value is in units of 0.4 Hz. Reasonable values for this parameter might be from 150 to 500. The default value is 500. Low rates require less overhead, but higher rates generate smoother sounding envelopes and better timing resolution.

The user update routine is a pointer to a routine that is called during every timer interrupt. Sequencer programs are an example of software that might use routines that run during Note Synthesizer interrupts.

Parameters

Stack before call

```
|- previous contents  _|
|-    updateRate      _|   Word—Rate of envelope generation
|-                    _|
|-  userUpdateRtn     _|   Long—Pointer to interrupt routine
|-                    _|   <-SP
```

Stack after call

```
|- previous contents  _|
|-                    _|   <-SP
```

Errors

$1901	AlreadyInit; the Note Synthesizer was already started up
$1902	SoundNotInit; the Sound Tools were not started up
$1925	SoundWrongVer; incompatible version of the Sound Tools

C

```
extern pascal void  NSStartUp() inline(0x0219, dispatcher);
```

NSShutDown $0319

Shuts down the Note Synthesizer and turns off all generators. An application should make this call before quitting.

Parameters

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

Errors

$1923 NotInit; the Note Synthesizer was not started up

C

```
extern pascal void  NSShutDown() inline(0x0319, dispatcher);
```

NSVersion $0419

Returns the version number of the Note Synthesizer . The version number is in the standard format specified by Toolbox version number protocol.

Parameters

Stack before call

```
|_ previous contents _|
|_      space        _|   Word—Space for result
|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|
|_    versionNum     _|   Word—Version number of the Note Synth
|_                   _|   <-SP
```

C

```
extern pascal Word  NSVersion() inline(0x0419, dispatcher);
```

NSReset $0519

Resets the Note Synthesizer. Applications must not make this call.

Parameters

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

C

```
extern pascal void  NSReset() inline(0x0519, dispatcher);
```

NSStatus $0619

Returns a Boolean value indicating whether the Note Synthesizer is active. If the Note Synthesizer is active, NSStatus returns TRUE. Otherwise, the call returns FALSE.

Parameters

Stack before call

```
|- previous contents  _|
|-       space        _|   Word—Space for result
|-                    _|   <-SP
```

Stack after call

```
|- previous contents  _|
|-    startStatus     _|   Word—Boolean; TRUE if the Synthesizer is started
|-                    _|   <-SP
```

C

```
extern pascal Boolean  NSStatus() inline(0x0619, dispatcher);
```

AllNotesOff $0D19

Turns off all Note Synthesizer generators and sets their priorities to zero. It does not affect generators not used by the Note Synthesizer, such as those allocated to the Free-Form Synthesizer.

Parameters

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

C

```
extern pascal void  AllNotesOff() inline(0x0D19, dispatcher);
```

AllocGen $0919

Requests the allocation of a sound generator. Returns a generator number from 0 to 13. The call will reallocate a generator if all generators are allocated and the specified requestPriority exceeds that of one of the previously allocated generators.

Parameters

Stack before call

```
|_ previous contents _|
|_      space        _|  Word—Space for result
|_  requestPriority  _|  Word—Desired generator priority
|_                   _|  <-SP
```

Stack after call

```
|_ previous contents _|
|_     genNum        _|  Word—Number of generator allocated
|_                   _|  <-SP
```

Errors

$1921 NoneAvail; no generators available to allocate

$1923 NotStarted; Note Synthesizer hasn't been started up

C

```
extern pascal Word  AllocGen() inline(0x0919, dispatcher);
```

DeAllocGen $0A19

Sets the named generator's allocation priority to zero. Any subsequent allocation request with a valid requestPriority will then succeed.

Parameters

Stack before call

```
|- previous contents  -|
|-      genNum        -|  Word—Generator number to deallocate
|-                    -|  <-SP
```

Stack after call

```
|- previous contents  -|
|-                    -|  <-SP
```

Errors

$1922 BadGenNum; invalid generator number

C

```
extern pascal void  DeallocGen() inline(0x0A19, dispatcher);
```

NoteOff $0C19

Switches the specified generator to release mode, which causes the note being generated to die out. When the note's volume is zero, the generator's priority is set to zero, and it is considered to be off. The *genNum* and *semitone* should be the same values specified in the corresponding NoteOn call.

Parameters

Stack before call

```
|_ previous contents _|
|_      genNum       _|   Word—Generator number
|_     semitone      _|   Word—Note being played
|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|
|_                   _|   <-SP
```

C

```
extern pascal void  NoteOff() inline(0x0C19, dispatcher);
```

NoteOn $0B19

Initiates the generation of a note on a specified generator. Normally the *genNum* parameter should be a value returned by the AllocGen call. The *semitone* parameter is a, standard MIDI value from 0 to 127, where middle C is designated by the value 60. The *volume* parameter is a value from 0 to 127 that can be treated as synonymous with MIDI velocity. The value is copied into the Generator Control Byte, and is used to scale the note's amplitude. A change of 16 steps in this parameter specifies a change of 6 decibels in amplitude. The *instrumentPtr* parameter is a pointer to an instrument. See the section on the instrument data structure for more information.

‡ *Note:* If the sum of the *volume* parameter and the envelope amplitude is less than 128, then the note will be inaudible because of the 48 decibel dynamic range of the DOC.

Parameters

Stack before call

```
|- previous contents  -|
|-     genNum         -|   Word—Desired generator number
|-     semitone       -|   Word—Desired pitch
|-     volume         -|   Word—Desired volume
|-                    -|
|-   instrumentPtr    -|   LONG—Desired voice
|-                    -|   <-SP
```

Stack after call

```
|- previous contents  -|
|-                    -|   <-SP
```

Errors

$1924 AlreadyOn; the specified note is already being played

C

```
extern pascal void  NoteOn() inline(0x0B19, dispatcher);
```

Example

The following example shows assembly language code that allocates a generator, passes the correct parameters to NoteOn, plays a note, and turns off the note:

```
pushword #0              ;space for GenNum
pushword #64             ;priority of this note
_AllocGen                ;retrieve an allocated
                         ;generator
pla                      ;get the generator
                         ;number
sta GenNum               ;store it

pushword GenNum          ;push
                         ;parameters:generator
pushword Semitone        ;note
pushword #127            ;maximum volume
pushlong #Instrument     ;LONG pointer to
                         ;instrument definition
_NoteOn
```

After some time...

```
pushword GenNum          ;push parameters:
                         ;generator
pushword Semitone        ;note
_NoteOff                 ;turn off the note
```

Chapter 15

Print Manager

This chapter documents new features of the Print Manager. The complete reference to the Print Manager is in Volume 1, Chapter 15 of the *Apple IIGS Toolbox Reference.*

New information

The following functions have been added to the Print Manager:

- The call PrChooser has been completely redesigned. It now has a new user interface, and supports printing over AppleTalk zones.
- Printer.Setup now saves separate settings for direct and network connections to printers, and saves the User Name for use on a network. Old version of the Printer.Setup file are incompatible with these changes, so the Print Manager will delete such files and create a new one in the correct format. Old settings are discarded, and the default settings are used to create the new setup file.
- If the System disk is locked, the Print Manager will not be able to save changes in Chooser settings, and will display a dialog to warn the user of this fact. It will, however, save any changes to the Chooser settings in RAM, so the new settings will remain in effect until the current application quits.
- If the Print Manager attempts to load a driver and finds that it is missing, it will pass control to a routine that determines what call was being made to the driver, pops the parameters off the stack, and returns a "missing driver "error. It will also display an alert asking the user to make sure a printer and port driver are selected.
- In version of the System software after 3.1.1, the PMStartup call does notload any drivers into memory. Drivers are loaded only when they are needed. The Print Manager does not require that the Drivers folder be present, and if it is present, does not require that there be any drivers in it.
- PMStartup checks to see whether the List Manager has been loaded. Since PrChooser uses List Manager calls, PMStartup will load the List Manager if it has not already been loaded.

New calls

Two new Print Manager calls are PMLoadDriver and PMUnloadDriver. With these calls, an application can load or unload a specific printer driver or port driver as needed.

PMUnloadDriver $3413

Unloads the current port driver, printer driver, or both, depending on the input parameter. Legal values for the driver parameter include

0	unload both drivers
1	unload printer driver
2	unload port driver.

Parameters

Stack before call

```
|_ previous contents _|
|_    whichDriver    _|   Word—Printer driver to unload
|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|
|_                   _|   <-SP
```

Errors

$1308 BadParam

C

```
extern pascal void PMUnloadDriver() inline(0x3413,dispatcher);
```

PMLoadDriver $3513

Loads the current printer driver, port driver, or both, depending on the input parameter. The current driver is determined by the settings saved in the Printer.Setup file. Legal values for the driver parameter include

0	load both drivers
1	load printer driver
2	load port driver.

Parameters

$1308 BadParam

Stack before call

```
|- previous contents  -|
|-    whichDriver     -|   Word—Printer driver to load
|-                    -|   <-SP
```

Stack after call

```
|- previous contents  -|
|-                    -|   <-SP
```

C

```
extern pascal void PMLoadDriver() inline(0x3513,dispatcher);
```

Chapter 16

QuickDraw II

This chapter documents new features of QuickDraw II. The complete reference to QuickDraw II is in Volume 2, Chapter 16 of the *Apple IIGS Toolbox Reference.*

Error corrections

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

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

Chapter 17

QuickDraw II Auxiliary

This chapter documents a new call in QuickDraw II Auxiliary. The complete reference to QuickDraw II Auxiliary is in Volume 2, Chapter 17 of the *Apple IIGS Toolbox Reference.*

SpecialRect $0C12

Frames and fills a rectangle in a single call, making separate calls to FrameRect and FillRect unnecessary. The single call to SpecialRect is considerably faster than separate calls to FrameRect and FillRect.

Parameters

Stack before call

```
|_ previous contents _|

|_                   _|
|_      recPtr       _|   Long—Pointer to rectangle to draw

|_    frameColor     _|   Word—Color of rectangle frame

|_     fillColor     _|   Word—Color of rectangle interior

|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|

|_                   _|   <-SP
```

C

```
extern pascal void SpecialRect() inline(0x0C12,dispatcher);
```

Chapter 18

Sound Tool Set

This chapter documents new features of the Sound Tool Set. The complete reference to the Sound Tools is in Volume 2, Chapter 21 of the *Apple IIGS Toolbox Reference.*

New information

This section provides new information about the Sound Tool Set.

- The four sound and music tools, that is, The Note Sequencer, Note Synthesizer, MIDI Tool Set, and Sound Tool Set, work together , and must be compatible versions.
- The Sound Tools now return the same version number and behave identically whether running with old or new versions of the Apple IIGS ROM.
- The routine called by SoundBootInit that initializes the MidiInitPoll vector ($E101B2) has been changed to an RTL.

FFStartSound

There is new information about the FFStartSound call.

Parameter block

waveStart	The starting location of the waveform
waveSize	The smallest waveform that can be played by FFStartSound is one page (256 bytes). A wave_size value of $FFFF plays 65536 pages.
freqOffset	The following formula gives the value of the frequency register: FR=((32*PF)/1645) where FR= frequency register value; PF equals playback frequency in cycles per second.
docBuffer	See the Ensoniq DOC ERS for further information on *docBuffer* and *volSetting* values.
bufferSize	This field assigns a size for the DOC buffer used for the waveform being played. The Sound Tools assign one such buffer for each of the two oscillators used to play a waveform. The second oscillator's buffer is at the address specified by *docBuffer+bufferSize.*
nextWavePtr	This is a 3-byte field which contains the address of the next waveform to be played. If the field's value is zero, then the current waveform is the last waveform to be played.
volSetting	See the Ensoniq DOC ERS for further information on *docBuffer* and *volSetting* values.

New Error Code

$0817 IRQNotAssignedErr; No Master IRQ was assigned

Example code

```
               PEA Gen.mode                  ; Generator/mode word

               Pushlong Pblock               ; Parameter block pointer

               _FFStartSound                 ; Start free-form
                                             ; synthesizer

               ...

Pblock equ *   ; Waveform parameter block

               DC I4 waveStart               ; Waveform start address

               DC I2 waveSize                ; Wave size in pages

               DC I2 freqOffset              ; DOC frequency register
                                             ; value

               DC I2 docBuffer               ; DOC RAM buffer start
               ; address

               DC I2 bufferSize              ; DOC buffer size code
               ; ($00 to $FF)

               DC I4 nextWavePtr             ; Pointer to next waveform
                                             ; parameter block

               DC I2 volSetting              ; DOC volume register
               ; value

               ...

nextWave equ * ; Next waveform parameter     ; block

               ...
```

Error corrections

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

- The documentation of the FFSoundDoneStatus call includes an error. You will note that the paragraph that describes the call does not agree with the "Stack after call" diagram. The text states that the call returns TRUE if the specified sound is still playing, while the diagram states that it returns FALSE if still playing. The diagram, not the text, is correct.

- There is also an undocumented distinction between a generator that is playing a sound and one that is "active." A generator that is playing a sound returns FALSE in response to an FFSoundDoneStatus call. One that is "active" may or may not be playing a sound; the value of its bit is 1 in the flags returned by FFSoundStatus.

Chapter 19

Tool Locator

This chapter documents new features of the Tool Locator. The complete reference to the Tool Locator is in Volume 2, Chapter 24 of the *Apple IIGS Toolbox Reference.*

New information

This section explains new features of the Tool Locator.

- The Tool Locator uses a new algorithm to load tools from disk. It will only load tools from disk if it cannot findd a tool in ROM with a version number as high as the requested version. The Tool Locator makes no assumptions about which tools are in ROM and which are on the System disk.

  For every tool that is to be loaded, the Tool Locator makes a version call. If the version call returns an error because the tool is not present, or the resulting version number is too low, then the tool is loaded from the System disk.

- The Tool Locator no longer unloads all RAM-based tools every time TLShutDown is called. Instead, it returns the system to a default state, set by a new call in the Tool Locator, SetDefaultTPT. This call can make any collection of RAM and ROM tools the default state. The system returns to the default state when TLShutdown is called.

SetDefaultTPT $1601

Sets the default Tool Pointer Table (TPT) to the current TPT. Used to permanently install a tool patch. An application should not make this call.

Parameters

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

C

```
extern pascal void SetDefaultTPT() inline(0x1601,dispatcher);
```

Chapter 20

Window Manager

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

New information

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

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

Alert windows

The new AlertWindow call (see the section "New Window Manager calls" later in this chapter) can be used to create Alerts for presenting the user with important messages. The call does all the work of creating and displaying the window and contents for the Alert, and returns the ID of the button that the user chooses.

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

Size character

Character 1 is the size of the alert window. The character can be 0-9. The meaning of the values is as follows:

Character	Approximate maximum number of characters.		
0	Custom size and position; followed by		
	h1	WORD	x-coordinate of upper left corner
	v1	WORD	y-coordinate of upper left corner
	h2	WORD	x-coordinate of lower right corner
	v2	WORD	y-coordinate of lower right corner
1	30		
2	60		
3	110		
4	175		
5	110		
6	150		
7	200		
8	250		
9	300		

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

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

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

Icon number

The next character is the icon number. The icon number can be 0–9. The meanings of the icon number values are

0. No icon

1. custom icon, followed by

LONG	Pointer to image data
WORD	Width of image data in bytes
WORD	Height of image data in scan lines

2. Stop icon

3. Note icon

4. Caution icon

5. Disk icon

6. Disk swap icon

7-9 are reserved - DO NOT USE THEM

Separator character

The next character is a separator character. The separator can be any character, but cannot appear in the message text or button strings. The separator divides the message from the first button string and button strings from each other. For purposes of standardization, the slash (/) character is recommended.

Message text

The message text follows the first separator character . Any characters allowed by LETextBox2 are allowed in the message text. See the later section "Special characters" for additional functions of message text. The total size of message text, after substitution of strings, is limited to 1000 characters.

Button strings

The first character after the separator at the end of the message string is the beginning of the first button's title. The title can then be followed with either another separator character and another button title, or a string termination character (zero) to end the alert string. A total of three button titles can be included at the end of the alert string. These buttons will be evenly spaced and centered at the bottom of the alert window. The width of each button is the same and is set by the widest button title. The maximum length of button text after substitution of strings is 80 characters.

Termination of Alert string

A zero byte marks the end of the alert string.

Special characters

The following special characters can be embedded in the message text and button strings of an alert. In order for a special character to appear in the text of a button or message, you must enter it twice in the string. For example, if you want "^" to appear in an Alert message, you must enter it in the message string as "^^".

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

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

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

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

Substitution string array

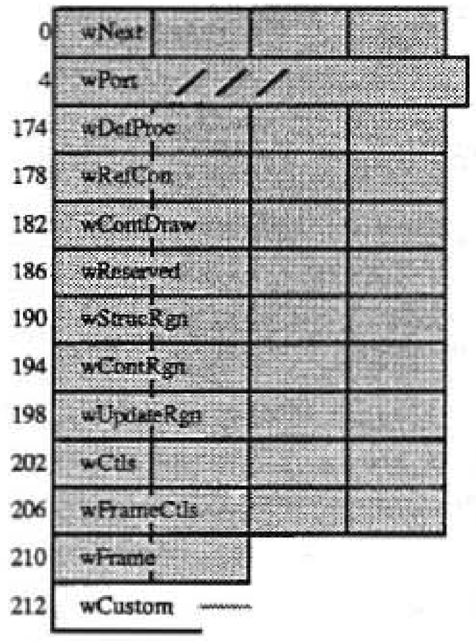
LONG[0]	Pointer to string that will substitute for *0
LONG[1]	Pointer to string that will substitute for *1
LONG[2]	Pointer to string that will substitute for *2
LONG[3]	Pointer to string that will substitute for *3
LONG[4]	Pointer to string that will substitute for *4
LONG[5]	Pointer to string that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

LONG[8]	Pointer to string that will substitute for *8
LONG[9]	Pointer to string that will substitute for *9

Substitution strings can be C strings, Pascal strings, or terminated by a carriage return. C strings and carriage-return terminated strings are selected by passing 0 to **AlertWindow** as the string flag. A value of 1 in the string flag specifies a Pascal string.

Window records

The Window Record data structure has been redefined. The new definition is illustrated here:



LONG - Pointer to next window record, zero is end of list.

BYTE[170] - Window's grafPort.

LONG - Address of window's definition procedure.

LONG - Reserved for application's use.

LONG - Address of routine that will draw window's content.

LONG - Reserved by Window Manager, do not use.

LONG - Handle of window's structure region.

LONG - Handle of window's content region.

LONG - Handle of window's update region.

LONG - Handle of first control in window's content.

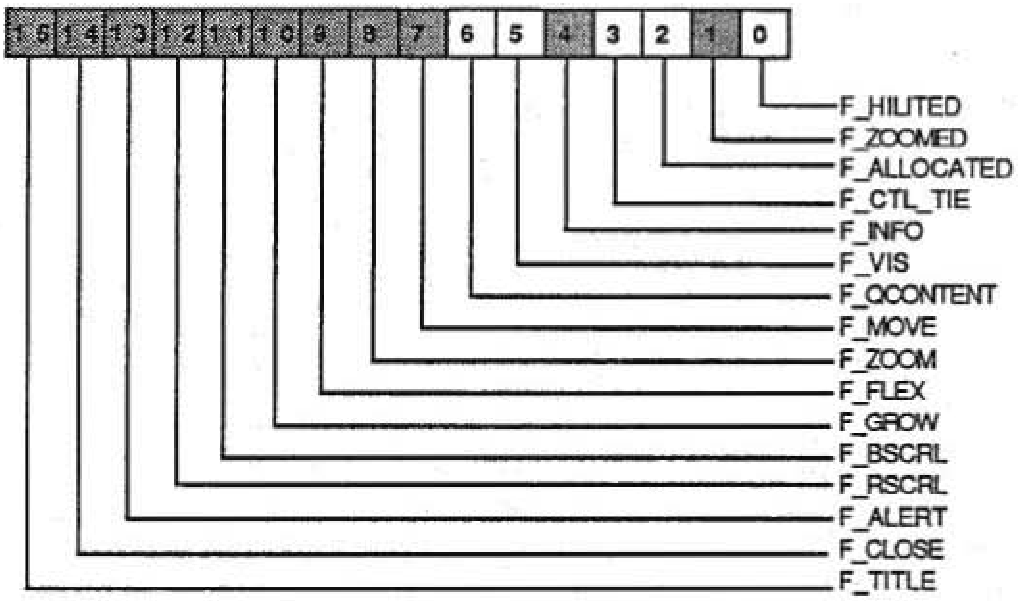
LONG - Handle of first control in window's frame.

WORD - Flags that define window.

BYTE[n] - Additional data space defined by window's defProc.

The wReserved field is a new data field reserved by Apple for future expansion.

The wFrame field is illustrated as follows. The shaded bits in the diagram are for use by window defProcs. The values named in the diagram are those used by the standard document window defProc. Unshaded bits are reserved by the Window Manager and are the same for all windows.



Error corrections

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

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

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

- *Apple IIGS Toolbox Reference* page 25-126, third line:
  If wmTaskMask bit tmInfo (bit 15) = 1

should read:
  If wmTaskMask bit tmInfo (bit 15) = 0

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

New Window Manager calls

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

AlertWindow $590E

Creates an alert window that displays a message pointed to by *alertStrPtr*. The message can be either a C or Pascal string, as specified by *stringType*. A value of 0 signifies that the message is a C string, and a value of 1, that it is a Pascal string. The *subStrPtr* parameter points to an array of substitution strings for use with substitution characters. For more detailed information, see the previous section "Alert Windows" in this chapter.

Parameters

Stack before call

```
|_ previous contents _|
|_      space       _|   Word—Space for result
|_    stringType    _|   Word—0 if C string; 1 if Pascal string
|_                  _|
|_    subStrPtr     _|   Long—Pointer to substitution array
|_                  _|
|_    alertStrPtr   _|   Long—Pointer to alert string
|_                  _|   <-SP
```

Stack after call

```
|_ previous contents _|
|_      result      _|   Word—Button number selected
|_                  _|   <-SP
```

DrawInfoBar $550E

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

Parameters

Stack before call

```
|_ previous contents _|

|_                   _|
|_   grafPortPtr     _|   Long—Pointer to grafPort of window

|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|
|_                   _|   <-SP
```

EndFrameDrawing $5B0E

Restores Window Manager variables after a call to StartFrameDrawing.

Parameters

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

GetWindowMgrGlobals $580E

Returns a pointer to the Window Manager global data area. An application should never make this call.

Parameters

Stack before call

```
|_ previous contents _|
|_                   _|
|_      space        _|   Long—Space for result
|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|
|_                   _|
|_  globalDataPtr    _|   Long—Pointer to the global data area
|_                   _|   <-SP
```

ResizeWindow $5C0E

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

Parameters

Stack before call

```
|- previous contents -|
|-    hiddenFlag     -|   Word—Boolean; whether to hide covered areas
|-                   -|
|-      recPtr       -|   Long—Pointer to window's content region
|-                   -|
|-    grafPortPtr    -|   Long—Pointer to window's grafPort
|-                   -|   <-SP
```

Stack after call

```
|- previous contents -|
|-                   -|   <-SP
```

StartFrameDrawing $5A0E

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

Parameters

Stack before call

```
|_ previous contents _|

|_                   _|
|_    windowPtr      _|   Long—Pointer to window to draw
|_                   _|   <-SP
```

Stack after call

```
|_ previous contents _|
|_                   _|   <-SP
```

Indexes

Index of new tool calls

This index lists new Tool calls alphabetically. These are calls that were not documented in the first edition of the *Apple IIGS Toolbox Reference*. The page number on which a description appears in this Update appears to the right of the call name.

Index of all tool calls

This index lists all the Tool calls alphabetically. Callslisted are from both the *Apple IIGS Toolbox Reference* and from this update. Page numbers in the Reference are on the left, and are hypenated; page numbers from this update are on the right, and are not hyphenated.

A

Call name	Reference	Update
ACEBootInit		11
ACECompBegin		17
ACECompress		18
ACEExpand		20
ACEExpBegin		21
ACEInfo		22
ACEReset		15
ACEShutDown		13
ACEStartUp		12
ACEStatus		16
ACEVersion		14
ADBBootInit	3-10	
ADBReset	3-12	
ADBShutDown	3-11	
ADBStartUp	3-10	
ADBStatus	3-12	
ADBVersion	3-11	
AddFamily	8-23	
AddFontVar	8-24	
AddPt	16-68	
Alert	6-31	
AlertWindow		185
AllNotesOff		147
AllocGen		148
AsyncADBReceive	3-14	

B

Call name	Reference	Update
BeginUpdate	25-35	
BlockMove	12-21	
BringToFront	25-36	
Button	7-31	

C

Call name	Reference	Update
CalcMenuSize	13-33	
CautionAlert	6-35	
CharBounds	16-69	
CharWidth	16-70	
CheckHandle	12-22	
CheckMItem	13-34	
CheckUpdate	25-37	
ChooseCDA	5-12	
ChooseFont	8-26	
ClampMouse	14-30	
ClearIncr		118
ClearMouse	14-31	
ClearScreen	16-71	
ClearSRQTable	3-15	
ClipRect	16-72	
CloseAllNDAs	5-12	
CloseDialog	6-36	
CloseNDA	5-13	
CloseNDAByWinPtr	5-14	
ClosePicture	17-9	
ClosePoly	16-72	
ClosePort	16-73	
CloseRgn	16-74	
CloseWindow	25-38	
ClrHeartBeat	14-53	
CompactMem	12-22	
CopyPixels	17-10	
CopyRgn	16-75	
CountFamilies	8-28	
CountFonts	8-29	
CountMItems	13-35	
CreateList	11-16	
CStringBounds	16-76	
CStringWidth	16-77	
CtlBootInit	4-41	
CtlNewRes	4-45	
CtlReset	4-44	
CtlShutDown	4-43	
CtlStartUp	4-42	
CtlStatus	4-44	
CtlTextDev	23-15	
CtlVersion	4-43	

D

Call name	Reference	Update
DeallocGen		149
Dec2Int	9-8	
Dec2Long	9-9	
DefaultFilter	6-37	
DeleteID	14-59	
DeleteMenu	13-36	
DeleteMItem	13-37	
DelHeartBeat	14-52	
DeskBootInit	5-9	
DeskReset	5-11	
DeskShutDown	5-10	
DeskStartUp	5-9	
DeskStatus	5-11	
Desktop	25-39	
DeskVersion	5-10	
DialogBootInit	6-27	
DialogReset	6-30	
DialogSelect	6-38	
DialogShutDown	6-29	
DialogStartUp	6-28	
DialogStatus	6-30	
DialogVersion	6-29	
DiffRgn	16-78	
DisableDItem	6-40	
DisableMItem	13-38	
DisposeAll	12-23	
DisposeControl	4-45	
DisposeHandle	12-24	
DisposeMenu	13-39	
DisposeRgn	16-79	
DlgCopy	6-41	
DlgCut	6-42	
DlgDelete	6-43	
DlgPaste	6-44	
DoWindows	7-32	
DragControl	4-46	
DragRect	4-48	
DragWindow	25-44	
DrawChar	16-80	
DrawControls	4-54	
DrawCString	16-81	
DrawDialog	6-45	
DrawIcon	17-11	
DrawInfoBar		186
DrawMember	11-17	
DrawMenuBar	13-40	
DrawOneCtl	4-55	
DrawPicture	17-12	

Call name	Reference	Update
DrawString	16-82	
DrawText	16-83	

E

Call name	Reference	Update
EMBootInit	7-27	
EmptyRgn	16-84	
EMReset	7-30	
EMShutDown	7-29	
EMStartUp	7-27	
EMStatus	7-30	
EMVersion	7-29	
EnableDItem	6-46	
EnableMItem	13-40	
EndFrameDrawing		187
EndInfoDrawing	25-47	
EndUpdate	25-47	
EqualPt	16-85	
EqualRect	16-86	
EqualRgn	16-87	
EraseArc	16-88	
EraseControl	4-56	
EraseOval	16-89	
ErasePoly	16-90	
EraseRect	16-91	
EraseRgn	16-92	
EraseRRect	16-93	
ErrorSound	6-47	
ErrWriteBlock	23-17	
ErrWriteChar	23-18	
ErrWriteCString	23-19	
ErrWriteLine	23-20	
ErrWriteString	23-21	
EventAvail	7-33	

F

Call name	Reference	Update
FakeMouse	7-34	
FamNum2ItemID	8-31	
FFGeneratorStatus	21-11	
FFSoundDoneStatus	21-13	
FFSoundStatus	21-14	
FFStartSound	21-15	
FFStopSound	21-18	
FillArc	16-94	
FillOval	16-95	
FillPoly	16-96	
FillRect	16-97	
FillRgn	16-98	
FillRRect	16-99	
FindControl	4-57	
FindDItem	6-48	
FindFamily	8-32	
FindFontStats	8-34	
FindHandle	12-25	
FindWindow	25-48	
Fix2Frac	9-10	
Fix2Long	9-11	
Fix2X	9-12	
FixAppleMenu	5-15	
FixATan2	9-13	
FixDiv	9-14	
FixFontMenu	8-36	
FixMenuBar	13-41	
FixMul	9-15	
FixRatio	9-16	
FixRound	9-17	
FlashMenuBar	13-41	
FlushEvents	7-35	
FMBootInit	8-18	
FMGetCurFID	8-38	
FMGetSysFID	8-39	
FMReset	8-22	
FMSetSysFont	8-40	
FMShutDown	8-21	
FMStartUp	8-19	
FMStatus	8-22	
FMVersion	8-21	
ForceBufDims	16-100	
Frac2Fix	9-18	
Frac2X	9-19	
FracCos	9-20	
FracDiv	9-21	
FracMul	9-22	
FracSin	9-23	
FracSqrt	9-24	
FrameArc	16-101	
FrameOval	16-102	
FramePoly	16-103	
FrameRect	16-104	
FrameRgn	16-105	
FrameRRect	16-106	
FreeMem	12-26	
FrontWindow	25-50	
FWEntry	14-17	

G

Call name	Reference	Update
GetAbsClamp	14-38	
GetAbsScale	3-15	
GetAddr	14-19	
GetAddress	16-107	
GetAlertStage	6-49	
GetArcRot	16-109	
GetBackColor	16-110	
GetBackPat	16-111	
GetBarColors	13-42	
GetCaretTime	7-36	
GetCharExtra	16-112	
GetClip	16-113	
GetClipHandle	16-114	
GetColorEntry	16-115	
GetColorTable	16-116	
GetContentDraw	25-51	
GetContentOrigin	25-52	
GetContentRgn	25-53	
GetControlDItem	6-50	
GetCtlAction	4-59	
GetCtlDPage	4-60	
GetCtlParams	4-61	
GetCtlRefCon	4-62	
GetCtlTitle	4-63	
GetCtlValue	4-64	
GetCursorAdr	16-117	
GetDAStrPtr	5-16	
GetDataSize	25-54	
GetDblTime	7-37	
GetDefButton	6-51	
GetDefProc	25-55	
GetDItemBox	6-52	
GetDItemType	6-53	

Call name	Reference	Update
HomeMouse	14-32	
HUnlock	12-32	
HUnlockAll	12-33	
I		
IMBootInit	9-5	
IMShutDown	9-6	
IMStartUp	9-5	
IMStatus	9-7	
IMVersion	9-6	
InflateTextBuffer	16-157	
InitColorTable	16-158	
InitCursor	16-160	
InitMouse	14-32	
InitPalette	13-56	
InitPort	16-161	
InitTextDev	23-28	
InsertMenu	13-57	
InsertMItem	13-58	
InsetRect	16-162	
InsetRgn	16-163	
InstallCDA	5-18	
InstallFont	8-43	
InstallNDA	5-19	
InstallWithStats		43
Int2Dec	9-29	
Int2Hex	9-30	
IntSource	14-25	
InvalRect	25-80	
InvalRgn	25-81	
InvertArc	16-164	
InvertOval	16-165	
InvertPoly	16-166	
InvertRect	16-167	
InvertRgn	16-168	
InvertRRect	16-169	
IsDialogEvent	6-63	
ItemID2FamNum	8-45	
K		
KillControls	4-68	
KillPicture	17-13	
KillPoly	16-170	
L		
LEActivate	10-16	
LEBootInit	10-12	
LEClick	10-17	
LECopy	10-18	
LECut	10-19	
LEDeactivate	10-20	
LEDelete	10-21	
LEDispose	10-22	
LEFromScrap	10-23	
LEGetScrapLen	10-23	
LEGetTextHand	10-24	
LEGetTextLen	10-25	
LEIdle	10-26	
LEInsert	10-27	
LEKey	10-28	
LENew	10-30	
LEPaste	10-32	
LEReset	10-15	

Call name	Reference	Update
LEScrapHandle	10-33	
LESetCaret	10-34	
LESetHilite	10-35	
LESetJust	10-36	
LESetScrapLen	10-37	
LESetSelect	10-38	
LESetText	10-39	
LEShutDown	10-14	
LEStartUp	10-13	
LEStatus	10-15	
LETextBox	10-40	
LETextBox2	10-42	
LEToScrap	10-45	
LEUpdate	10-46	
LEVersion	10-14	
Line	16-171	
LineTo	16-172	
ListBootInit	11-13	
ListReset	11-15	
ListShutDown	11-14	
ListStartup	11-13	
ListStatus	11-15	
ListVersion	11-14	
LoadFont	8-46	
LoadOneTool	24-10	
LoadScrap	20-15	
LoadSysFont	8-48	
LoadTools	24-11	
LocalToGlobal	16-173	
Long2Dec	9-31	
Long2Fix	9-32	
Long2Hex	9-33	
LongDivide	9-34	
LongMul	9-35	
LoWord	9-36	
M		
MapPoly	16-174	
MapPt	16-175	
MapRect	16-176	
MapRgn	16-177	
MaxBlock	12-34	
MenuBootInit	13-29	
MenuGlobal	1359	
MenuKey	13-61	
MenuNewRes	13-63	
MenuRefresh	13-64	
MenuReset	13-32	
MenuSelect	13-66	
MenuShutDown	13-31	
MenuStartUp	13-30	
MenuStatus	13-32	
MenuVersion	13-31	
MessageCenter	24-14	
MidiBootInit		67
MidiClock		73
MIDIControl		76
MidiDevice		81
MidiInfo		83
MidiReadPacket		85
MidiReset		71
MidiShutDown		69
MidiStartUp		68
MidiStatus		72

Call name	Reference	Update
ReadKeyMicroData	3-17	
ReadKeyMicroMem	3-18	
ReadLine	23-30	
ReadMouse	14-34	
ReadRamBlock	21-24	
ReadTimeHex	14-14	
RealFreeMem		51
ReallocHandle	12-41	
RectInRgn	16-203	
RectRgn	16-204	
RefreshDesktop	25-91	
RemoveDItem	6-76	
ResetAlertStage	6-76	
ResetMember	11-21	
ResizeWindow		189
RestAll	5-21	
RestoreBufDims	16-205	
RestoreHandle	12-42	
RestoreTextState	24-16	
RestScrn	5-21	
S		
SANEBootInit	18-11	
SANEDecStr816	18-15	
SANEElems816	18-15	
SANEFP816	18-15	
SANEReset	18-14	
SANEShutDown	18-13	
SANEStartUp	18-12	
SANEStatus	18-14	
SANEVersion	18-13	
SaveAll	5-22	
SaveBufDims	16-206	
SaveScrn	5-22	
SaveTextState	24-17	
ScalePt	16-207	
SchAddTask	19-7	
SchBootInit	19-4	
SchFlush	19-8	
SchReset	19-6	
SchShutDown	19-5	
SchStartUp	19-4	
SchStatus	19-6	
SchVersion	19-5	
ScrapBootInit	20-7	
ScrapReset	20-9	
ScrapShutDown	20-8	
ScrapStartUp	20-7	
ScrapStatus	20-9	
ScrapVersion	20-8	
ScrollRect	16-208	
SDivide	9-38	
SectRect	16-209	
SectRgn	16-210	
SelectIText	6-77	
SelectMember	11-22	
SelectWindow	25-92	
SendBehind	25-93	
SendInfo	3-19	
SeqAllNotesOff		121
SeqBootInit		111
SeqReset		116
SeqShutDown		114
SeqStartUp		112
SeqStatus		117
SeqVersion		115
ServeMouse	14-35	
SetAbsClamp	14-37	
SetAbsScale	3-23	
SetAllSCBs	16-211	
SetArcRot	16-212	
SetAutoKeyLimit		39
SetBackColor	16-213	
SetBackPat	16-214	
SetBarColors	13-69	
SetBufDims	16-215	
SetCharExtra	16-217	
SetClip	16-218	
SetClipHandle	16-219	
SetColorEntry	16-220	
SetColorTable	16-221	
SetContentDraw	25-94	
SetContentOrigin	25-95	
SetContentOrigin2	25-96	
SetCtlAction	4-74	
SetCtlIcons	4-75	
SetCtlParams	4-76	
SetCtlRefCon	4-77	
SetCtlTitle	4-78	
SetCtlValue	4-79	
SetCursor	16-222	
SetDAFont	6-79	
SetDAStrPtr	5-23	
SetDataSize	25-97	
SetDefaultTPT		176
SetDefButton	6-80	
SetDefProc	25-98	
SetDItemBox	6-81	
SetDItemType	6-82	
SetDItemValue	6-83	
SetEmptyRgn	16-223	
SetErrGlobals	23-32	
SetErrorDevice	23-33	
SetEventMask	7-45	
SetFont	16-224	
SetFontFlags	16-225	
SetFontID	16-227	
SetForeColor	16-228	
SetFrameColor	25-99	
SetGrafProcs	16-229	
SetHandleSize	12-43	
SetHeartBeat	14-48	
SetIncr		122
SetInfoDraw	25-101	
SetInfoRefCon	25-102	
SetInGlobals	23-34	
SetInputDevice	23-35	
SetInstTable		123
SetIntUse	16-230	
SetIText	6-84	
SetMasterSCB	16-231	
SetMaxGrow	25-103	
SetMenuBar	13-71	
SetMenuFlag	13-72	
SetMenuID	13-73	
SetMenuTitle	13-74	
SetMItem	13-75	
SetMItemBlink	13-76	

Call name	Reference	Update
SetMItemFlag	13-77	
SetMItemID	13-79	
SetMItemMark	13-80	
SetMItemName	13-81	
SetMItemStyle	13-82	
SetMouse	14-36	
SetMTitleStart	13-84	
SetMTitleWidth	13-85	
SetOrigin	16-232	
SetOriginMask	25-104	
SetOutGlobals	23-37	
SetOutputDevice	23-38	
SetPage	25-105	
SetPenMask	16-223	
SetPenMode	16-234	
SetPenPat	16-236	
SetPenSize	16-237	
SetPenState	16-238	
SetPicSave	16-239	
SetPolySave	16-240	
SetPort	16-241	
SetPortLoc	16-242	
SetPortRect	16-243	
SetPortSize	16-244	
SetPt	16-245	
SetPurge	12-44	
SetPurgeAll	12-45	
SetPurgeStat	8-49	
SetRandSeed	16-246	
SetRect	16-247	
SetRectRgn	16-248	
SetRgnSave	16-249	
SetSCB	16-250	
SetScrapPath	20-17	
SetScroll	25-106	
SetSolidBackPat	16-251	
SetSolidPenPat	16-252	
SetSoundMIRQV	21-25	
SetSoundVolume	21-26	
SetSpaceExtra	16-253	
SetStdProcs	16-254	
SetSwitch	7-46	
SetSysBar	13-86	
SetSysField	16-255	
SetSysFont	16-256	
SetSysWindow	25-107	
SetTextFace	16-257	
SetTextMode	16-259	
SetTextSize	16-261	
SetTrkInfo		124
SetTSPtr	24-19	
SetUserField	16-262	
SetUserSoundIRQV	21-27	
SetVector	14-61	
SetVisHandle	16-263	
SetVisRgn	16-264	
SetWAP	24-20	
SetWFrame	25-108	
SetWindowIcons	25-109	
SetWRefCon	25-110	
SetWTitle	25-111	
SetZoomRect	25-112	
SFAllCaps	22-20	
SFBootInit	22-15	

Call name	Reference	Update
SFGetFile	22-21	
SFPGetFile	22-25	
SFPPutFile	22-27	
SFPutFile	22-30	
SFReset	22-18	
SFShutDown	22-17	
SFStartUp	22-16	
SFStatus	22-19	
SFVersion	22-18	
ShowControl	4-80	
ShowCursor	16-264	
ShowDItem	6-85	
ShowHide	25-113	
ShowPen	16-265	
ShowWindow	25-114	
SizeWindow	25-115	
SolidPattern	16-265	
SortList	11-23	
SoundBootInit	21-7	
SoundReset	21-10	
SoundShutDown	21-9	
SoundStartUp	21-8	
SoundToolStatus	21-10	
SoundVersion	21-9	
SpecialRect		166
SRQPoll	3-25	
SRQRemove	3-26	
StartDrawing	25-116	
StartFrameDrawing		190
StartInfoDrawing	25-117	
StartInts		125
StartSeq		126
StatusID	14-60	
StatusTextDev	23-39	
StepSeq		127
StillDown	7-47	
StopAlert	6-86	
StopInts		128
StopSeq		129
StringBounds	16-266	
StringWidth	16-267	
SubPt	16-268	
SyncADBReceive	3-27	
SysBeep	14-53	
SysFailMgr	14-54	
SystemClick	5-26	
SystemEdit	5-27	
SystemEvent	5-28	
SystemTask	5-29	
T		
TaskMaster	25-118	
TestControl	4-81	
TextBootInit	23-10	
TextBounds	16-269	
TextReadBlock	23-40	
TextReset	23-13	
TextShutDown	23-11	
TextStartUp	23-11	
TextStatus	23-14	
TextVersion	23-12	
TextWidth	16-270	
TextWriteBlock	23-41	
TickCount	7-48	

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh® computers and Microsoft® Word. Proof pages were created on the Apple LaserWriter® Plus. Final pages were created on the Varityper® VT600™. POSTSCRIPT®, the LaserWriter page-description language, was developed by Adobe Systems Incorporated.

Text type is ITC Garamond® (a downloadable font distributed by Adobe Systems). Display type is ITC Avant Garde Gothic®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier, a fixed-width font.