Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#11: Ensoniq DOC Swap-Mode Anomaly

Revised by: Jim Mensch November 1988
Written by: Jim Merritt April 1987

Under certain conditions, the IIGS Ensoniq Digital Oscillator Chip (DOC) inserts a spurious zero-crossing byte into the output sample stream. The output sample waveform may mask the anomaly, but if it does not, the user may hear intermittent clicks or even a more pervasive "static." This Technical Note discusses the situations in which the DOC produces this spurious zero crossing, as well as strategies to avoid or mask this undesirable behavior.

Background

The Ensoniq DOC in the Apple IIGS is actually a microprocessor dedicated to producing sound. Like a time-sharing computer, the DOC continually scans through its array of sound oscillators, proceeding from lower-numbered oscillators to higher-numbered ones, and updates the signal output level of each active one to match that indicated by the oscillator's current sample byte.

An oscillator can operate in any one of several functional modes, as described in the *Apple IIGS Hardware Reference*. Here, however, we are concerned only with swap mode, where two consecutive oscillators are considered as a single generator. The low-numbered oscillator in the pair is always even. For example, the pairs of oscillators 0 & 1, 2 & 3, … , 12 & 13, and 14 & 15 constitute generators. The IIGS Sound Tool Set – the `FFStartSound` call in particular – configures the oscillators it uses to operate in swap mode. In swap mode, the even-numbered oscillator plays its waveform first, halts its own playback, then starts its partner which also plays its waveform, halts its own playback upon exhausting its waveform, and restarts the even-numbered oscillator. At any time between the start of any particular `FFStartSound` call and the time the oscillator finishes playing a wave, the Sound Tool Set interrupt handler may be busy transferring waveform information from the IIGS main RAM to the dormant oscillator's buffer in DOC RAM. Since one oscillator is producing sound while the Sound Tool Set interrupt handler is transferring waveform information to the other oscillator, you can use a generator pair to produce continuous sound of arbitrary length, and you are limited only by the amount of memory you can devote to the waveform in the main RAM.

Each oscillator draws its output samples from a dedicated buffer in DOC RAM, the size and location of which are specified by parameters to the `FFStartSound` call. The maximum size for an oscillator buffer is 32K, but since buffers may neither coincide nor overlap, the practical maximum may be lower when more than one generator is active. For instance, if four generators

(eight paired oscillators) are active simultaneously, the maximum buffer size is 8K, since eight non-overlapping buffers of 8K each would occupy the entire 64K available in the DOC RAM.

The Problem

Whenever a swap occurs from a higher-numbered oscillator to a lower-numbered one, the output signal from the corresponding generator temporarily falls to the zero-crossing level (silence); this anomaly does not occur during swaps from lower-numbered oscillators to higher-numbered ones. The spurious level change lasts no longer than a single sample period, at which time the interrupted waveform resumes. However, even this tiny glitch in the output can be audible as a pop or click; the further away the waveform is from the zero crossing when the swap interrupts it, the louder the ear will perceive the pop or click. When high-to-low swaps occur with great frequency, the pops and clicks happen so often that they are perceived as gentle, but pervasive, static.

Several Workarounds

There is no ideal solution to the problem of signal interruption in swap mode. This problem is an anomaly of the DOC design, which may or may not be addressed in later versions of the chip. However, we have found three general strategies for mitigating the audible damage to the output waveform caused by the chip's undesirable behavior.

Minimize Oscillator Swaps per Unit Time

The more often swaps from high-numbered oscillators to low-numbered ones occur, the more obtrusive the brief signal interruptions will seem. To minimize the interruptions, you must make the oscillators play for a longer period of time before swapping to their partners. This means that they must play at slower output sample rates, use larger buffers in DOC RAM, or use the two in tandem. Commensurate with the number of active generators you wish to use and the level of output signal fidelity that you desire, always specify the largest DOC buffer size and the lowest output sample rate that you possibly can. Remember that a large number of active generators implies a very small maximum buffer size for any particular oscillator, so you should always try to minimize the number of generators that are active at any one time. As a rough benchmark, the clicks of signal interruption begin to blend into highly audible static when you specify buffers smaller than 8K for use at the maximum-fidelity output sample rate of about 26 kHz. (Note: The DOC supports greater sample rates, but these rates are limited by the output filtering on the IIGS which permits no greater signal fidelity than that possible using the 26 kHz rate.) Our figures suggest that output fidelity must suffer, or signal noise must increase, when more than four generators (eight oscillators in swap mode) are operating simultaneously.

Avoid Silent or Quiet Passages

The signal content of your waveform can hide the additional noise caused by the "swap-mode anomaly." The more complex and louder a waveform, the less your ear will perceive the brief interruption that occurs whenever a higher-numbered oscillator swaps to a lower-numbered one; pop and rock music is far less susceptible to this problem than classical, folk, or jazz pieces, which typically include many quiet passages. In addition, a signal that naturally contains a large amount of "pink noise," such as recordings of rainstorms or the surf at the beach, can mask the anomalous noise altogether.

Arrange for Swaps to Occur at or Near Zero Crossings

If the high-to-low swap occurs at a time when the normal output signal level sits at or near the zero crossing, the swap will cause little or no audible damage to the waveform. When reproducing arbitrary sampled sound, it is almost impossible to insure that the output signal level is near the zero crossing. However, when constructing long waveforms for playback, you may be able to sidestep the chip's anomalous behavior by ensuring that the waveform values lie at or near $80 at the end of every waveform segment, where a waveform segment spans twice the length of one oscillator buffer. For example, if you specify a buffer size of 4K, make sure that your constructed waveform crosses the baseline after every 8,192 samples, and for 16K buffers, make sure that the waveform makes a zero crossing after every 32K.

The length of the waveform segment should be twice the buffer length only if you are going to reproduce the waveform exactly once per `FFStartSound` call. It may be necessary to shorten the length of the waveform segment to exactly the specified DOC buffer length if you use the `nextwave_start` parameter in the `FFStartSound` parameter block to invoke automatic looping of the waveform. In other words, you may need to arrange for twice as many zero crossings in your constructed waveform in the looping case as you would under normal circumstances since subsequent repetitions of the waveform during the single `FFStartSound` call may begin with either the even or odd oscillator, depending upon which member of the pair was active when the previous repetition ended. If the playback of a waveform starts with the odd oscillator, then the odd-to-even swaps will occur at different points in the waveform than they would when the playback starts with the even oscillator.

Also note that the use of larger buffers causes a progressively longer disabling of interrupts while the Sound Tool Set moves the waveform into the DOC RAM.

Further Reference

- *Apple IIGS Toolbox Reference*, Volume 2
- *Apple IIGS Hardware Reference*

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#12: Tool Set Interdependencies

Revised by: Matt Deatherage & Dave Lyons May 1992
Written by: Jim Merritt April 1987

This Technical Note lists all known interdependencies between system tool sets on the Apple IIGS.
Changes since January 1990: Added new and changed dependencies for System Software 6.0.

A tool set is dependent upon another if you must start the latter before starting the former. You should start tool sets in the order listed below. Names marked with an asterisk (*) indicate a recommendation to start the corresponding tool set, but the order is not required for operation of the dependent tool. Apple recommends using `StartUpTools` to start up all the tool sets your application needs. See the *Apple IIGS Toolbox Reference*, Volume 3 for more details.

Tool Set Interdependencies

Tool Locator **Tool #1 ($01)**
No dependencies. Always start this tool set before any others.

Memory Manager **Tool #2 ($02)**
Tool Locator (#1)

Miscellaneous Tools **Tool #3 ($03)**
Tool Locator (#1)
Memory Manager (#2)

QuickDraw II **Tool #4 ($04)**
Tool Locator (#1)
Memory Manager (#2)
Miscellaneous Tools (#3)

Desk Manager **Tool #5 ($05)**
Tool Locator (#1)
Memory Manager (#2)
Miscellaneous Tools (#3)
QuickDraw II (#4)
Event Manager (#6)
Window Manager (#14)
Control Manager (#16)
Menu Manager (#15)
Line Edit (#20)
Dialog Manager (#21)
Scrap Manager (#22)

Event Manager **Tool #6 ($06)**

Tool Locator	(#1)	
Memory Manager	(#2)	
Miscellaneous Tools	(#3)	
Scheduler		**Tool #7 ($07)**
Tool Locator	(#1)	
Memory Manager	(#2)	
Miscellaneous Tools	(#3)	
Sound Tools Set		**Tool #8 ($08)**
Tool Locator	(#1)	
Memory Manager	(#2)	
Miscellaneous Tools	(#3)	
Apple Desktop Bus (ADB)		**Tool #9 ($09)**
Tool Locator	(#1)	
SANE (Standard Apple Numeric Environment)		**Tool #10 ($0A)**
Tool Locator	(#1)	
Memory Manager	(#2)	
Integer Math Tools		**Tool #11 ($0B)**
Tool Locator	(#1)	
Text Tools		**Tool #12 ($0C)**
Tool Locator	(#1)	
Window Manager		**Tool #14 ($0E)**
Tool Locator	(#1)	
Memory Manager	(#2)	
Miscellaneous Tools	(#3)	
QuickDraw II	(#4)	
Event Manager	(#6)	
* QuickDraw Auxiliary	(#18)	Required in 6.0 and later, and the Window Manager loads and starts it for you.
Control Manager	(#16)	
Menu Manager	(#15)	
* Line Edit	(#20)	For `AlertWindow` call only
* Font Manager	(#27)	For `AlertWindow` call only
* ResourceManager	(#30)	For using resources in Window Manager calls.

Menu Manager **Tool #15 ($0F)**

Tool Locator	(#1)	
Memory Manager	(#2)	
Miscellaneous Tools	(#3)	
QuickDraw II	(#4)	
Event Manager	(#6)	
Window Manager	(#14)	
Control Manager	(#16)	
* ResourceManager	(#30)	For using resources in Menu Manager calls.

Control Manager **Tool #16 ($10)**

Tool Locator	(#1)	
Memory Manager	(#2)	
Miscellaneous Tools	(#3)	
QuickDraw II	(#4)	
Event Manager	(#6)	
Window Manager	(#14)	
Menu Manager	(#15)	
* QuickDraw Auxiliary	(#18)	For `statText` controls.
* Line Edit	(#20)	For `editLine` controls.
* Font Manager	(#27)	For `statText` controls.
* List Manager	(#28)	For `list` controls.
* ResourceManager	(#30)	For using resources in Control Manager calls.
* Text Edit	(#34)	For `editText` controls.

Note: You should consider the Window, Control, and Menu Managers as one unit and start them in the given order.

System Loader **Tool #17 ($11)**

Tool Locator	(#1)
Memory Manager	(#2)
Miscellaneous Tools	(#3)

QuickDraw Auxiliary Routines **Tool #18 ($12)**

Tool Locator	(#1)
Memory Manager	(#2)
Miscellaneous Tools	(#3)
QuickDraw II	(#4)
* Font Manager	(#27)

Note: QuickDraw Auxiliary uses the Font Manager in the picture drawing routines. For proper operation, you should start the Font Manager before using the QuickDraw Auxiliary picture routines; however, the picture routines do not fail if the Font Manager is not present.

Print Manager **Tool #19 ($13)**

Tool Locator	(#1)
Memory Manager	(#2)
Miscellaneous Tools	(#3)
QuickDraw II	(#4)
QuickDraw Auxiliary	(#18)
Event Manager	(#6)
Window Manager	(#14)
Control Manager	(#16)
Menu Manager	(#15)
Line Edit	(#20)
Dialog Manager	(#21)
List Manager	(#28)
Font Manager	(#27)

Line Edit **Tool #20 ($14)**

Tool Locator	(#1)	
Memory Manager	(#2)	
Miscellaneous Tools	(#3)	
QuickDraw II	(#4)	
Event Manager	(#6)	
* QuickDraw Auxiliary	(#18)	For `Text2` items; see below.
Scrap Manager	(#22)	
* Font Manager	(#27)	For `Text2` items; see below.

Dialog Manager **Tool #21 ($15)**

Tool Locator	(#1)	
Memory Manager	(#2)	
Miscellaneous Tools	(#3)	
QuickDraw II	(#4)	
Event Manager	(#6)	
Window Manager	(#14)	
Control Manager	(#16)	
Menu Manager	(#15)	
* QuickDraw Auxiliary	(#18)	For `Text2` items; see below.
Line Edit	(#20)	
* Font Manager	(#27)	For `Text2` items; see below.

Note: Line Edit, the Dialog Manager, and the Control Manager require the presence of the Font Manager and QuickDraw Auxiliary if you use `LETextBox2`, `statText` controls, or `LongStatText2` items which require any font styling (e.g., outline, boldface, etc.).

Scrap Manager **Tool #22 ($16)**

Tool Locator	(#1)
Memory Manager	(#2)

Standard File Operations **Tool #23 ($17)**

Tool Locator	(#1)	
Memory Manager	(#2)	
Miscellaneous Tools	(#3)	
QuickDraw II	(#4)	
Event Manager	(#6)	
Window Manager	(#14)	
Control Manager	(#16)	
Menu Manager	(#15)	
* QuickDraw Auxiliary	(#18)	Required in 6.0 and later, and the Window Manager loads and starts it for you.
Line Edit	(#20)	
Dialog Manager	(#21)	
* List Manager	(#28)	
* Resource Manager	(#30)	For using resources in Standard File Operations calls.

Note: Standard File 3.0 and later use the List Manager for displaying a list of file names. Although Standard File functions properly if the application has not started the List Manager, it saves time if the application does so.

Note Synthesizer **Tool #25 ($19)**

Tool Locator	(#1)
Memory Manager	(#2)
Sound Tools	(#8)

Note Sequencer **Tool #26 ($1A)**

Tool Locator	(#1)
Memory Manager	(#2)
Sound Tools	(#8)
Note Synthesizer	(#25)

Note: The Note Sequencer automatically handles the start and shutdown of the Free-Form Sound Tools (#8) and the Note Synthesizer (#25), so programs that use the Note Sequencer must **not** execute start or shutdown calls for those tools. Automatic start does not imply automatic **loading**. If you plan to use the Note Sequencer, you must still load the Free-Form Sound Tool and the Synthesizer Tool explicitly through calls to the Tool Locator routines `LoadTools` or `LoadOneTool` or by calling the System Loader and Tool Locator directly in appropriate cases.

Font Manager **Tool #27 ($1B)**

Tool Locator	(#1)	
Memory Manager	(#2)	
* Miscellaneous Tools	(#3)	For `ChooseFont` call only.
QuickDraw II	(#4)	
* Integer Math Tools	(#11)	For `ChooseFont` call only.
* Window Manager	(#14)	For `ChooseFont` call only.
* Control Manager	(#16)	For `ChooseFont` call only.
* Menu Manager	(#15)	For `FixFontMenu` call only.
* List Manager	(#28)	For `FixFontMenu` and `ChooseFont` calls.
* Line Edit	(#20)	For `ChooseFont` call only.
* Dialog Manager	(#21)	For `ChooseFont` call only.

List Manager **Tool #28 ($1C)**

Tool Locator	(#1)
Memory Manager	(#2)
Miscellaneous Tools	(#3)
QuickDraw II	(#4)
Event Manager	(#6)
Window Manager	(#14)
Control Manager	(#16)
Menu Manager	(#15)

Audio Compression and Expansion (ACE) **Tool #29 ($1D)**

Tool Locator	(#1)
Memory Manager	(#2)

Resource Manager **Tool #30 ($1E)**

Tool Locator	(#1)
Memory Manager	(#2)

MIDI Tools **Tool #32 ($20)**

Tool Locator	(#1)
Memory Manager	(#2)
Miscellaneous Tools	(#3)
Sound Manager	(#8)
* Note Synthesizer	(#25)

Note: The MIDI Tools require the Note Synthesizer if you intend to use the MIDI clock feature. If you are not using the MIDI clock, the Note Synthesizer is not required.

Text Edit **Tool #34 ($22)**

Tool Locator	(#1)	
Memory Manager	(#2)	
Miscellaneous Tools	(#3)	
QuickDraw II	(#4)	
Event Manager	(#6)	
Window Manager	(#14)	
Menu Manager	(#15)	
Control Manager	(#16)	
QuickDraw Auxiliary	(#18)	
Scrap Manager	(#22)	
Font Manager	(#27)	
* Resource Manager	(#30)	For using resources in Text Edit calls.

MIDI Synth **Tool #35 ($23)**

Tool Locator	(#1)
Memory Manager	(#2)
Miscellaneous Tools	(#3)
Sound Tools	(#8)

Media Control Tool **Tool #38 ($26)**

Tool Locator	(#1)
Memory Manager	(#2)
Miscellaneous Tools	(#3)
Integer Math	(#11)
Resource Manager	(#30)

Recommended Start Order

A close look at the preceding information will reveal apparent "circular dependencies" between various tool sets (i.e., two or more tool sets may depend upon each other). To resolve the issue of which tool set to start first in such a situation, here is a list of the most commonly used tool sets, given in the order in which an application should start them. You may start those tools which are indented at a specific level at that time or any time thereafter.

Tool Locator	(#1)	
ADB Tools	(#9)	
Integer Math Tools	(#11)	
Text Tools	(#12)	
Memory Manager	(#2)	
SANE	(#10)	
ACE	(#29)	
Resource Manager	(#30)	
Miscellaneous Tools	(#3)	
Scheduler	(#7)	
System Loader	(#17)	`LoaderStartup` does nothing.
Media Control	(#38)	
QuickDraw II	(#4)	
QuickDraw II Auxiliary	(#18)	
Event Manager	(#6)	
Window Manager	(#14)	
Control Manager	(#16)	
Menu Manager	(#15)	
Line Edit	(#20)	
Dialog Manager	(#21)	
either		
Sound Tools then	(#8)	
Note Synthesizer	(#25)	
or		
Note Sequencer	(#26)	
MIDI Tools	(#32)	
MIDI Synth	(#35)	
Standard File Operations	(#23)	
Scrap Manager	(#22)	
List Manager	(#28)	
Font Manager	(#27)	
Print Manager	(#19)	
Text Edit	(#34)	
Desk Manager	(#5)	

Note: Although you may start the sound-related tools any time after the Miscellaneous Tools, we recommend you start them after most of the Desktop-related tools. We also recommend you start the Desk Manager last and shut it down first.

Further Reference

- *Apple IIGS Toolbox Reference*

Apple IIGS
#13: ROM 1.0 Modem Firmware Bug

Revised by: Matt Deatherage November 1988
Written by: Mike Askins April 1986

This Technical Note formerly discussed a bug involving buffering and serial port setting commands in the modem firmware in ROM 1.0.

Apple IIGS ROM 2.0 fixes a bug involving buffering and serial port setting commands in the modem firmware. You should not have to write a special case to handle this bug since it is reasonable to expect users to have the updated ROM which is offered as a free upgrade from Apple.

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#14: Standard File Screwiness

Revised by: Dave Lyons May 1992
Written by: Guillermo Ortiz, Matt Deatherage, & Dave Lyons June 1987

This Technical Note describes known anomalies in Standard File.
Changes since December 1991: Updated for System 6.0. Problems with the infinite loop and SFMultiGet2 reply record are fixed.

Prefix Check is Case Sensitive

When you advance to the next volume using Command-Tab (or just Tab, before 6.0), Standard File checks your prefix against the name of the volume now in the same device you were just using, to see if you switched disks (this is possible on a 5.25 drive, for example). If the name doesn't match, you stay at the same device.

Unfortunately, the comparison in 6.0 and earlier is case sensitive. If you have a volume called "MyDisk" and prefix zero is set to ":MYDISK", advancing to the next volume doesn't get you anywhere the first time (but the prefix changes from ":MYDISK" to ":MyDisk").

The following two problems are fixed in System 6.0:

Infinite Loop with Empty Prefixes

In System Software versions 5.0 through 5.0.4, all Standard File dialogs can hang if both prefixes 0 and 8 are empty (GS/OS uses prefix 8 to expand partial pathnames if prefix 0 is empty).

If this affects your software, use `GetPrefix` to check for empty prefixes before calling Standard File. If 0 and 8 are both empty, set prefix 0 to "`*:`" (or any other convenient pathname).

SFMultiGet2 (and SFPMultiGet2) reply record

`SFMultiGet2` and `SFPMultiGet2` in System 5.0.4 and earlier accidentally validate the multi-file reply record as if it were a regular new-style reply record (as for `SFGetFile2`, for example). The validation is a check that the words at offsets $08 and $0E in the record are not $0002 (these are `nameRefDesc` and `pathRefDesc` in a new-style reply record).

To ensure that Standard File does not erroneously reject your multi-file reply record (and return error $1704), you may include ten bytes of $00 following the six-byte record.

Further Reference

- *Apple IIGS Toolbox Reference*, Volumes 2 & 3

Apple IIGS
#15: InstallFont and Big Fonts

Revised by: Eric Soldan & Matt Deatherage July 1989
Written by: Guillermo Ortiz June 1987

When the Font Manager executes `InstallFont`, it may try to scale the selected font if bit 15 of the `ScaleWord` is clear; a font larger than 32K causes this call to fail.
Changes since November 1988: Noted System Software 5.0 enhancements.

Before System Software 5.0, the Font Manager could not scale a font larger than 32K, so `InstallFont` would fail if scaling was required and the desired font exceeded this limit. If the call failed for this reason, it reported an `FMScaleSizeErr` ($1B0C) error.

This is not the same situation as when there is not enough memory available to hold a newly scaled font. The situation will generate Memory Manager errors.

System Software 5.0 can scale fonts to be larger than 32K, so there is no longer the limit imposed by System Software 4.0 and earlier. In addition, System Software 5.0 can handle font sizes up to 255 points, if memory is available. Note that this is a different situation than trying to scale a font which was originally larger than 32K, but both work under 5.0 and later.

Apple IIGS
#16: Notes on Background Printing

Revised by: Mike Askins November 1988
Written by: Mike Askins June 1987

This Technical Note attempts to pinpoint some of the common problems people encounter when using background printing as available through the serial firmware.

Calling Sequence

`Init call`	Starts the serial firmware
`SetOutBuff`	Specifies a buffer to place data to be printed
	Places data in buffer (amount < buffer size)
`SendQueue`	Starts the background printing process

Correctly Making the SendQueue Call

The *Apple IIGS Firmware Reference* incorrectly documents the parameters you pass to `SendQueue`. The correct specification of the recharge address does not correspond to the standard method of passing a full 32-bit address. Set the parameters as follows:

SendQueue
Launches background printing.

```
CmdList     DFB $04                    ;Parameter Count
            DFB$18                     ;Command Code
            DW $00                     ;Result Code (output)
            DW  DataLength
            DFB RechargeAddress (bank)
            DFB RechargeAddress (high)
            DFB RechargeAddress (low)
            DFB $00
```

Using the Default Buffer

You can use the area which the firmware reserves for transparent buffering to place data for background printing. This is advantageous since the firmware calls the Memory Manager to

allocate space for the buffer (you must allocate the space from the Memory Manager if you use the `SetOutBuff` call to set up a buffer).

To use the serial firmware's buffer, you must first enable buffering by initializing the port with `PINIT` and sending it the string "^IBE" with `PWRITE`. Once you enable buffering, call `GetOutBuff` to find the size and location of the buffer, then place your data (buffersize - 1) in the buffer and call `SendQueue`.

Data Size

Make sure that the amount of data you place in the buffer is at least one byte less than the size of the buffer since the firmware uses one byte of the buffer for bookkeeping purposes; if you place too much data in the buffer, it will continually print the buffer's contents and never call your recharge routine.

The Recharge Routine

You should treat the recharge routine as an interrupt handler and execute it at interrupt time. Interrupts are disabled at this time, and it is illegal to enable them within the recharge routine. Like all interrupt handlers, the recharge routine should take care of its business as quickly as possible then exit; any excessive delays cause interrupt dependent processes (e.g., AppleTalk) to fail. You should also remember that most of the system code is non-reentrant; you should use the Scheduler when calling system code which may have been running when the serial interrupt that invoked the recharge routine occurred.

The serial firmware is not generally reentrant and does not interact with the Scheduler. If you want to make serial firmware calls (through $C1xx, $C2xx) from your recharge routine, you must preserve `MSLOT` (the byte at $0007F8) across those calls. Be aware that any non-recharge code must not make calls to the serial firmware that will disrupt the background printing process; sending the string "^BD" (disable buffering command), for example, is guaranteed to confuse a running background printing process.

Further Reference

- *Apple IIGS Firmware Reference*

Apple IIGS
#17: Application Memory Management and the MMStartUp User ID

Revised by: Steven Glass & Rich Williams November 1988
Written by: Jim Merritt June 1987

This Technical Note describes a technique which permits an application to dispose of any memory it has used with a single Memory Manager call without clobbering other system components or itself.

Ground Rules for Application Memory Usage

Apple IIGS programs must be responsible for allocating and disposing of any memory they use, over and above that which the operating system itself gives them. In general, no IIGS program should use any memory except that which the Memory Manager has explicitly granted to it. A program may request additional memory for its own use at any time with one or more calls to the `NewHandle` routine. At program termination, the application is responsible for explicitly disposing of any memory that it explicitly acquired, and if it fails to do so, it could leave the IIGS memory management system in a corrupted state.

You may dispose of memory on a handle-by-handle basis, or you may dispose of it *en masse* by calling `DisposeAll`, but you should **never** use `DisposeAll` with the user ID that the `MMStartUp` routine provides. This user ID is the "master user ID" for the application, and it tags the memory space which the operating system reserves for the program's code and static data at load time. Calling `DisposeAll` with this user ID results in immediate deallocation of the memory in which the calling program resides; therefore, an application which allocates dynamic data space using only the user ID that `MMStartUp` gives it should not use `DisposeAll` to deallocate that space, but rather use `DisposeHandle` to deallocate it handle by handle.

Cleaning Up With DisposeAll

It is possible, however, for a program to use a different, unique user ID when allocating its own RAM, then pass that user ID to `DisposeAll` when it terminates to deallocate all of its private memory at once without endangering itself or other parts of the IIGS system. With this technique, the question is how best to acquire a new user ID? One method to acquire a new user

ID is to request a completely new one of the appropriate type from the User ID Manager in the Miscellaneous Tools. In this case, when the application terminates, it must not only deallocate the memory it used, but also the additional user ID which it requested from the User ID Manager.

Actually, it is not necessary for a program to acquire a completely new user ID to use `DisposeAll` without clobbering itself. Instead, the application may modify the `auxID` field of the master user ID which `MMStartUp` assigns to create a unique user ID for allocating its own memory. The 16-bit user ID contains the `auxID` field in bits $8 – $B. The value of this field, which may range from $0 to $F, is always zero in the application's master user ID, but you can fill it with any non-zero value to create up to 15 new and distinct user IDs, each of which you can pass to `NewHandle` to allocate memory.and to `DisposeAll` to deallocate memory without endangering the memory tagged by the master user ID. The following assembly code fragment illustrates this technique:

```
        ; assumes full native mode
                pushword #0                 ; room for user ID
                _MMStartUp
                pla                         ; master user ID
                sta MasterID
                ora  #$0100                 ; auxID:= 1

        ;       (COULD HAVE BEEN ANYTHING FROM $1 to $F)

                sta MyID                    ; use this to allocate private memory
                ...
                etc.
                ...

        ; ready to exit program
                pushword MyID
                _DisposeAll                 ; dumps only my own RAM

; now do any remaining processing related to termination
```

You do not need to explicitly deallocate any user ID that you derive by changing the `auxID` field of a valid master user ID. When the system (usually the one to deallocate the master) deallocates the master user ID, it also deallocates its derivatives.

One Word of Caution

Several of the Memory Manager's "All" calls (e.g., `DisposeAll`) treat a zeroed `auxID` field as a wildcard which matches any value that the field may contain, thus if you call `DisposeAll` with the application's master user ID (where the `auxID` field is zero), the Memory Manager will not only deallocate all memory belonging to the master user ID, but also all handles and memory segments that are associated with user IDs which are derived from that master. The Loader's `UserShutDown` mechanism typically executes such a call when cleaning up after a normal (i.e., non-restartable) application to keep the memory management system from clogging. This action is purely a defensive measure, and well-behaved applications – particularly restartable ones – should dispose of their own memory and never rely upon the operating system to clean up after them.

Further Reference

- *Apple IIGS Toolbox Reference*, Volume 1

Apple IIGS
#18: Do-It-Yourself SCC Access

Revised by: Jim Luther July 1990
Written by: Jim Luther, Mike Askins, Matt Deatherage & Jim Mensch June 1987

This Technical Note describes how to install and remove a interrupt handler routine for the Z8530 Serial Communications Controller (SCC) on the Apple IIGS without breaking other parts of the system. This Note includes many suggestions that, if unheeded, could come back to haunt you in the form of bug fixes to your program.

Changes since March 1990: Added a method for finding which serial port AppleTalk is using under GS/OS.

Free Serial Routines Inside

The Z8530 SCC has 2 serial channels, supports several synchronous and asynchronous data communications protocols, and has 9 read registers and 16 write registers per channel. (Compare this to the 5 registers of the 6551 Asynchronous Communications Interface Adapter.) To program the SCC correctly, you must understand five things: the SCC, the Apple IIGS hardware environment in which the SCC lives, the Apple IIGS interrupt handler firmware, the interrupt support provided by the operating system, and the data communication protocol you want to use. If you don't understand all of these components, stick to the serial firmware.

The Apple IIGS serial firmware is a robust environment for almost every asynchronous serial programming application. If you want to handle all SCC operations and SCC interrupts on the IIGS without using the serial firmware, then you must really **know** the firmware won't do the job for you or you wouldn't be going to a lot of trouble to recreate the services the firmware routines already provide.

Don't Eat Your Serial with Your Mouth Open

Your mother has rules and so does Apple. On many systems, your application may be sharing the SCC chip with System Software such as AppleTalk or the serial firmware. If you want to access the SCC chip directly without breaking the system (or the system breaking you), then follow these simple rules.

Rule #1: Before using a serial port, make sure AppleTalk is not already using it.

If AppleTalk is active, it uses one of the serial ports. The user selects which serial port AppleTalk uses with the Control Panel. Before using one of the serial ports, you should always check to make sure AppleTalk is not using that port. If AppleTalk is using the serial port your application wants to use, tough luck; tell the user about it, but don't even think about using that port.

Under ProDOS 8, use the method shown in the following sample code to determine if AppleTalk is using a serial port:

```
;
; This routine checks to see which serial port, if any, AppleTalk is using.
; The routine sets a flag byte, ApTalkPort, and the accumulator to indicate
; which port (if any) AppleTalk is using.
;    $00 = AppleTalk is not using a serial port
;    $01 = AppleTalk is using serial port 1 (printer port)
;    $02 = AppleTalk is using serial port 2 (modem port)
; Note:  This method should be used under ProDOS 8 only.  Under GS/OS, use the
;        .AppleTalk driver's GetPort DStatus subcall.
;
; Enter routine in emulation mode
;
                    longa off
                    longi off
                    mcopy 2/AInclude/M16.MiscTool

WhichPort           start

IDROUTINE           equ $FE1F           returns system ID information

                    stz ApTalkPort      default to not AppleTalk

                    jsr IDROUTINE       call to the system ID routine
                    cpy #$03
                    bcs NewIIGS

OldIIGS             anop                this is a pre-ROM 03 IIGS
                    clc                 to native mode
                    xce
                    rep #$30            16 bit m and x
                    longa on
                    longi on

                    pea $0000           space for result
                    pea $0021           Slot 1 setting
                    _ReadBParam         read battery RAM parameter
;                                         (2 byte result left on stack)

                    pea $0000           space for result
                    pea $0027           Slot 7 setting
                    _ReadBParam         read battery RAM parameter
                    pla                 get slot 7 setting (2 bytes)

                    sec                 emulation mode
                    xce
                    longa off
                    longi off

                    beq FindYourCard    AppleTalk is active
                    pla                 remove slot 1 setting LSB (1 byte)
                    bra OldExit

FindYourCard        inc ApTalkPort      default to port 1
```

```
                    pla                 is slot 1 "your card"? (1 byte)
                    beq ItsPort2        no, must be port 2
                    bra OldExit

ItsPort2            inc ApTalkPort      port 2 is AppleTalk

OldExit             pla                 remove slot 1 setting MSB (1 byte)
                    lda ApTalkPort
                    rts                 return to caller

NewIIGS             anop                ROM 03 or greater IIGS
                    clc                 to native mode
                    xce
                    rep #$30            16 bit m and x
                    longa on
                    longi on

                    pea $0000           space for result
                    pea $000C           port 2 type
                    _ReadBParam         read battery RAM parameter
;                                         (2 byte result left on stack)

                    pea $0000           space for result
                    pea $0000           port 1 type
                    _ReadBParam         read battery RAM parameter
                    pla                 get port 1 setting (2 bytes)

                    sec                 emulation mode
                    xce
                    longa off
                    longi off

                    cmp #$02            is port 1 AppleTalk?
                    bne TryPort2        no
                    inc ApTalkPort      yes
                    pla                 then remove port 2 setting LSB (1 byte)
                    bra NewExit           and exit

TryPort2            pla                 get port 2 setting LSB (1 byte)
                    cmp #$02            is port 2 AppleTalk?
                    bne NewExit         no
                    lda #$02            yes
                    sta ApTalkPort

NewExit             pla                 remove port 2 setting MSB (1 byte)
                    lda ApTalkPort
                    rts                 return to caller

ApTalkPort          entry
                    ds  1               will be 0, 1, or 2
                    end
```

Under GS/OS, use the method shown in the following sample code to determine if AppleTalk is using a serial port:

```
;
; This routine checks to see which serial port, if any, AppleTalk is using.
; The routine sets a flag byte, ApTalkPort, and the accumulator to indicate
; which port (if any) AppleTalk is using.
;    $0000 = AppleTalk is not using a serial port
;    $0001 = AppleTalk is using serial port 1 (printer port)
;    $0002 = AppleTalk is using serial port 2 (modem port)
; Note:  This method should be used under GS/OS only.
;
```

```
; Enter routine in native 16 bit mode
;
                    longa on
                    longi on
                    mcopy 2/AInclude/M16.GSOS
```

```
CheckPort           Start

GetPort             equ $8001           The .AppleTalk DStatus subcall to get
;                                       the port number AppleTalk is currently
;                                       using.

                    phb                 save data bank
                    phk                 data bank = code bank
                    plb

                    lda #$0001          start with device #1
                    sta DIdevNum

FindATDriver        anop
                    _DInfoGS DInfoParms ;call Dinfo
                    bcs DIError         stop searching if error
                    lda DIdeviceIDNum
                    cmp #$001D          is it the AppleTalk main driver?
                    beq ATDriverFound   yes
                    inc DIdevNum        check the
                    bra FindATDriver    next device number

ATDriverFound       anop
                    lda DIdevNum        store device number
                    sta DSdevNum        in the DStatus parm list
                    _DStatusGS DStatusParms ;call DStatus
                    lda portNum         get the port number
                    sta ApTalkPort
                    bra Exit

DIError             anop
;                   cmp #$0011          invalid device number, so the
;                   beq NotFound        AppleTalk main driver wasn't found
;
; Add your code to handle any other errors from DInfo here, because the
; end of the device list was not found.

NotFound            stz ApTalkPort      neither port is in use
                    bra Exit

Exit                anop
                    lda ApTalkPort
                    plb                 restore data bank
                    rtl                 return to caller

ApTalkPort          entry
                    ds 2                will be 0, 1, or 2

DInfoParms          anop
                    dc i2'8'            pCount = 8 parameters
DIdevNum            dc i2'1'            devNum
                    dc a4'NameBuffer'   devName
                    ds 2                characteristics
                    ds 4                totalBlocks
                    ds 2                slotNum
                    ds 2                unitNum
                    ds 2                version
DIdeviceIDNum       ds 2                deviceIDNum

NameBuffer          anop
                    dc i2'31'           Class 1 input string. Max Length=31
                    ds 33
```

```
DStatusParms        anop
                    dc i2'5'            pCount = 5 parameters
DSdevNum            ds 2                devNum
                    dc i2'GetPort'      statusCode = GetPort
                    dc a4'GetPortSList' statusList = GetPortSList
                    dc i4'2'            requestCount = 2
                    ds 4                transferCount

GetPortSList        anop                the GetPort subcall's statusList
portNum             ds 2                $0001 = AppleTalk is using port 1 (printer
port)
;                                       $0002 = AppleTalk is using port 2 (modem port)
                    dc i2'0'

                    end
```

Rule #2: Don't use the SCC Interrupt Handler Vector.

Contrary to what you may have read in a previous version of this Note, you cannot reliably attach your SCC interrupt handler to the SCC Interrupt Handler Vector (vector reference number $0009). The Apple IIGS serial firmware owns the SCC Interrupt Handler Vector (or at least **it** thinks it does). Anytime the serial firmware is used, there is a chance that the serial firmware can grab the SCC Interrupt Handler Vector for its use. CDAs and NDAs that print, the Print Manager tool set, the Text tool set, and the generated GS/OS character drivers associated with the serial ports are examples of code that can and do use the serial firmware.

The only safe place to connect into the interrupt chain is through the operating system. The ProDOS 8 and GS/OS ProDOS 16 call, `ALLOC_INTERRUPT` is the correct place to attach your interrupt handler. The GS/OS `BindInt` call cannot be used to attach your interrupt handler to the SCC Interrupt Handler Vector (VRN $0009) for the same reason that you cannot use the SCC Interrupt Handler Vector directly.

Rule #3: Be very, very careful with SCC Write Register 9 (WR9).

The Z8530 SCC has four registers which are shared by both channels (ports). Of those four, only two are commonly used in the Apple IIGS, `RR3` and `WR9`. `RR3`, which only exists in channel A, lets you check the interrupt pending bits for both SCC channels. `WR9` is the Master Interrupt Control register for both SCC channels and contains the Reset command bits.

You must never reset the channel AppleTalk is using (resetting the channel AppleTalk is using kills AppleTalk). This means you should **never** perform a Force Hardware Reset command (11xxxxxx to `WR9`) even though the *Z8530 Serial Communications Controller Technical Manual* tells you to in the SCC initialization procedure. A hardware reset is performed at system startup, so you shouldn't need to perform a channel reset, even to the channel you are using.

The interrupt control bits (bits D5 - D0) in `WR9` should not be modified (an exception is when you are installing your own SCC interrupt handler). AppleTalk expects the interrupt control bits to always be 001010. If you find the need to perform a channel reset on your channel, remember that the interrupt control bits are programmed at the same time as a channel reset.

Hints for the Serial Adventure

Next are a few hints for those who would like to explore the world of knocking on the registers of the Z8530 SCC.

Hint #1: Synchronize your code with the SCC logic.

Before writing to the SCC chip for the first time, you should make an attempt to ensure your code is synchronized with the SCC's logic. This needs to be done only once when you are initializing the SCC. This can be accomplished with a single read of SCC Read Register 0 (`RR0`). For example, if you're using serial port 2 (the modem port), the following code reads `RR0` of SCC channel B:

```
                    longa off           must be using 8-bit accumulator
                    lda $C038           read RR0 of SCC Channel B
```

Hint #2: Watch out for interrupts from the other SCC channel.

Except for `RR0`, `WR0`, and the two SCC data registers, all SCC registers are accessed in a two-step process. First, the register number you want to select is written to `WR0`. After the register number is set, the next read from or write to the command register accesses the register selected in the first step. Because several of the SCC registers are shared between the two SCC channels and because code accessing them may not always be yours (i.e., AppleTalk), interrupts should be disabled during the two steps. The following code shows two quick subroutines to access the SCC's Read and Write registers while preventing interrupts between the register number set and the register read or write steps:

```
                    longa off           must be using 8-bit accumulator
                    longi off             and index registers
;
; Write to a SCC command register - channel A or B.
; Input:  A = value to store
;         X = SCC register number ($0-$F)
;         Y = $01  channel A
;             $00  channel B
;
WriteSCC            php                 save the current interrupt status
                    sei                 disable interrupts
                    pha                 save value to write
                    txa                 get SCC register number from X
                    sta $C038,y         set the register number
                    pla                 restore value to write
                    sta $C038,y         write the value
                    plp                 restore the interrupt status
                    rts
```

```
;
; Read from a SCC command register - channel A or B.
; Input:  A = SCC register number ($0-$F)
;         Y = $01  channel A
;             $00  channel B
; Output: A = register value
;
ReadSCC             php                 save the current interrupt status
                    sei                 disable interrupts
                    sta $C038,y         set the SCC register number
                    lda $C038,y         get the value from the SCC register
                    xba                 look ahead 2 lines...
                    plp                 restore the interrupt status
                    xba                 set N and Z flags for exit
                    rts
```

Just to be complete, here's how `RR0`, `WR0`, the receive buffer, and the transmit buffer SCC registers are accessed on the Apple IIGS:

```
                    longa off           must be using 8-bit accumulator
                    longi off             and index registers
;
; Read RR0 - channel A or B
; Input:  Y = $01  channel A
;             $00  channel B
; Output: A = RR0 register value
;
ReadRR0             lda $C038,y         get the value from RR0
                    rts
;
; Write WR0 - channel A or B
; Input:  A = value to store at WR0
;         Y = $01  channel A
;             $00  channel B
;
WriteWR0            sta $C038,y         write the value to WR0
                    rts
;
; Read from SCC receive buffer - channel A or B
; Input:  Y = $01  channel A
;             $00  channel B
; Output: A = value of data received
;
ReadData            lda $C03A,y         get the value from SCC data register
                    rts
;
; Write to SCC transmit buffer - channel A or B
; Input:  A = value of data to transmit
;         Y = $01  channel A
;             $00  channel B
;
WriteData           sta $C03A,y         write the value to SCC data register
                    rts
```

Hint #3: All SCC channels are not created equal.

In the IIGS, the SCC's receive and transmit clocks for both channels are driven by a single crystal oscillator circuit. This is accomplished by connecting a 3.6864 MHz crystal between the /RTxC and /SYNC pins of channel A. Channel B's /RTxC pin is connected to Channel A's /SYNC pin to drive channel B's clocks from channel A's oscillator circuit.

Because of this single circuit, Write Register 11 (WR11) bit 7 must be set to 1 for SCC channel A and must be set to 0 for SCC channel B.

Hint #4: RR3 is available only in SCC channel A.

When your interrupt handler is checking to see if the interrupt condition was caused by your SCC channel, remember to always look at RR3 in SCC channel A. RR3 in channel A contains the interrupt pending bits for both SCC channels. RR3 in channel B always returns all zeros, which doesn't tell you a lot about what's happening.

Don't be a Serial Killer

How to Install and Remove your SCC Interrupt Handler

If you're going to handle serial I/O and don't want your application to have to poll the SCC chip all the time to see if something has happened, you probably want to install an interrupt handling routine that is called every time a SCC chip condition you want to know about occurs. This section of the Note shows how to install and remove your own SCC interrupt handler.

The steps for installing your SCC interrupt handler are:

1. Ensure the serial firmware's Input and Output buffering is disabled. The state of I/O buffering can be checked by looking at bit 14 of the ModeBitImage parameter returned by the GetModeBits extended interface call. I/O buffering can be disabled with the firmware's BD control command.
2. Disable the SCC Master Interrupt Enable (WR9, bit 3) briefly while performing the next six steps. The value you should write to WR9 is 00000010.
3. Get the address of the system interrupt flag byte, SerFlag. The ROM version determines the method of finding the address of SerFlag. In ROM version 01 and later, you can get the address with a call to the Miscellaneous Tools GetAddr using a reference number of $000E. With ROM version 00 (the original IIGS ROM), the address of SerFlag is $E10104. Refer to the Apple II Miscellaneous Technical Note #7, Apple II Family Identification for information on identifying Apple IIGS ROM versions.
4. Once you have the correct address of SerFlag, preserve the byte's current value, then turn on the bits in the byte which reflect the port from which you are handling interrupts. The bits for the different ports are as follows (note the relationship of the bits of RR3 to SerFlag):

```
Port 1:    ORA    #%00111000
Port 2:    ORA    #%00000111
```

5. Initialize the SCC modes. The *Z8530 Serial Communications Controller Technical Manual* shows the order the SCC registers must be programmed. However, you must stray from the manual slightly due to the hardware implementation of the SCC in the IIGS. A typical initialization sequence to set the SCC up for asynchronous serial communications through channel B (the modem port) would look similar to the following:

SCC Register	Value	Comment
RR0	-	ensure synchronization with SCC
WR4	01000100	x16 clock, 1 stop, no parity
WR3	11000000	8 bit receive data, auto enables off, receiver disabled
WR5	01100010	DTR is active, 8 bit transmit data, no break, transmit disabled, RTS is inactive
WR11	01010000	no Xtal on channel B, receive and transmit clock = baud rate generator output
WR12	01011110	low byte of baud rate generator time constant = $5E - 1200 baud
WR13	00000000	high byte of baud rate generator time constant = $00 - 1200 baud
WR14	00000000	no local loopback or auto echo, /DTR follows inverted DTR bit in WR5, use /RTxC for baud rate generator clock, disable baud rate generator
WR14	00000001	enable the baud rate generator
WR3	11000001	receiver enabled
WR5	01101010	transmit enabled
WR15	00000000	no interrupts on this channel for now...

6. Tell the SCC which external and status conditions can cause an interrupt by setting the appropriate bits in WR15. This step is not needed unless you are setting bit 0 of WR1 (External/Status Master Interrupt Enable) in the next step.
7. Enable the interrupts modes you want by setting the appropriate bits in WR1 (00010011 for all SCC interrupt conditions).
8. Use ALLOC_INTERRUPT to add your interrupt handler to the operating system's interrupt vector table. The interrupt identification number returned by ALLOC_INTERRUPT is needed when you remove your interrupt handler.
9. Reenable the SCC Master Interrupt flag (WR9, bit 3). The value you should write to WR9 is 00001010.

The interrupt handling routine must conform to the rules listed in the *ProDOS 8 Technical Reference Manual* and *GS/OS Reference*, Volume 2.

When you get ready to shut down your application, you need to remove your interrupt handler. The steps for removing the SCC interrupt handler you installed are as follows:

1. Disable the SCC Master Interrupt Enable (`WR9`, bit 3) briefly while performing the next six steps. The value you should write to `WR9` is 00000010.
2. Disable all interrupts modes for your port by writing a $00 to `WR1`.
3. Remove any character that might be left in the receive data register by reading it once.
4. Clear any pending transmit overrun and external and status interrupts by writing 11010000 to `WR0`.
5. Clear any pending transmit interrupt by writing 00101000 to `WR0`.
6. Use `DEALLOC_INTERRUPT` to remove your interrupt handler from the operating system's interrupt vector table.
7. Restore `SerFlag` to its original value.
8. Reenable the SCC Master Interrupt flag (`WR9`, bit 3). The value you should write to `WR9` is 00001010.

Further Reference

- *Apple IIGS Toolbox Reference Manual*, Volume 1
- *Apple IIGS Firmware Reference Manual*
- *Apple IIGS Hardware Reference Manual, Second Edition*
- *GS/OS Reference*, Volumes 1 and 2
- *ProDOS 8 Technical Reference Manual*
- Apple II Miscellaneous Technical Note #7, Apple II Family Identification
- GS/OS Technical Note #9, Interrupt Handling Anomalies
- *Z8530 Serial Communications Controller Technical Manual* (Zilog Corporation)
- *Z85C30 Serial Communications Controller Technical Manual* (Advanced Micro Devices, Inc.)

Apple IIGS #19: Multichannel Output with the Apple IIGS Note Synthesizer

Revised by: Jim Mensch November 1988
Written by: John Worthington & Jim Merritt June 1987

This Technical Note discusses multichannel sound with the IIGS Note Synthesizer.

It is possible to play multichannel sound using the IIGS Note Synthesizer Tool Set. The Ensoniq Digital Oscillator Chip (DOC) supports 16 independent output channels. Since only the low three bits of the output channel number are available through the IIGS sound expansion connector, multichannel circuitry may only decode eight output channels (zero through seven). Output channel eight maps onto channel zero, channel nine onto channel one, etc., and this mapping continues through all 16 channels.

The setting of the high nibble of the `DOCMode` byte in a `waveform` of the `waveList` portion of the instrument definition determines the routing of output from a Note Synthesizer instrument to a particular channel (the actual `DOCMode` information is in the low nibble of the `DOCMode` byte). You may assign each separate element in a `waveList` to a different output channel to create multisampled instruments in which some samples play on the left speaker and others on the right.

Apple standards require stereo expansion cards to map all even output channels to the right and odd channels to the left. To be compatible with cards that decode more than two of the chip's output channels, software should use channel zero for right and channel one for left. This convention ensures that output is always positioned properly in the stereo space with channel zero information going to the right front and channel one information going to the left front.

Further Reference

- *Apple IIGS Toolbox Reference*, Volume 2
- *Apple IIGS Toolbox Reference Update*

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#20: Catalog of APW Language Numbers

Revised by: Matt Deatherage March 1990
Written by: Jim Merritt August 1987

This Technical Note formerly listed APW Language Number assignments, which correspond to auxiliary type values of file type $B0.
Changes since November 1988: This information is now documented in Apple II File Type Notes, specifically Notes of file type $B0.

The correspondence between APW Language Numbers and auxiliary type values for $B0 files is no longer one-to-one. Although all APW Language Numbers are stored with their source files in the auxiliary type field, there now exist assignments of auxiliary type values for file type $B0 which are not APW languages.

Therefore, the contents of this Note can now be found in the File Type Note for file type $B0, where all such assignments of either kind are still called "APW Language Numbers."

Further Reference

- File Type Note for file type $B0, Apple IIGS source code files

Apple IIGS
#21: DMA Compatibility for Expansion RAM

Revised by: Glenn A. Baxter November 1988
Written by: Jim Merritt August 1987

This Technical Note discusses the Apple IIGS Extended Memory Slot specification.

The Apple IIGS Extended Memory Slot specification provides for DMA access to no more than four rows of RAM on a single board through the CROW0 and CROW1 signals. Expansion board designs that involve more than four rows of RAM are not compatible with DMA accesses. Each of the four rows can hold either 256K or 1 MB of data. The designs of the Fast Processor Interface (FPI) and its successor, the CYA, impose this limit. Each row can be organized in any of the following configurations to yield the respective board capacities assuming there are no more than four rows:

Chips	Configuration	Board Capacity
8	256K x 1 DRAM	1 MB
8	1 MB x 1 DRAM	4 MB
2	256K x 4 DRAM	1 MB
2	1 MB x 4 DRAM	4 MB

The CROW0 and CROW1 signals properly decode the row addresses for both normal and DMA cycles. The Extended Memory Slot interface does not support the latching of bank address information off the data bus during a DMA cycle, and a card which attempts to latch the bank address will likely get the **last** CPU cycle's bank address. Getting the last address is not a problem if it accidently happens to be the bank to which you wish to talk, but this is rarely the case. The card gets the last CPU cycle's bank address because DMA essentially shuts off the CPU, so it cannot emit the bank address. The FPI and CYA, which contain the DMA bank address register ($C037), do not emit the DMA bank address either, thus preventing bus contention with the processor as it is being removed from that bus. The DMA bank address register inside the FPI affects the addressing and control information that the Extended Memory Slot sees; it does not affect the data bus. Therefore, during DMA, the bank address time is filled with what is essentially random bank address information. Using this random information could result in damaging the contents of the memory (destroying little things like the operating system).

Suppose a card were designed to latch the bank address directly from the data bus with the rising edge of the PH2 clock signal. It could use the bank address to derive the proper RAM row address and never bother with CROW0 and CROW1 at all. Directly latching the bank address

would permit the card to accommodate any desired RAM arrangement in 64K increments, including an odd number of rows. Although the technique is valid during CPU cycles, it does not work during DMA cycles since the FPI never emits the DMA bank address onto the data bus. During DMA cycles, any card that tries to latch the bank address directly, instead latches the bank address that was put on the data bus during the last CPU cycle, which is probably the wrong value.

Currently, there does not seem to be a solution for the DMA situation. There the possibility of "limited DMA compatibility." An example of a limited-compatibility card would be one with six banks of memory. It's lower four banks are DMA compatible since they use the CROW0 and CROW1 lines, but the upper two banks do not work properly with DMA. This limited approach should be safe, but it is not guaranteed since DMA cards are sometimes aware of the total system memory and may expect, quite reasonably, to have access to **all** of the memory when in fact it does not. There are currently no "memory intelligent" DMA cards, but that could change at any point. The best we can suggest at this time is for hardware developers to build only four-row cards allowing up to 4 MB of memory, which is sufficient for most current applications.

Further Reference

- *Apple IIGS Hardware Reference*

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#22: Proper Use of Dynamic Segments

Rewritten by: Eric Soldan & Andy Stadler September 1990
Written by: Guillermo Ortiz October 1987

This Technical Note discusses strategies that applications can use to deal with dynamic segments.

Changes since November 1988: Rewrote from scratch to address current problems.

When reading the documentation on dynamic segments, it initially appears that they are even **better** than sliced bread. While they are incredibly useful, there are two issues that make dealing with them somewhat tricky. The first involves loading a dynamic segment; the second involves unloading a dynamic segment. Everything else works fine.

Loading Dynamic Segments

Loading dynamic segments is supposed to happen automatically. You are supposed to be able to call the code in the dynamic segment, and the system automatically loads it. As long as there is enough RAM to load the segment, this is exactly what happens.

The problem arises when there isn't enough memory. Immediately you have a number of questions, such as "How do I know if it didn't load?" and "How is the not-enough-memory error returned?" Unfortunately, neither of these questions is applicable. Instead, you get a Fatal System Error, which is not the most useful thing that could happen.

However, there are some reasons for this error. For example, in the Pascal or Toolbox stack frame system, the called function is responsible for removing the parameters pushed onto the stack. If the dynamic segment did not load, these parameters cannot be pulled from the stack, and if they are not pulled from the stack, the operating system cannot return to the caller.

Due to this problem, the best thing to do is to try to load the dynamic segment with `LoadSegName`. If it loads, then there is (obviously) enough RAM for it. If it does not load, then there was not enough RAM; it's that simple. So, to call a function named `dynFN` in a dynamic segment called `dynSeg`, you would do the following:

```
LoadSegName("\pDynSeg");
if (!_toolErr) {
    dynFN(some, number, of, parameters);
    UnLoadSeg(dynFN);
```

```
    }
    else ErrorAlert("\pOut of RAM.");
```

Unloading Dynamic Segments

`UnLoadSeg` used to have a problem, so the above technique would not have worked. As of System Software 5.0.3, this problem has been fixed. In the example, the code `UnLoadSeg(dynFN)` does not pass the address of the `dynFN` that was loaded into RAM. Instead, that address represents the entry in the dynamic segment jump table for that particular function. The jump table is always in RAM. So, you are not actually passing an address of the segment to be unloaded, but an address in the jump table.

The loader is responsible for figuring out that the address is actually an address in the jump table, and it is supposed to unload the segment to which the jump table entry refers. The loader did not handle this case properly until 5.0.3. So, for system disks prior to System Disk 5.0.3, you can preserve the segment number returned by the `LoadSegName` call to issue an `UnLoadSegNum` call to dispose of the dynamic segment. Due to `UnLoadSeg` not doing the job prior to 5.0.3, you could use `UnLoadSegNum`. This also has problems. ExpressLoad changes the segment numbers, so it is difficult to maintain the segment numbers if you change the link script. For these reasons, the below technique should be used for system disks prior to 5.0.3:

```
void   sample()
{
       struct    LoadSegNameOut    dynSegInfo;

       dynSegInfo = LoadSegName("\pDynSeg");
       if (!_toolErr) {
           dynFN(some, number, of, parameters);
           UnLoadSegNum(dynSegInfo.segNum);
       }
       else ErrorAlert("\pOut of RAM.");
}
```

Dynamic Segment Interdependencies: Just Say No

Dynamic Segments calling each other almost **always** lead to unloading conflicts, and more importantly, they defeat the purpose (if they both have to be in simultaneously then they might as well be static). Figure 1 is a sample program layout you may want to consider when designing your application dynamic segment usage:

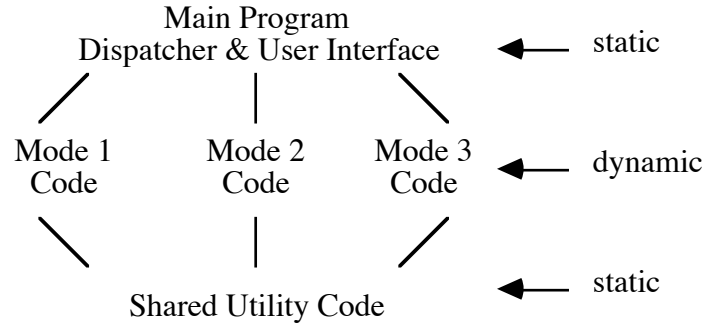

Figure 1–Sample Program Layout

Also, if one of the dynamic segments described is much more than, say, 32K or 40K, you may wish to load a pair (or more) of dynamic segments. These dynamic segment pairs would always be loaded and unloaded simultaneously. Why? Because loading two 25K segments is more likely to succeed than loading one 50K segment.

A Final Warning:

Data in a dynamic segment is a tricky issue. When you call a dynamic segment, you are not sure if it got loaded, or if it was already in RAM, and therefore you cannot be sure of the values in your global data. For example, say that you have a global variable that represents the number of times that you call the dynamic segment. Every time you call the segment, you would increment this variable. This technique works great until the dynamic segment gets purged. Once it is purged, the next time you call it, the variable area would be loaded from disk again, with its original initial value. The count is no longer valid. To fix this, you can place the global could variable in the static globals space for the main code. Then the variable would not get purged, and your count would be valid. Of course, if you have global data that does not ever change, then it is okay for the data to be in the global segment.

Further Reference

- *GS/OS Reference*
- *Apple IIGS Programmer's Workshop Assembler Reference*

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#23: Toolbox Use of DOC RAM

Revised by: Matthew Denman & Matt Deatherage November 1988
Written by: Jim Merritt October 1987

This Technical Note explains why you must be careful about which values you store in the first page of the Ensoniq Digital Oscillator Chip (DOC) RAM when using Note Synthesizer and MIDI Tool Sets on the Apple IIGS.

The Apple IIGS Note Synthesizer uses an oscillator as a free-running timer to clock the update of waveform envelopes when the DOC sounds notes. To act as a timer, the oscillator "plays" the contents of bytes $00 – $FF in DOC RAM at zero volume. Once it scans through the entire "waveform buffer," the oscillator generates an interrupt, which the appropriate Note Synthesizer routines service.

When using the Note Synthesizer or the Note Sequencer without the MIDI Tool Set, there is no need to avoid using DOC RAM locations $00 – $FF for general waveform storage. More than one oscillator can play from the same waveform buffer at the same time, so the function of the timer oscillator does not affect normal use of the DOC for sound generation purposes in any way. However, you should not fill the first page of DOC RAM with waveforms that are delimited by zero bytes (as is sometimes appropriate in special situations, discussion of which is beyond the scope of this Note). The presence of zero bytes in the first page of DOC RAM can cause serious system performance degradation and can even cause the system to hang. In particular, it is always inappropriate to store arbitrary, non-waveform data in the first page of DOC RAM since such data often includes zero bytes (which would be corrupted were you to remove or modify them).

The Apple IIGS MIDI Tool Set also uses bytes $00 – $FF of DOC RAM for timing purposes, but it uses a different oscillator than the Note Synthesizer. If you want MIDI time stamping, you may not use the first page (bytes $00 – $FF) of DOC RAM for your own purposes since the MIDI Tool Set uses the contents of those bytes for time-stamping purposes.

You may use the MIDI, Note Synthesizer, and Note Sequencer Tool Sets together, but you must not use bytes $00 – $FF of DOC RAM for any purpose if using MIDI time stamping, nor store zero bytes in this area when using the Note Synthesizer. You might consider it appropriate to avoid using the first page of DOC RAM, if possible, to facilitate adding MIDI support to your application at a later date.

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#24: *Apple IIGS Toolbox Reference* Updates

Revised by: Dave Lyons — May 1992
Written by: Rilla Reynolds, Matt Deatherage, Dave Lyons, C.K. Haun & Eric Soldan — October 1987

This Technical Note documents changes to the *Apple IIGS Toolbox Reference* manuals. Please contact Apple II Developer Technical Support at the address listed in Apple II Technical Note #0 if you have additional corrections or suggestions for any of the Apple IIGS Toolbox documentation.

Changes since December 1991: Added corrections to Dialog Manager, Menu Manager, Tool Locator, Window Manager, and Appendix E.

The current Apple IIGS Toolbox reference material is *Apple IIGS Toolbox Reference*, volumes 1 to 3 as well as this Technical Note. (The *Apple IIGS Toolbox Reference Update* beta draft from APDA is obsolete and should **not** be used.)

Corrections to Volume 1

Desk Manager—FixAppleMenu Can Die With Error $0512

Fatal system error $0512 comes from `FixAppleMenu` (in the Desk Manager). It means that one of your installed New Desk Accessories does not have a well-formed menu title string. In particular, the required backslash (\) character was not found (make sure bit seven is off).

Dialog Manager—`editLine` Item Value

On page 6-12, the description of an `editLine` item value should read "Maximum length of the item text (0 to 255 characters)."

The List Manager Wants the Port Set Properly

The List Manager expects the current `grafPort` to be set properly before you make most List Manager calls; drawing can occur in funny places if the `grafPort` is not set properly before calls that draw (like `SelectMember2`). Most List Manager calls, and many other toolbox calls, require that the current `grafPort` be explicitly set. Before you call List Manager routines that draw, set the current port to your window with a `SetPort` call. Remember the note in Volume 2 under the `NewWindow` call—"Important: `NewWindow` does not set the current port, but many routines require that a current port exist. Use the QuickDraw II routine

`SetPort` to set the current port." Using `SetPort` can prevent toolbox confusion and reduce your debugging time.

DeleteMItem Operates on the Current Menu Bar

Page 13-37 says `DeleteMItem` removes the specified item from the current menu. It means the item is removed from the current menu bar.

Error $0F02 from GetMItem

`GetMItem` returns error $0F02 if the specified menu item is not found.

On page 13-45, the return value from `GetMenuFlag` should read "Word—`menuFlag` value for the specified menu."

On page 13-56, in the description of the `hiliteFlag` parameter to `HiliteMenu`, no particular value of "TRUE" is specified. $0001 is a good value ($8000 does not work; bit 15 is special).

On page 13-72, `SetMenuFlag` doesn't bother to actually explain what it does. If bit 15 of `newValue` is zero, each set bit set forces the corresponding bit in the menu's flag value to be set. If bit 15 of `newValue` is one, each clear bit forces the corresponding bit in the menu's flag value to be clear. Knowing this, you can set or clear more than one bit at a time, if you want.

SetVector Reference Numbers

On page 14-62, vector reference number $002C is listed as "Message pointer vector." $002C is actually the stack-based GS/OS call vector. (The real message pointer vector is not accessible through `GetVector` and `SetVector`.)

Getting a clean Mouse Mode from `ReadMouse`

On ROM 3 computers, the mouse mode byte returned from `ReadMouse` sometimes has extra bits set in the high nibble. Before feeding a `ReadMouse` value to `SetMouse`, mask off all but the low nibble (`AND #$000F`).

ReadAsciiTime Result Buffer

The description of `ReadAsciiTime` (in the Miscellaneous Tools) on page 14-16 should say the most significant bit (not byte) of each character is set to one.

SystemEvent Is All Backwards

Although applications still should not call `SystemEvent`, we should note for completeness that the input parameters listed in Volume 1 are exactly backwards in the stack diagram.

Corrections to Volume 2

QuickDraw Auxiliary Error Codes

Following are some error codes from QuickDraw Auxiliary that are not listed in volume 2.

```
$1210: picEmpty
$1211: picAlreadyOpen
$1212: pictureError

$1221: badRect
$1222: badMode
```

FrameRgn Does Not Contribute to an Open Region

The description of the `FrameRgn` routine on page 16-105 in the *Apple IIGS Toolbox Reference*, Volume 2 states that `FrameRgn` will contribute to a region definition if a region is open when `FrameRgn` is called. This is **incorrect**; `FrameRgn` does not contribute to the region being defined. To add a region to another region, use `XorRgn` or `UnionRgn`.

Tool Locator, `TLMountVolume`

On page 24-21, the description of `TLMountVolume` does not bother to mention that QuickDraw II and Event Manager must be active. If they are not, you should use `TLTextMountVolume` instead.

Tool Locator, `SetTSPtr`

When using `SetTSPtr` to patch a system tool set, the Tool Locator and Desk Manager are special. See Apple IIGS Technical Note #101, Patching the Toolbox.

Window Manager, "Draw Information Bar Routine"

On page 25-23, the code to clean up the stack is incorrect. On the `sta <14`, the comment "Works because stack and direct page are equal" is no longer true—they **were** equal until the `PLY` two lines earlier. One way to correct the code is to replace `sta <14` with `sta 14,s` and `sta <12` with `sta 12,s`.

Window Manager, `InvalRect`

The description of `InvalRect` on page 25-80 claims that `InvalRect` modifies the input rectangle; the rectangle is actually not modified.

Window Manager, PinRect

On page 25-89, in the description of `PinRect`, the two greater-than comparisons should be greater-than-or-equal.

Window Manager, SetZoomRect

The description of SetZoomRect on page 25-112 refers to fZoomed as bit 2 in the window frame. fZoomed is actually bit 1, with value $0002.

Window Record Offsets

On page 25-142, note that the offsets given into the window record refer to the record as the Window Manager treats it internally, with a wNext field at the beginning. When dealing with a window pointer as seen by an application, you need to subtract four from the offsets shown. For example, wPort is $00 (not $04), and wControls is $C6 (not $CA).

Appendix A, "Writing Your Own Tool Sets"

At the bottom of page A-8, "lda #$90" should read "lda #$8100" for version 1.0 prototype.

On page A-10, the figure should show **two** RTL addresses (6 bytes) on the stack.

Corrections to Volume 3

Control Manager: Menu Events

On page 28-15, note that a Menu Event is identified by the value wInSpecial ($0019) in the what field of the task record. The menu item ID is in the low word of the wmTaskData field.

Control Manager: Dimmed Custom Controls

In the Draw routine for both extended and non-extended controls, the high word of ctlParam (which was previously undocumented) contains a flag which the definition procedure can use to draw a normal or dimmed control. The value is $0000 normally, but it is $FFFF when the control is inactive (hilite value equals $00FF), or when the control's state is tied to the window's state and the window is inactive.

Control Manager: Size Box Controls

The part code for an extended Size Box control is normally 10. If the fCallWindowMgr bit is set in ctlFlag, the part code is $80; and if the size box is managed by a Text Edit control, the part code is $84.

When a Size Box control's fCallWindowMgr bit is set, the control needs to pass a minimum window size to GrowWindow. It gets this value from its ctlData field, which you can get with GetCtlTitle and set with SetCtlTitle (the low word is the minimum height, and the high word is the minimum width). A height of zero defaults to 50, and a width of zero defaults to 130.

Desk Manager: Errors from AddToRunQ and RemoveFromRunQ

The Desk Manager chapter, page 29-6, states no errors are possible for `AddToRunQ`, but any errors from the Miscellaneous Tools routine `AddToQueue` are returned unchanged.

Page 29-8 states no errors are possible from `RemoveFromRunQ`, but any errors from `DeleteFromQueue` are returned unchanged.

Event Manager: What SetAutoKeyLimit Really Does

Page 31-6 says that `PostEvent` will add up to the new auto-key limit number of auto-key events before reverting to the rule that auto-key events are only to be posted if the event queue is empty. This is not quite right. Actually, the parameter to `SetAutoKeyLimit` is used in a size comparison on the event queue—if there are `newLimit` or more events in the queue, auto-key events will not be posted. Volume 3 incorrectly states that up to `newLimit` auto-key events will be posted; this is only true if you assume the event queue is empty before the first auto-key event comes in.

List Manager

On page 35-9, the description of `ResetMember2` does not point out an important difference between `ResetMember2` and `NextMember2`. `ResetMember2` deselects the member found, but `NextMember2` does not change the member's status.

On page 35-3, bit 5 of the `memFlag` field is defined—it makes an item inactive. To make use of this bit, you must also set bit 6 of the List control's `ctlFlag` field; if you don't set this bit, the user will still be able to select members using the mouse.

Memory Manager

If the Memory Manager detects a corrupted entry in the Out Of Memory Queue, fatal system error $0209 occurs.

Menu Manager

On page 28-65, the description of the `initialValue` field is misleading. Cross out the text "that is, its relative position within the array of items for the menu." `initialValue` is simply a menu item ID, not an offset into an array.

Page 37-7 states "Because caching does not work with menus in windows, the `InsertMenu` call automatically disabled caching for such menus." Actually, `InsertMenu` doesn't do that. You should not set the `allowCache` bit for a menu in a window.

Miscellaneous Tools: Interrupt State Record Not Always Complete

The interrupt state record returned from `GetInterruptState` (and passed to `SetInterruptState`) is not always completely filled in. The Interrupt Manager, in the interest of serving AppleTalk and serial interrupts as rapidly as possible, does not take the time to save all the items in the record until those timing-critical interrupt handlers have been called. Some items are not saved at all unless the interrupt is determined to be a `BRK` instruction. Table 1 shows all items in the current interrupt state record and when they become valid:

Record variable	When valid
`irq_A`	always
`irq_X`	always
`irq_Y`	always
`irq_S`	after serial
`irq_D`	always
`irq_P`	only on break
`irq_DB`	after serial
`irq_e`	after serial
`irq_K`	only on break
`irq_PC`	only on break
`irq_state`	after serial
`irq_shadow`	always
`irq_mslot`	after serial

Table 1—Validity of Interrupt Record

Standard File

On page 48-39, the description of `origNameRef` reads "On output, this string contains the string confirmed by the user, which may not be the same length as the default value." This sentence is confused; ignore it. The string is not changed at all; Standard File doesn't even know how long the buffer is.

Tool Locator: Notes on StartUpTools

`StartUpTools` in System Software 5.0.4 and earlier is intended to be used from applications only, not from NDAs.

The order of the `toolArray` entries in the `StartStop` record is not important. `StartUpTools` and `ShutDownTools` always start up and shut down tools in a correct order.

`StartUpTools` in System Software 5.0.4 and earlier fails to open your application's resource fork if the application's filename contains a slash (/) or if the application directory path is longer than 64 characters.

For maximum compatibility, pass your application's master user ID with any `auxID` to `StartUpTools` instead of allocating a new user ID.

Window Manager:NewWindow2 Parameters Override Template Even When You Pass NIL

The description of the NewWindow2 call on page 52-32 is in error. The description of the titlePtr, refCon, contentDrawPtr, and defProcPtr says, "To prevent NewWindow2 from replacing the template values, supply NIL pointers…" This is only true for the titlePtr parameter—if you pass NIL for any of the other parameters then the value of that parameter in your window record is also NIL, no matter what the template value was. In other words, if you have the value $99 stored in your template refCon field, and you pass NIL for the refCon value in a NewWindow2 call, the value of the refCon in the returned grafPortPtr is NIL.

Appendix E: `rTextForLETextBox2` Resources

Page E-68 of Volume 3 shows a `length` field at the beginning of an `rTextForLETextBox2` resource. This field is not actually present. The length is simply the size of the resource—it is not stored redundantly.

Appendix E: rTwoRects Resources

When the two rectangles are for 320- and 640-mode, by convention the rectangle for 320 mode comes first.

Further Reference:

- *Apple IIGS Toolbox Reference*, Volumes 1–3
- Apple IIGS Technical Note #101, Patching the Toolbox

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#25: *Apple IIGS Firmware Reference* Updates

Revised by: Dave Lyons May 1992
Written by: Rilla Reynolds, Dave Lyons, & Jim Luther October 1987 to September 1990

This Technical Note includes updates to the May 1987 edition of the Apple IIGS Firmware Reference, published by Addison-Wesley (Part Number 030-3121-A). The new Monitor commands require an Apple IIGS revised ROM (Part Number 342-0077-B), which is available without charge from an authorized Apple dealer. Please contact Apple II Developer Technical Support at the address listed in Apple II Technical Note #0 if you have additional corrections or suggestions for this manual.

Changes since September 1990: Added a reference to Apple IIGS Technical Note #102 for `TOBRAMSETUP`.

Contents

Page vii, Chapter 7 SmartPort Firmware: Change "Generic SmartPort calls 121" to "Standard and Extended SmartPort calls 121."

Chapter 2: Notes For Programmers

Page 11, Environment for the Firmware Routines: Refer to Apple IIGS Technical Note #88, The Page One Stack in a 16-Bit World for more information on manipulating the stack pointer.

Chapter 3: System Monitor Firmware

Page 24, Table 3-1 (continued), Monitor commands grouped by type: Add a miscellaneous-type and a debugging-type Monitor command to the table, as follows:

Command type	Command format
…	
Quit Monitor	Q
Install Visit Monitor and MemoryPeeker desk accessories	*#*
…	
Enter mini-assembler	!
Set flags (e, m, x) for full-native mode	*Control-N*

Page 43, Back to BASIC: The last paragraph should read: "If you are using DOS 3.3 or ProDOS®, use the Monitor Q (Quit) command to return to the language you were using with your program and variables intact."

Page 48, Table 3-6, Commands for program execution and debugging: Add a Monitor command to the table:

Command type	Command format
…	
Enter mini-assembler	!
Set flags (e, m, x) for full-native mode	*Control-N*

Page 66, after final paragraph: Add a new Monitor instruction heading and description:

Native Mode Set Control-N (Native Mode)

Control-N sets the m, x, e flags to 0 for full-native mode. All other registers are unchanged.

Page 67, after final paragraph: Add a new Monitor instruction heading and description:

Turn on ROM Desk Accessories, #

Enables the currently available ROM desk accessories, Visit Monitor and Memory Peeker. These desk accessories remain active in the desk accessory menu until power is shut off. Control-Open Apple-Reset has no affect on these items. To exit the Visit Monitor desk accessory, press Control-Y then press Return. To exit the Memory Peeker desk accessory, press Q.

Chapter 4: Video Firmware

Page 77, Table 4-4, Control characters with 80-column firmware on: Change the actions taken by Control-E and Control-F to read (they were reversed):

Control character	Action taken by C3COUT1
Control-E	Turns cursor on
Control-F	Turns cursor off

Chapter 5: Serial-Port Firmware

Page 82, Compatibility: The second half of the third sentence in the first paragraph should read: "…the Apple IIGS hardware *is different* from that used on the SSC."

Page 91, Input buffering, BE and BD: This heading should be "Input/Output buffering, BE and BD."

Page 94, Table 5-6: The Extended Interface footnote which states, "If the function call returns with the carry bit set..." is incorrect. For Apple IIGS ROM 01, the Extended Serial Interface does not return the error condition in the carry bit. Programs using the Extended Serial Interface should check for a non-zero result value in the result code rather than the carry bit to determine if an error has occurred. For additional error handling information using the Extended Interface, see Apple IIGS Technical Note #50, Extended Serial Interface Error Handling.

Page 95, Error handling: The second sentence should read: "If the character has a framing or parity error (assuming that the parity option is not set to None), the character is deleted from the input stream and the appropriate *mode bit* is set."

Page 96, Note: The Note should read: "The InQStatus elapsed-time counter functions correctly only if *a* heartbeat interrupt task has been started. *A* heartbeat interrupt task is a set of functions called by interrupt code that run automatically at one-thirtieth of a second intervals.

Page 96, Interrupt notification: The fourth sentence in the first paragraph should be: "The system interrupt handler will transfer control to the user's interrupt vector *at* $03FE in bank $00."

Page 97, Interrupt notification: The last three paragraphs should be replaced with this paragraph: "The interrupt completion routine executes as *part of the firmware interrupt handler* and must be run in that environment. The interrupt completion routine must preserve the DBR, speed, 8-bit native mode, `D` register, stack pointer (or just use the current stack), and `MSLOT` for proper operation. `A/X/Y` need not be preserved."

Page 100, SetModeBits: The first sentence in the paragraph following the `CMDLIST` should read: "Use this call to alter any of the mode bits whose function is described *below*."

Page 105, GetIntInfo: The command list should read:

```
CMDLIST     DFB     $03                 ;Parameter count
            DFB     $0C                 ;Command code
            DW      $00                 ;result code (output)
            DW      $00                 ;interrupt setting (output)
            DL      Completion address  ;(output)
```

The following should be added after the command list: "Note: The Parameter count of $03 is correct even though there are four parameters."

The following should be added after the last paragraph: "Note: Before making this call from an interrupt completion routine, you must set the operating environment to look and act exactly like a 6502 in all respects. During interrupt completion routines, you must preserve the DBR, speed, 8-bit native mode, `D` register, stack pointer (or just use the current stack), and `MSLOT` for proper operation. `A/X/Y` need not be preserved. See "Environments for the Firmware Routines" in chapter 2, Notes for Programmers for details about setting and restoring the operating environment.

Page 106, SetIntInfo: The command list should read:

```
CMDLIST     DFB     $03                 ;Parameter count
            DFB     $0D                 ;Command code
            DW      $00                 ;result code (output)
            DW      Interrupt setting   ;(input)
            DL      Completion address  ;(input)
```

The following should be added after the command list, "Note: The Parameter count of $03 is correct even though there are four parameters."

Chapter 7: SmartPort Firmware

Page 120, Issuing a call to SmartPort: The standard and extended SmartPort call examples should be:

This is an example of a standard SmartPort call:

```
SP_CALL         JSR     DISPATCH          ;Call SmartPort command dispatcher
                DC      i1'CMDNUM'        ;This specifies the command type
                DC      i2'CMDLIST'       ;Word ptr to param list in bank $00
                BCS     ERROR             ;Carry is set on an error
```

This is an example of an extended SmartPort call:

```
SP_EXT_CALL     JSR     DISPATCH          ;Call SmartPort command dispatcher
                DC      i1'CMDNUM+$40'    ;This specifies the ext cmd type
                DC      i4'CMDLIST'       ;Pointer to the parameter list
                BCS     ERROR             ;Carry is set on an error
```

Page 121, Generic SmartPort calls: Change occurrences of "Generic SmartPort Calls" to "Standard and Extended SmartPort Calls" in the header and the first sentence. Refer to SmartPort Technical Note #2, SmartPort Calls Updated, for updated information on the SmartPort `STATUS` call.

Page 122, Statcode = $00: Change the function of bit 0 of the first device status byte to: "1 = Device currently open (character devices only) or disk switched (block device only)."

Page 124: SmartPort device types should be same as those documented in SmartPort Technical Note #4, SmartPort Device Types.

Page 125, SmartPort driver status: See SmartPort Technical Note #2, SmartPort Calls Updated, for the correct format of the status list for unit 0, status code 0.

Vendors must request a Vendor ID Assignment from Developer Technical Support before using a specific value in bytes two and three.

Page 125, Possible errors: Add the following:

$1F No interrupt. Interrupts not supported.
$2B No write. Disk write-protected.
$2F Offline. Disk off-line or no disk in drive.

Page 126, ReadBlock: Add a sentence at the end of the first paragraph which reads, "On return, the X and Y registers indicate the number of bytes transferred."

Page 131, Open: The following changes apply for the CMDNUM:

	Standard call	**Extended call**
CMDNUM	*$06*	*$46*

Page 132, Read: Add a sentence at the end of the first paragraph which reads, "On return, the X and Y registers indicate the number of bytes transferred."

Page 140, Figure 7-8, Disk-sector format: Change to the following:

13 5-Nibble SelfSync Fields	FF	D5	AA	96	Track	Sector	Side	Format	AdrsLRC	DE	AA	FF	1 5-Nibble SelfSync Field	FF	D5	AA	AD	Sector	699 GCR Nibbles	4 Checksum	DE	AA	FF

A SelfSync Field is four 20us selfsync nibbles written as a sequence of five 16us nibbles.

Page 140, ResetHook: The Control code and Control list should be:

Control Code	Control list	
$06	Count low byte	*$04*
	Count high byte	$00
	Hook reference number	*$xx, $00, $00, $00*

Page 141, SetInterleave: The Control code and Control list should be:

Control Code	Control list	
$0A	Count low byte	*$01*
	Count high byte	$00
	Interleave	$01 to $0C

Page 143, UniDiskStat: The Status code and Status list should be:

Status Code	Status list	
$05	Byte	$04
	Soft error	$00
	Retries	*$xx*
	A register after execute	*$xx*
	Y register after execute	$xx
	P register after execute	$xx
	Byte	*$xx*

Page 152, Passing parameters to a ROM disk: Add a sentence to the end of the second paragraph which reads: "These locations will not be preserved between SmartPort calls."

Page 156, Table 7-6, SmartPort error codes: Add the following error code:

Acc value	Error type	Description
$69	IOTERM	I/O terminated due to new line

Page 166, Table 7-8, Standard command packet contents":
Byte 3 descriptions should read "Byte 2 of param list."
Byte 4 descriptions should read "Byte 3 of param list."
Byte 5 descriptions should read "Byte 4 of param list."
Byte 6 descriptions should read "Byte 5 of param list."
Byte 7 descriptions should read "Byte 6 of param list."
Byte 8 descriptions should read "Byte 7 of param list."
Byte 9 descriptions should read "Byte 8 of param list."

Chapter 8: Interrupt-Handler Firmware

Page 184, Serial-port interrupt notification: The last three paragraphs should be replaced with this paragraph: "The interrupt completion routine executes as *part of the firmware interrupt handler* and must be run in that environment. The interrupt completion routine must preserve the DBR, speed, 8-bit native mode, `D` register, stack pointer (or just use the current stack), and `MSLOT` for proper operation. `A/X/Y` need not be preserved."

Chapter 9: Apple DeskTop Bus Microcontroller

Page 191, Sync, $07: The first sentence should read: "This command performs *the three preceding commands in sequence*."

Page 194, Receive Bytes, $48: The fourth sentence should read: "The second byte value is a combination of *the device address in the high nibble and the ADB command in the low nibble* (see the *Apple IIGS Hardware Reference*)."

Chapter 10: Mouse Firmware

Page 201: Mouse button positions should be changed as follows:

- **X data byte**
  If bit 7 = 0, then mouse button 1 is *down*.
  If bit 7 = 1, then mouse button 1 is *up*.

- **Y data byte**
  If bit 7 = 0, then mouse button 0 is *down*.
  If bit 7 = 1, then mouse button 0 is *up*.

Page 205, Figure 10-1, Position and status information:
Bit 7 description should be: "Currently, *button 0* is up/down (0/1)."
Bit 6 description should be: "Previously, *button 0* was up/down (0/1)."

Appendix B: Firmware ID Bytes

Page 223, Table B-2, Register bit information: Change the table to show that Bits 7-0 of the *Y* register hold the ROM version number, and the *X* register is reserved. In addition, the table description should read: "The Y register contains the machine ID and the ROM version number. The X register is reserved."

Page 249, COUT1: In the third sentence, change the value of line feed from $8C to $8A.

Page 277, RDALTZP: Change the comment to read: "Bit 7 = 1 if *alt* zp enabled."

Appendix D: Vectors

Page 272: At the end of the introductory paragraph, add "The vectors `TOWRITEBRAM` through `TOPRINTMSG8` must be called in eight-bit native mode."

See Apple IIGS Technical Note #102, Various Vectors, for more information about the `TOBRAMSETUP` vector.

Further Reference:

- *Apple IIGS Firmware Reference*
- *Apple IIGS Firmware Reference 1MB Apple IIGS Update*
- Apple IIGS Technical Note #50, Extended Serial Interface Handling
- Apple IIGS Technical Note #102, Various Vectors
- SmartPort Technical Note #2, SmartPort Calls Updated