Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#1: How to Install Custom BRK and /NMI Handlers

Revised by: Jim Mensch & Jim Merritt November 1988
Written by: Jim Merritt October 1986

This Technical Note discusses a method to install a custom debugger or debugging stub within the Apple IIGS system.

Introduction

This Technical Note discusses a particular method that you may use to install a custom debugger or debugging stub within the Apple IIGS system. The strategy and techniques described here should be of special interest to those who wish to operate the Apple IIGS as a slave to a debugger that resides on another machine.

Typically, an interrupt handler should pass control to a debugger or debugging stub whenever the processor executes a `BRK` instruction, or when an interface card triggers a non-maskable interrupt (/NMI). To simplify the design of the debugger, the Apple IIGS Monitor should be responsible for the following:

- saving all machine state information in locations that the debugger can access
- setting the machine to a known state
- passing control to an arbitrary debugger
- restoring the remembered machine state upon regaining control from the debugger
- resurrecting the interrupted process

The Monitor is designed to provide all of the services above for the `BRK` instruction, but only the third for /NMI interrupts. In addition, Apple II family systems are generally intolerant of /NMI interrupts. In this Technical Note we concentrate on the means by which you can install your own custom `BRK` handler, although we also briefly examine /NMI considerations.

Dealing With BRK

A `BRK` interrupt handler may reside at any address in memory. The Monitor passes control to your code by executing a `JSL` instruction; consequently, your routine must terminate with an `RTL` instruction. To install your `BRK` handler, simply load it into memory, call the Miscellaneous Tool Set `GetVector` routine to fetch the address of the current `BRK` handler, put that address in a safe place, then supply the address of your handler to the Miscellaneous Tool Set `SetVector` routine. To deactivate your handler, restore the previous handler address using `SetVector` as follows:

```
;
;  NOTE: All Listings are in APW assembler format.
;

INSTMYBRK     anop                 ;Example code to install user's BREAK handler.
              PushLong #0          ;Space for function call result.
              PushWord #$1C        ;We want BREAK vector address.
              _GetVector           ;Make the call using standard macro.

;  The stack now holds address of the current break handler.
              PLA                  ;Get and save low word of address…
              STA    SBRKADR
              PLA                  ; …and now high word.
              STA    SBRKADR+2
              PushWord #$1C        ;We want to change BREAK vector address.
              PushLong #MYHANDLR   ;Address of user's BRK handler.
              _SetVector           ;Make the call using standard macro.

;  Custom handler is in place, now go off and do whatever we like…

DEACMYBRK     anop                 ;Example code to deactivate the BRK handler.
              PushWord #$1C        ;We want to change BREAK vector address.
              PushLong SBRKADR     ;The previous BRK handler address.
              _SetVector           ;Make the call using standard macro.
```

Upon entry to your code, the machine will be in eight-bit native mode. Specifically, the m and x bits will be set (forcing eight-bit accumulator, memory access, and index registers), the processor will be running at the normal (1 MHz) speed, all memory shadowing will be enabled, and both the direct page and data bank registers will be reset to zero. The same conditions must hold when your `BRK` handler returns control to the Monitor. While your code is active, however, it is free to affect the machine state in arbitrary ways, including (but not limited to) widening the registers, increasing the clock rate, and disabling shadowing. Before returning control to the Monitor, your break handler must also clear the processor's carry flag, as an indication that the `BRK` was indeed serviced by an external handler. (Note: The default `BREAKVECTOR` points to a "no-op" handler that simply sets the carry flag to indicate that there is no external handler available, and it then executes an `RTL`.)

When a `BRK` occurs, the processor saves the machine's state in the `BRK.VAR` area, and you may obtain this address with the Miscellaneous Tool Set `GetAddr` routine as follows:

```
              PushLong #0          ; space for result
              PushWord #9          ; we want BRK.VAR address
              _GetAddr             ; make the call using standard macro

; The stack now holds the address of the BRK.VAR area, expressed as a long word (four
bytes).
```

Coping With /NMI

Handling /NMI interrupts is, by far, a trickier proposition than fielding `BRK` instructions. For example, the user-definable /NMI jump-vector, `/NMI` ($0003FB), only has room in its three-byte `JMP`-absolute instruction for a two-byte address. Because of this size limitation, at least the "front end" of any /NMI handler must reside in bank $00. In addition, the Monitor does not "condition" the system in any way before transferring control through the /NMI hook, so the system could be in native mode, emulation mode, or any hybrid mode (with any screen condition) upon entry to your handler. (Note: Although the 65816 processor provides for separate /NMI vector addresses in native and emulation modes, the Apple IIGS implementation of these two vectors pass control to the same user hook at $0003FB.) The processor only saves minimal machine state information when an /NMI occurs; if the handler needs to preserve more than the program counter and status register (which are saved automatically), then it must do so explicitly. Because the 65816 assumes any program running in emulation mode has its program bank register in bank zero, it will not save the program bank register for any program running in emulation mode outside of bank zero. Code which runs in this manner will always crash if it makes <u>any</u> attempt to return from the interrupt. Finally, /NMI interrupts can create havoc with disk access and other aspects of the system; consequently, the only way you can safely use /NMI interrupts is as a one-way "escape hatch" to emergency debugging code.

Here are some ground rules for /NMI interrupt handlers.

- On entry, store any interesting registers or machine state in RAM space owned by the handler.
- Determine whether the processor is in emulation mode or native mode.
- Take appropriate action, depending upon the processor mode.
- <u>Under no circumstances</u> try to return from the interrupt! Restart the system instead.

To install an /NMI handler, load it into some free RAM in bank $00, put the two-byte address currently at location /NMI+1 in a safe place, then replace it with the address of your handler. To deactivate your handler (assuming nothing has yet invoked it), simply restore the previous handler address to /NMI+1.

Apple IIGS
#2: Transforming I/O Subroutines for Use in “Native” Mode

Revised by: Pete McDonald November 1988
Written by: Pete McDonald October 1986

This Technical Note outlines a number of techniques useful when transforming Apple II I/O subroutines for use in the “native” Apple IIGS environment.

The Apple IIGS execution environment represents quite a departure from the environment to which the average Apple II developer is accustomed. This fact results in a number of unique problems when one attempts to convert existing Apple II applications for use in the “native” Apple IIGS environment. (Note: If you intend to let your application remain an eight-bit “classic” Apple II application, then you can ignore the information this Technical Note presents.)

I/O subroutines which depend upon critically timed code present some of the biggest conversion problems due to two major issues. In the native IIgs environment, you cannot guarantee that there will be memory available in a given bank, and I/O locations are not available in every bank.

There are a number of possible solutions to this problem. Which ones you should use depend upon what the program in question is doing. This Note attempts to describe some of the problem situations and possible solutions.

Examine the 6502 code segment below. It serves no useful purpose, other than to illustrate a simple manifestation of the problem. Assume `IoLoc` is a location in the $C000 – $CFFF range of memory.

```
Loop           LDA           IoLoc
               DEY
               BPL           Loop
```

Because the $C000 – $CFFF range of memory in bank 2 or higher contains RAM instead of I/O circuitry unless hardware shadowing is enabled, if you place the fragment above in one of these banks, it will have no effect on the I/O device you intend it to control.

There are two possible solutions in this case. Either change the instruction `LDA  IoLoc` so it uses long addressing, thereby forcing the CPU to reference the the proper bank. (Note: The problem with this is the long version of `LDA` requires an extra CPU cycle to execute. If the code segment is timing critical, then this method is likely to be unacceptable.) Alternately, in the timing-critical case, we could set the data bank register before entering the loop which would mean the `LDA IoLoc` would take the same number of cycles as it did previously, thus leaving the timing loop unchanged.

These solutions seem pretty easy; therefore, you know there is a catch. The catch, unfortunately, is that most code is not isolated as in the example. Specifically, code commonly tries to load from or store to some location in memory other than the I/O location at the same time it is trying to access the I/O location.

Take, for example, the following fragment:

```
Loop         LDA          Data,y
             STA          IoLoc
             DEY
             BPL          Loop
```

In this example, we assume that the label `Data` refers to some kind of table which normally resides in the same bank as the program. Now if you set the data bank register to access I/O locations, then the reference to `Data` will also reference the same bank as the I/O; this solution is likely not acceptable. One thing you can do is move the data table to the direct page (zero page for 6502 programmers), but now the `LDA Data,y` instruction will take one less cycle to execute. There is a solution, although it is a little complicated. If we set the direct page register to a non page-aligned location, then we effectively apply a one-cycle penalty to all direct page references and solve our problem.

Of course, nothing is ever as simple as it seems. What happens to references to other direct page locations that expect to operate without the one-cycle penalty? To properly address this question, I would need much more space than I have here, so in lieu of further examples, I offer some general information. (As an aside, I used these techniques to transform the old "Apple II Disk II formatter module" for use in any bank of memory in the native IIGS environment. I accomplished this using, almost exclusively, editor find and replace commands, and I finished in hours instead of the days which would have been required to completely rewrite the program.)

In addition to the techniques already covered, there are a few other things which may be necessary to complete a transformation (they were necessary in the case of the formatter module).

As I already mentioned, one problem is what to do in the case where a program references I/O, local program-bank data, and the zero-page. In this case, significant rewrites could be required, but not necessarily.

In the case of the disk formatter, it turned out that some modules used both normal zero-page addressing and normal 16-bit absolute indexed addressing. Since the transformation process dictates that we change 16-bit absolute addressing to direct-page addressing with a non page-aligned direct page, there could have been a problem had both uses of the direct page been timing critical. Fortunately, by treating each module of the program separately, when I needed both types of addressing, only one was critical. The solution was to set the direct page to a non page-aligned value in some modules and to a page-aligned value in others. There are some minor logistical issues when a direct page's base address can be at either $xxx0 or $xxx1, the biggest of which is keeping track of which is in effect at a given point and knowing to reference the label as label, label+1, or label-1, depending upon the particular case.

With the formatter transformation, there was one other major issue: there are not direct-page versions of all the 16-bit absolute addressing modes (i.e., one cannot convert 16bitaddress,x to 8bitaddress,x). In the case of the formatter, I was able to solve this by reversing all the register use (i.e., all `LDY` instructions became `LDX` instructions, all `STY` instructions became `STX` instructions, etc.).

There are still a number of other ways in which one can approach these issues; one that comes to mind would be using some form of the new stack-relative addressing modes to yield yet another range of semi-independently accessible addresses.

The real point of this Technical Note is that with a little thought and effort, one can successfully convert a large subset of likely configurations for use in the native IIGS environment without major rewrites. The bottom line is to be creative!

Further Reference

- *Programming the 65816 Including the 6502, 65C02, and 65802* (Eyes/Lichty)
- *Apple IIGS Firmware Reference*

Apple IIGS
#3: Window Information Bar Use

Revised by: Dave Lyons — January 1991
Written by: Dan Oliver — October 1986

This Technical Note details the use of a window's information bar, including a code sample which places a menu in an information bar.
Changes since November 1988: Added a note about the current Resource Application when inside an `InfoDefProc` procedure, and information about information bars and `NewWindow2`.

Apple IIGS window information bars are not as straightforward as other window features, and one reason for this is the small amount of space originally allocated for their processing. If you feel your application can benefit from the use of information bars, you can implement them, and this Technical Note explains how to do it and includes some suggestions for their use. The code samples below demonstrate how to place a menu bar in an information bar, but your use of information bars is not limited to those described here.

Information Bar Initialization

You can create an information bar in a window when you create the window by setting the following fields in the parameter list you pass to `NewWindow`:

`wFrame`	Set bit 4.
`wInfoHeight`	Set to the height of the information bar (should not exceed window height).
`wInfoDefProc`	Set to the address of the information bar definition procedure (see below).

If you create a window as visible, the Window Manager will call your information bar definition procedure (`InfoDefProc`) before returning from `NewWindow`. If you have to create the contents of the information bar after the window, you will have a problem since the Window Manager will expect your `InfoDefProc` to draw things which do not yet exist. You can solve this problem by creating the window as invisible, creating the contents of the information bar, then showing the window. Another solution would be to detect, in the `InfoDefProc`, that the contents of the information bar do not yet exist.

NewWindow2, however, does not let you override the information bar drawing procedure in the template. If you pass a window template in a resource, creating the window as visible crashes (since the address of your information bar drawing procedure cannot possibly be in the window template resource). Instead, create the window as invisible and call SetInfoDraw to set the address of the information bar drawing procedure **before** calling ShowWindow.

Below is an example of initializing a window's information bar to contain a menu bar. The three key fields of the parameter list which you pass to NewWindow are as follows:

wFrame	Set bit 4 = 1 and bit 5 = 0 for an invisible window; the other bits do not affect the information bar, so you can set them as you wish.
wInfoHeight	Assuming you are using a system menu bar and initializing it before the window, set to the height FixMenuBar returned when you created the system menu bar. If you would rather use an absolute value, which we do not advise, you could use 14 which should be about right for the current system font.
wInfoDefProc	Set to the address of the InfoDefProc, in this case draw_info.

After you create the window, but before you show it, you can create the menu bar to place in the information bar. The code to create the menu bar might look like the following:

```
window             Direct page location that contains pointer to window's port.
;
; --- Create a menu bar -------------------------------------------------------------
------
;
             pha                         Space for result.
             pha
             pea    $FFFF                Set "use current port" flag.
             pea    $FFFF
             _NewMenuBar                 Create a menu bar.
             pla                         Get returned menu bar handle.
             sta    <menuBar             Remember menu bar handle.
             pla
             sta    <menuBar+2
;
;
; --- Store menu bar's handle in the window's InfoRefCon -----------------------------
------
;
             pei    <menuBar+2           Pass menu bar handle.
             pei    <menuBar
             pei    <window+2            Window to set refCon.
             pei    <window
             _SetInfoRefCon              Store menu bar handle in window's infoRefCon.
;
;
; --- Make the window's menu bar the current menu bar --------------------------------
------
;
             pei    <menuBar+2           Pass menu bar handle.
             pei    <menuBar
             _SetMenuBar                 Make new menu bar the current menu bar.

;
```

```
;
; --- Get the RECT of the window's information bar -----------------------------------
------
;
             pea    tempRect|-16        Pass pointer of RECT.
             pea    tempRect
             pei    <window+2           Pass pointer of window.
             pei    <window
             _GetRectInfo               tempRect = interior RECT of window's Info Bar.


; --- Dereference menu bar handle ----------------------------------------------------
------
;
             ldy    #2
             lda    [menuBar],y
             tay
             lda    [menuBar]
             sta    <menuBar            Now menuBar is the pointer to the Menu Bar.
             sty    <menuBar+2
;
;
; --- Set size of menu bar -----------------------------------------------------------
------
;
;
             lda    <tempRect+y1
             dec    a                   Overlap top side.
             ldy    #CtlRect+y1
             sta    [menuBar],y
;
             lda    <tempRect+x1
             dec    a                   Overlap left side.
             ldy    #CtlRect+x1
             sta    [menuBar],y
;
             lda    <rect+y2
             inc    a                   Overlap bottom side.
             ldy    #CtlRect+y2
             sta    [menuBar],y
;
;
; --- Set flag to tell Menu Manager to draw menu in current port ---------------------
------
;
             ldy    #CtlOwner+2         Set high bit in CtlOwner.
             lda    [menuBar],y
             ora    #$8000
             sta    [menuBar],y
;
;
; --- Create the menus and add them to the window's menu bar ------------------------
------
;
             lda    #4
loop         pha                        Save index into menu list.
             tay                        Switch index to Y.
;
             pha                        Space for return value.
             pha
             lda    menu_list+2,y       Pass address of menu/item lines.
             pha
             lda    menu_list,y
             pha
             _NewMenu
;                                       Menu handle already on stack.
```

```
             pea    0                   Insert menu list at front of list.
             _InsertMenu                Add my menus to the system menu bar.
;
             pla
             sec
             sbc    #4
             bpl    loop
;
;
; --- Initialize the size of the menu bar and menus ---------------------------------
------
;
             pha                        Space for returned bar height.
             _FixMenuBar                Fix up positions in the menu bar.
             pla                        Discard height of menu bar.
;
;
; --- Restore the system menu bar as the current menu -------------------------------
------
;
             pea    0                   Pass flag for system menu bar.
             pea    0
             _SetMenuBar                Make system menu bar current.
```

The window's menu bar is now initialized, and you can make the window visible with a call to `ShowWindow`; the `InfoDefProc` will draw the menu bar.

Information Bar Definition Procedure (InfoDefProc)

The `InfoDefProc` is slightly misleading; it is only responsible for drawing the interior, above the background, of the information bar. The `InfoDefProc` is not responsible for defining the information bar, drawing the frame and background, testing for hits, or tracking the user. The `InfoDefProc` is located inside your application, and the Window Manager calls it whenever it needs to draw the part of the window frame that contains the information bar. Each window with an information bar can have its own `InfoDefProc`, or they can all share a common `InfoDefProc`. When the Window Manager calls your `InfoDefProc`, it sets the proper port, the Window Manager's port, and the proper state, an origin local to the window frame and clipped to any windows above it. The direct page and data bank are not defined and should be considered unknown.

The Window Manager passes your `InfoDefProc` the following information:

- Pointer to the information bar's interior rectangle (less frame), local coordinates.
- Value of the window's `wInfoRefCon`, set and used only by your application.
- Pointer to the window's port (do **not** switch to this port for drawing).

Note: When the Window Manager calls your `InfoDefProc`, there is no guarantee that the current Resource Application is set to the value you expect. If your `InfoDefProc` makes Resource Manager calls, directly or indirectly, be sure to save, set, and restore the Resource Application using `GetCurResourceApp` and `SetCurResourceApp`.

A window that has an information bar containing a menu bar (handle stored in the window's `InfoRefCon`) might have a `InfoDefProc` as follows:

```
draw_info     START
;
theWindow     equ    6                    Offset to the information bar owner window.
infoRefCon    equ    theWindow+4          Offset to the window's information bar RefCon.
infoRect      equ    infoRefCon+4         Offset  to  the  information  bar's  enclosing
RECT.
;
              phd                         Save original direct page.
              tsc                         Switch to direct page in stack.
              tcd
;
;
; --- Draw the window's menu bar in the window's information bar ---------------------
------
;
              pei    infoRefCon+2         Pass handle of window's menu bar handle.
              pei    infoRefCon
              _SetMenuBar                 Make  the  window's  menu  bar  the  current  menu
bar.
;
              _DrawMenuBar                Draw the window's menu bar, as requested.
;
              lda    #0                   Zero is the flag for the system menu bar.
              pha
              pha
              _SetMenuBar                 Make the system menu bar current again.
```

```
;
;
; --- Remove input parameters from the stack ---------------------------------------
------
;            ldx    #12
             ply                        Pull original direct page off stack, save in
Y.
;
             tsc                        Move direct page point to stack.
             tcd
             lda    2,s                 Move  return  address  down  over  input
parameters.
             sta    2,x
             lda    0,s
             sta    0,x
;
             tsc                        Adjust stack for stripped input parameters.
             phx                        Number of bytes of input parameters.
             clc
             adc    1,s                 Add  number  of  input  parameters  to  stack
pointer.
             tcs                        And reset stack.
;
             tya                        Restore original direct page.
             tcd
;
             rtl                        Return to Window Manager.
             END
```

Information Bar Environment

An information bar is part of a window's frame, that is, not part of the window's content region. Because it is part of the frame, an information bar is in the Window Manager's port, so before an interaction (drawing or mouse selecting), the proper port (Window Manager's) must be in the proper state. The proper state means the origin must be at the window's upper-left corner and clipped to any windows above.

When the Window Manager calls the `InfoDefProc` it sets the proper port to the proper state; however, to interact with the information bar outside the `InfoDefProc`, you must set the proper port to the proper state. You can accomplish this with a call to `StartInfoDrawing`. When the interaction is completed, you must allow the Window Manager to return its port to a general state via a call to `EndInfoDrawing`. You are in a special state that requires some constraints (discussed later) between the calls to `StartInfoDrawing` and `EndInfoDrawing`.

Here is an example of interacting with our window's menu bar.

```
;
poll         pha                        Space for return value.
             pea    %0000111101101110   Pass event mask to use.
             pea    TaskRec|-16         Pass pointer to Task record.
             pea    TaskRec
             _TaskMaster
             pla                        Get returned value.
             beq    poll                Does event need further processing?
;
```

```
;
; --- Handle button down in window's information bar ---------------------------------
------
;
             cmp    #InInfo             In Information bar?
             bne    poll
;
             pha                        Space for result.
             pha
             lda    TaskRec+TaskData+2  Pass pointer of window.
             pha
             lda    TaskRec+TaskData
             pha
             _GetInfoRefCon             Get menu bar handle from window's InfoRefCon.
             pla
             sta    menuBar
             pla
             sta    menuBar+2
;
;
; --- Switch to proper port in proper coordinate system ------------------------------
------
;
             pea    tempRect|-16        Pass pointer to RECT to store info bar RECT.
             pea    tempRect
             lda    TaskRec+TaskData+2  Pass pointer of window.
             pha
             lda    TaskRec+TaskData
             pha
             _StartInfoDrawing
;
;
; --- Handle menu selection from window's menu bar -----------------------------------
------
;
             pea    TaskRec|-16         Pass pointer to Task record for MenuSelect.
             pea    TaskRec
             pei    menuBar+2           Pass handle of menu bar.
             pei    menuBar
             _MenuSelect                Let user make selection.
;
             lda    event+TaskData      Get the item's ID number.
             beq    exit                Was a selection made?
;
             _EndInfoDrawing            Switch back to original port.

;
;         (Handle the menu selection.)
;
;            The EndInfoDrawing followed by the StartInfoDrawing call is only
;            needed when code between them calls the Window Manager.
;
             pea    tempRect|-16        Pass pointer to RECT to store info bar RECT.
             pea    tempRect
             lda    TaskRec+TaskData+2  Pass pointer of window.
             pha
             lda    TaskRec+TaskData
             pha
             _StartInfoDrawing          Switch to the proper port in the proper state.
;
             pea    0                   Pass unhilite flag.
             lda    TaskRec+TaskData+2  Pass menu's ID number.
             pha
             _HiliteMenu                Unhilite menu's title.
;
```

```
;
; --- Clean up and return to polling -----------------------------------------------
------
;
exit          _EndInfoDrawing           Switch back to original port.
;
              pea    0                  Make system menu bar current.
              pea    0
              _SetMenuBar
;
              jmp    poll               Return to polling user.
;
```

Information Bar Shutdown

When the Window Manager closes the window, it is up to you to resolve any shutdown necessities associated with the information bar. Using our window menu bar example, the close window might look like the following:

```
;
              pei    menuBar+2          Pass handle of menu bar
              pei    menuBar
              _SetMenuBar
;
              pha                       Space for returned menu handle.
              pha
              pea    2                  ID number of second menu.
              _GetMHandle               Get the menu's handle.
              _DisposeMenu              Free menu record and associated data.
;
              pha                       Space for returned menu handle.
              pha
              pea    1                  ID number of first menu.
              _GetMHandle               Get the menu's handle.
              _DisposeMenu              Free menu record and associated data.
;
              pea    0                  Make system menu bar current.
              pea    0
              _SetMenuBar
;
              pha                       Space for menu bar's handle.
              pha
              pei    <window+2          Pass pointer of window to close.
              pei    <window
              _GetInfoRefCon            Get the InfoRefCon from the window.
              _DisposeHandle            Free menu bar record.
;
              pei    <window+2          Pass pointer of window to close.
              pei    <window
              _CloseWindow              Now the window can be closed.
;
```

The type of shutdown you use depends upon the contents of the information bar.

Why didn't I put a `DisposeMenuBar` call in the Menu Manager? I didn't think of it until a week too late. Sorry.

Other Information Bar Uses

The following suggestions are only theories and have not been tested.

- Display text information, as in Finder windows.
- Split window. Like the content region, the information bar could be large enough to hold data.
- Hold controls. You could scroll data in the content region while keeping the controls which affect the display in place and within the user's reach. (Note: The Control Manager does not know about information bars. If you want to draw and track objects in information bars, you have to do it yourself using QuickDraw II calls.)

Further Reference

- *Apple IIGS Toolbox Reference*, Volumes 1-3
- Apple IIGS Technical Note #83, Resource Manager Stuff

Apple IIGS
#4: Changing Graphics Modes in Mid-Application

Revised by: Dave "Dave" Lyons, C.K. Haun, & Dan Oliver January 1991
Written by: Dan Oliver October 1986

This Technical Note discusses how to switch between the two graphics modes, 320 and 640 horizontal resolution, while running an application which uses the Window, Control, and Menu Managers.
Changes since May 1990: Added information about reinstalling fonts after restarting QuickDraw II.

Why Change Resolution?

Why not? There are certain applications where the ability to run in both modes is essential; most graphics applications fall into this category. Other applications might switch modes to provide features which their competitors lack; a financial application might display figures in 640 mode and charts in 320 mode. Still other applications may want to give the user the choice. A word processor might seem useful only in 640 mode, but what if the user wants to print greeting cards with pictures? The user does not need the line length provided in 640 mode but does need the added color of 320 mode for the pictures.

Let me preach a little. I have worked on other machines with different graphic modes and learned some things that might be of use to application programmers. Many application programmers fight mode switching with either rhetoric or apathy, then when users expect their software to run in either mode, they become frustrated when it does not allow switching. To avoid the problem of frustrating the user, you can provide mode switching (which is not as hard as you might think).

How To Change Modes

First, assume you are in an application which is running with a system menu bar, a few visible windows with scroll bars, and one window with some standard controls. At some point, the user decides to change modes, possibly via a menu item thoughtfully provided by the application programmer. Your change mode handler might look like the following:

```
;
; --- This step is necessary if QuickDraw Auxiliary is started -----------------------
------
               _QDAuxShutDown               ;Shut down QDAux first
; ------------------------------------------------------------------------------------
------
               _QDShutdown                  ;Shut down QuickDraw.
                                            ;This will turn graphics off so you will
see
                                            ;the text screen for a second (a
advertisement
                                            ;might go here).
               lda    <mode                 ;Variable that holds current resolution.
               eor    #$0080                ;Flip the mode bit, $0000 = 320, $0080 =
640.
               sta    <mode                 ;New value will be used to start the new
mode.
;
               pei    <QDzpage              ;Pass the direct pages allocated for
QuickDraw.
               pei    <mode                 ;New mode.
               pei    <QDwidth              ;0 for screen width; other numbers for
printing
               pei    <MyID                 ;Pass my ID number.
               _QDStartup                   ;Restart QuickDraw in the new mode.
;
               _GrafOff                     ;Turn screen off because changing mode
                                            ;may not be pretty.
; --- This step is necessary if you need QuickDraw Auxiliary  -----------------------
------
               _QDAuxStartUp                ;Start QDAux again
; ------------------------------------------------------------------------------------
------
;
;
; --- Fix up the cursor for the new mode ---------------------------------------------
------
;
               pea    0                     ;Pass minimum cursor X position.
               lda    #319                  ;Maximum X position for 320 mode.
               ldx    <mode                 ;320 or 640 mode?
               beq    store
               lda    #639                  ;Maximum X position for 640 mode.
store          pha                          ;Pass maximum cursor X position.
               pea    0                     ;Pass minimum Y cursor position.
               pea    199                   ;Pass maximum Y cursor position.
               _ClampMouse                  ;Clamp the cursor to the new screen size.
;
               _HomeMouse                   ;Move the cursor to 0,0 to make sure
                                            ;it is on screen.
               _ShowCursor                  ;Make cursor visible.
;
;
; --- Tell tools about the change ----------------------------------------------------
------
;
               _WindNewRes                  ;Tell Window Manager about the change.
               _MenuNewRes                  ;Tell Menu Manager about the change.
               _CtlNewRes                   ;Tell Control Manager about the change.
;
;
; --- Fix the screen to look good ----------------------------------------------------
------
;
;               Here you might want to change the color of the desktop, windows, menus
or
```

```
;               controls to look good for the new mode.
;
;               See example below.
;
; --- Redraw the screen in the new mode -------------------------------------------
------
;
              pea    0                     ;Pass flag to draw entire screen.
              pea    0
              _RefreshDesktop              ;Draw entire screen.
;
              _GrafOn                      ;Now show the new screen.
;
```

That is not too bad, but I left out the fun part. Before the `RefreshDesktop` there is a section named "Fix up the screen to look good." This section is where you might want to put some color into windows, controls, and menus if you are switching to 320 mode; changing colors is not required, but there are some things which are.

When switching from 640 mode to 320 mode, some windows (both visible and invisible) might be positioned off the screen in 320 mode. The first way to handle this problem is easy for you, the programmer, but not so great for the user: close all the windows before changing modes, then position them correctly when the user opens them in the new mode. The second way to handle the problem is to walk the window list and move all the windows, maybe even change their sizes. You could double each window's horizontal starting position and width when switching from 320 mode to 640 mode and halve it when changing from 640 mode to 320 mode. The vertical position and height are okay. An example of the second method is given below.

Windows with vertical scroll bars in the window frame are the same width when you change modes, so switching from 320 mode to 640 mode results in a narrower bar while changing from 640 mode to 320 mode produces a wider bar. The bars change to the correct size as soon as the user resizes the window, since `SizeWindow` deletes the old scroll bars and allocates new ones according to the current mode. If, as suggested above, you resize all the windows after the mode change and before calling `RefreshDesktop`, you should be in good shape. If you choose not the follow this recommendation, you should call `SizeWindow` for every window with scroll bars and change the size of each window at least one pixel since `SizeWindow` does not do anything if the passed size is not different than the current size.

You should dispose of scroll bars in a window's content region and recreate them; this is not nice, but very few applications have scroll bars in a window's content region.

You should not resize any open new desk accessory (NDA) windows. NDAs may be dependent on screen mode, or their current position, or other such things which may change with resolution. To be kind to the NDAs, you should issue a `CloseAllNDAs` call. This call allows the NDAs to go through their normal close procedures. If a user wants an NDA open in the new screen resolution he must reopen it. This assures that the NDA always knows its own position and the current screen resolution.

`WindNewRes` resets the desktop shape and pattern and the Window Manager's icon font to their defaults for the new mode, so if you changed any of these, you must add to or subtract from the desktop again and reinitialize to your custom pattern or icon font again.

`CtlNewRes` resets the Control Manager's icon font to the default for the new mode, so if you changed the Control Manager's icon font, you must reinitialize to your icon font again.

Reinstalling Large Fonts

After restarting QuickDraw II, you should call `InstallFont` again on the fonts your application is using. This causes the Font Manager to call `InflateTextBuffer` so that QuickDraw can draw text correctly in large font sizes.

Repositioning and Resizing Windows in the New Mode

Here is an example of how to reposition and resize windows in the new mode.

```
;               QuickDraw and the tools have already been reinitialized in the new
mode.
;               mode = $0000 if in 320 mode, $0080 if in 640 mode.
;
BoundsRect      equ    8                    ;Offsets in port record from QuickDraw
document.
PortRect        equ    16
;
                _CloseAllNDAs               ; close all open NDA windows
                pha                         ;Space for result.
                pha
                _FrontWindow                ;Start with the top most window, this
assumes
                bra    enter                ;there are no invisible windows ahead of
the
                                            ;active window in the window list.
                ldy    #BoundsRect+2
                lda    [window],y           ;Get window's starting horizontal position.
                eor    #$FFFF               ;Convert to screen coordinate (negate it).
                inc    a
                asl    a                    ;Double it if we're going to 640 mode.
                ldx    <mode                ;Going to 320 or 640 mode?
                bne    store1               ;Ready if we're going to 640.
                lsr    a                    ;Otherwise, undo the doubling,
                lsr    a                    ;and halve the starting horizontal
position.

store1          pha                         ;Pass window's new X starting position.
                ldy    #BoundsRect
                lda    [window],y           ;Get window's starting vertical position.
                eor    #$FFFF               ;Convert to screen coordinate.
                inc    a
                pha                         ;Pass window's current Y starting position.
                pei    <window+2            ;Pass window to move.
                pei    <window
                _MoveWindow                 ;Move the window to its new position.
;
                ldy    #PortRect+6          ;Get window's current width.
                lda    [window],y           ;(This assumes the window's origin is 0,0.)
                asl    a                    ;Double the window's width if going to 640
mode.
                ldx    <mode                ;Going to 320 or 640 mode?
                bne    store2               ;Ready if we're going to 640.
                lsr    a                    ;Otherwise, undo the doubling,
                lsr    a                    ;and halve the window's width.
store2          pha                         ;Pass window's new width.
                ldy    #PortRect+4
                lda    [window],y           ;Get window's height.
                pha                         ;Pass window's current height.
                pei    <window+2            ;Pass window to resize.
                pei    <window
                _SizeWindow                 ;Resize the window.
;
                pha                         ;Space for result.
                pha
                pei    <window+2            ;Pass pointer to window we just processed.
                pei    <window
                _GetNextWindow              ;Get the pointer to the next window.
;
enter           pla                         ;Remember the pointer to this window.
                sta    <window
                pla
                sta    <window+2
;
                ora    <window              ;Are there any more windows?
                bne    loop
```

```
;
```

WindNewRes

Generally, `WindNewRes` does the following:

- closes its port
- opens its port again, now in the new mode
- reinitializes the desktop size
- chooses the proper icon font for close and zoom boxes
- reinitializes the desktop pattern
- changes the SCB byte of each window's port to the new mode
- recomputes the `VisRgn` for each window

MenuNewRes

Generally, `MenuNewRes` does the following:

- closes its port
- opens its port again, now in the new mode
- reinitializes internal parameters, like vertical line width, for the new mode
- reinitializes the color palette via `InitPalette`
- subtracts the system menu bar from the desktop (this is why you must call `WindNewRes` first)
- draws the system menu bar

CtlNewRes

Generally, `CtlNewRes` does the following:

- chooses the proper icon font for radio button, check box, grow box and scroll bar arrows
- reinitializes internal parameters, like vertical line width, for the new mode

Further Reference

- *Apple IIGS Toolbox Reference*

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#5: Window and Menu Titles

Revised by: Matt Deatherage November 1990
Written by: Dan Oliver October 1986

This Technical Note discusses spacing for both window and menu titles.
Changes since November 1988: Revised to include new information on the default placement of the Apple menu.

Strings used for window titles should always have a space as the first and last characters. This spacing is especially important for windows that use a lined window title bar since, without the beginning and ending space, the line pattern in the title bar runs against the title. Since there will be window editor desk accessories which allow the user to change the title bar pattern without the application knowing, you should pad your window titles with spaces even if you are using black window title bars.

The Window Manager does not force spaces on either side of titles to optimize the window frame drawing speed; it is much faster to let the text punch a hole in the title bar pattern than to compute the rectangle, fill it, and draw the text.

To provide the user with a consistent visual interface, you should also pad your menu titles with spaces. If you use either one or two spaces (the Apple IIGS Finder has used two) before and after each menu title, your menu titles will be consistent and balanced (two spaces work well in 640 mode where one space usually suffices for 320 mode). Although it is true that a menu bar will look about the same if the first menu title has two spaces before it and no space following it and all the other menu titles have four spaces before them, when the user pulls down the menu, the Menu Manager's highlighting will clearly (and embarrassingly) show the spaces in the menu titles.

If you would like to place the Apple menu differently, you must use Menu Manager calls since you cannot place spaces around the at sign (@) which the Menu Manager uses to represent the Apple logo in a menu title. The easiest way to accomplish this is calling `SetMTitleStart` to set the starting position for the leftmost title (usually the Apple menu) within the current menu bar. The Apple IIGS Finder has used a value of 10 ($0A) pixels.

Beginning with System Software 5.0, the Apple menu is placed at a default of 10 pixels from the left edge of the menu bar in 640 mode or five pixels in 320 mode. If you use `SetMTitleStart` to change the default, the value is still interpreted as an absolute placement from the left edge of the menu bar. For example, `SetMTitleStart(6)` moves the Apple menu one pixel to the right of the default in 320 mode and four pixels to the left of the default in 640 mode. Be sure not to use `SetMTitleStart` to set the Apple menu starting place to the left of the default, as doing so interferes with the AppleShare activity arrows.

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#6: QuickDraw II Pattern Data Structure

Revised by: Dave Lyons July 1989
Written by: Guillermo Ortiz December 1986

Some QuickDraw II calls require a pen pattern as input or return one as output; regardless of the drawing mode (320 mode or 640 mode), a pen pattern takes 32 bytes.
Changed since November 1988: Starting with System Software 5.0, all 32 bytes are significant if bit 15 of the current port's `arcRot` field is set. Changed wording to cover QuickDraw II patterns in general, instead of pen patterns only.

Early QuickDraw II documentation described the pattern data structure as follows:

```
TYPE
      nibble  = 0..15;
      twobit   = 0..3;
      Pattern = RECORD CASE MODE OF
                   mode320:(PACKED ARRAY [0..63] OF nibble);      { 32 bytes }
                   mode640:(PACKED ARRAY [0..63] OF twobit);      { 16 bytes }
                END;
```

This declaration could lead one to believe that 16 bytes are enough when making calls to QuickDraw II in 640 mode. This is not true. A pattern **always** takes 32 bytes; QuickDraw II calls that copy or construct patterns access all 32 bytes. That means it is never safe to pass the address of a 16-byte area as a pattern. Toolbox calls that return data into your buffer overwrite 16 bytes immediately following your buffer. Calls that copy data from your buffer access those extra 16 bytes, possibly including soft switches or reserved space in the memory map.

The difference between modes is that QuickDraw II normally ignores the second 16 bytes if the current port's `locInfo` indicates 640 mode. Starting with System Software 5.0, all 32 bytes of patterns are significant in 640 mode when bit 15 of the current port's `arcRot` field has been set with `SetArcRot`. In this case, patterns are 16 pixels wide and 8 pixels high.

Further Reference

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

Apple IIGS
#7: Halt Mechanism in IIGS SANE

Revised by: Guillermo Ortiz & Matt Deatherage November 1988
Written by: Guillermo Ortiz December 1986

This Technical Note formerly described a bug of SANE on the Apple IIGS which caused it to jump through location $00/0018 instead of through the `HALT` vector in the SANE direct page.

The bug which caused SANE on the Apple IIGS to jump through location $00/0018 instead of through the `HALT` vector in the SANE direct page was fixed in the Apple IIGS ROM 2.0. 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 IIGS
#8: Elems Functions in IIGS SANE

Revised by: Matt Deatherage November 1988
Written by: Guillermo Ortiz December 1986

This Technical Note discusses a problem which existed with the `Elems` functions in the IIGS SANE Tool Set 1.0. Current IIGS System Disks contain a patch which corrects this problem.

Calls to any of the `Elems` functions in version 1.0 of the IIGS SANE Tool Set may return an invalid result unless you are evaluating data which resides in bank $00 due to a problem with the `Elems` parameter passing mechanism. These results are random because when SANE checks the validity of its input, it uses values that have no relations to the actual ones, and once it completes the validation, it uses the real operands.

All System Disks released on or after December 1, 1986 include a RAM patch which fixes the `Elems` parameter passing mechanism; therefore, you should not have to write a special case to handle this problem if you are shipping your application with the most recent Apple IIGS System Disk. You should contact Apple Software Licensing at Apple Computer, Inc.; 20525 Mariani Avenue, M/S 38-I; Cupertino, CA 95014 or (408) 974-4667 to obtain the most recent version of the Apple IIGS System Disk.

Further Reference

- *Apple Numerics Manual*

Apple IIGS
#9: IIGS Sound Expansion Connector: Analog Input/Output Impedances

Revised by: Jim Merritt & Jim Mensch November 1988
Written by: Jim Merritt December 1986

This Technical Note discusses the impedances of the analog signal pins on the IIGS sound expansion connector since an interface to this connector must take the impedance of the pins into account to function properly.

The analog output impedance of pin 3 depends upon the characteristics of the 5503 sound synthesis chip in any particular IIGS machine. Across systems, this impedance may range from 4.5 KΩ to 9 KΩ.

Pin 1, the A/D input, presents a dynamic load to the source, drawing at 10 KΩ for approximately 500 ns during every sample period. It is reasonable, however, to treat the input pin as if it presents a continuous load of 10 KΩ without compromising the interface or the fidelity of the input sample.

Consult the *Apple IIGS Hardware Reference* for further technical information about the Ensoniq 5503 sound synthesis chip used in the IIGS.

Further Reference

- *Apple IIGS Hardware Reference*

Apple IIGS
#10: InvalRgn Twist

Revised by: Steven Glass November 1988
Written by: Guillermo Ortiz April 1987

`InvalRgn(RgnHandle)` accumulates the region to which `RgnHandle` points into the update region of the current window's port; in the process, it makes the region global, thus causing problems if later calls expect the region to still be local.

The region you pass to `InvalRgn` is local to the window to which it is related; however, `InvalRgn` returns the region in global coordinates. To preserve the original region for your use after the call to `InvalRgn`, you should duplicate it and use the copy to make the call then dispose of the copy when `InvalRgn` returns. The following example demonstrates the process:

```
void MyInvalReg(RegHandle)

handle RegHandle;
{
handle AuxHandle;

AuxHandle = NewRgn();              /* create room */
CopyRgn(RegHandle,AuxHandle);      /* make a copy  */
InvalRgn(AuxHandle);               /* do it with the copy */
DisposeRgn(AuxHandle);             /* now get rid of it! */
}
```

Further Reference

- *Apple IIGS Toolbox Reference*, Volume 2