Apple][Computer Family Technical Documentation

Technical Notes

Apple Computer -- Developer CD Volume 2 -- September 1989

##
FILE: !TN.000.About.89.09
##

Apple II
Technical Notes

Developer Technical Support

#0: About Apple II Technical Notes September 1989

Technical Note #0 (this document) accompanies each release of Apple II Technical Notes. This release includes revisions to Apple IIGS Notes #26, #35, #36, #45, and #64, AppleTalk #3-#4, Apple II Miscellaneous #10, and SmartPort #2, new Notes for the Apple IIGS (#65-#70), and an index to all released Apple II Technical Notes. If there are any subjects which you would like to see treated in a Technical Note (or if you have any questions about existing Technical Notes), please contact us at one of the following addresses:

Apple II Technical Notes
Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, M/S 75-3T
Cupertino, CA 95014
AppleLink: AIIDTS
MCI Mail: AIIDTS (264-0103)

We want Technical Notes to be distributed as widely as possible, so they are sent to all Partners and Associates at no charge; they are also posted on AppleLink in the Developer Services bulletin board and other electronic sources, including the Apple FTP site (IP 130.43.2.2). You can also order them through APDA. As an APDA customer, you have access to the tools and documentation necessary to develop Apple-compatible products. For more information about APDA, contact:

APDA
Apple Computer, Inc.
20525 Mariani Avenue, M/S 33-G
Cupertino, CA 95014
(800) 282-APDA or (800) 282-2732
Fax: (408) 562-3971
Telex: 171-576
AppleLink: APDA

We place no restrictions on copying Technical Notes, with the exception that you cannot resell them, so read, enjoy, and share. We hope Apple II Technical Notes will provide you with lots of valuable information while you are developing Apple II hardware and software. The following pages list all Apple II Technical Notes that have been released.

Released Apple II Technical Notes September 1989

New ***

Revised *R*

Apple IIc

Apple IIe

Apple IIGS

Apple II Miscellaneous

AppleTalk

END OF FILE !TN.000.About.89.09

```
#######################################################################
### FILE: !TN.Index.89.09
#######################################################################
```

Apple II
Technical Notes

Developer Technical Support

Index

September 1989

END OF FILE !TN.Index.89.09

###
FILE: TN.AIIc.001
###

Apple II
Technical Notes

Developer Technical Support

Apple IIc
#1: Mouse Differences on IIe and IIc

Revised by: Matt Deatherage November 1988
Revised by: Cameron Birse February 1986

This Technical Note explains differences between the IIe and IIc when working with a mouse and how to write programs which function properly on both machines.

If you use the mouse firmware routines (i.e., SetMouse) to control the mouse, then these routines will perform the same function on the IIc as they do on the IIe. However, a program which uses the mouse may not behave the same on both computers, and there are two reasons for the possible differences.

If a program does not properly set the environment prior to calling the mouse firmware routines, it is possible for a program to work on one machine and not the other. In addition, there are differences in machines and although the ROM routines perform the same functions, there may be a noticeable difference in the mouse behavior between the two machines.

This Note explains the fundamental differences between the way the mouse works on the two machines. We point out precautions that you need to take to ensure your assembly language programs work properly on both machines. (With the exception of mouse movement scaling described below, neither BASIC nor Pascal programs need be concerned with setting the proper environment.)

The Apple IIe mouse card has a microprocessor on it which constantly polls the mouse to get status and position information. This data is kept on the card and is available whenever the program requests it through the ReadMouse routine. If the mouse is in passive mode, this information will be picked up by the main program whenever it gets around to it.

The SetMouse routine can set the mouse card to issue interrupts under certain conditions. When the mouse card determines that such conditions exist, it issues an interrupt. This interrupt stops the main computer and goes to whatever interrupt handling routine has been prepared. This routine then reads the information from where the card processor saved it and puts it in the screen holes. When using a mouse on an Apple with a mouse card, your program is only interrupted if you have requested it, and the data in the screen holes is changed only when the program's interrupt handler or polling routine calls ReadMouse. In addition, enabling and disabling interrupts does not affect the card's microprocessor from updating the mouse information.

The Apple IIc mouse does not have a card microprocessor, so mouse information is collected by interrupting the microprocessor of the IIc itself. When the interrupt occurs, the firmware captures it and processes it, which includes updating the screen holes. The interrupt is passed only if SetMouse set up the conditions to do so.

Having the mouse interrupt the computer's microprocessor also means that your program is being constantly interrupted, which affects program timing. This interruption also means that the screen holes are constantly updated with X and Y information, even in passive mode, since this information must be stored somewhere and there is no card to keep it in. If you have disabled interrupts, the mouse can never interrupt the microprocessor, so the X and Y values are never updated and calling ReadMouse will indicate that there has been no mouse movement.

Since the Apple IIc is constantly interrupted while the mouse is on, the program's performance may be affected. To minimize this effect, the IIc responds one-half as frequently to mouse movements as does the mouse card, which means the mouse must be moved twice as far to create the same on-screen effect. If you want the same behavior on both machines, multiply the IIc X and Y values by two and the clamping value by one half. You do not need to make any changes to these values if your program is running on a IIe.

With this exception for mouse movement, your assembly language program can ignore which machine it is running on by following the precautions listed in Mouse Technical Note #1, Interrupt Environment with the Mouse (you must take these conditions into account if you want your assembly language program to behave similarly on both machines). If you are working in BASIC or Pascal, these conditions are already handled for you.

Further Reference
- Apple IIc Technical Reference Manual
- Mouse Technical Note #1, Interrupt Environment with the Mouse

END OF FILE TN.AIIc.001

##
FILE: TN.AIIc.002
##

Apple II
Technical Notes

Developer Technical Support

Apple IIc
#2: 40-Column and Double High-Resolution Graphics

Revised by: Matt Deatherage November 1988
Revised by: Cameron Birse February 1986

This Technical Note describes how to properly handle the 40-column screen while using double high-resolution graphics on the Apple IIc.

Many developers using double high-resolution graphics may wish to use 40-column text displays so that the text can be read on a television set. There are a couple of possibilities for accomplishing this task:

1. You can define your own double high-resolution character set with any size characters you desire, then plot them on the double high-resolution screen.
2. You can print text to the Apple IIc text screen and toggle the screen on to display it.

Note: There is no way to display 4 lines of 40-column text at the bottom of the double high-resolution screen in mixed mode since the 80 column hardware must be active while double high-resolution mode is being used.

Using the second method outlined above requires some special considerations.

The Apple IIc scroll routine continues to use the window parameters when scrolling, but uses the 80COL softswitch to determine if it should scroll the 80-column screen or 40-column screen. Since the firmware has initialized a 40-column window, the scroll routines will move only the first 40 columns, but the 80COL flag has been turned on for double high-resolution. Because of the 80COL flag, the scroll routine takes every even column from auxiliary memory and every odd column from main memory. As a result, only the first 40 columns get scrolled, 20 columns from auxiliary memory and 20 columns from main memory.

One solution to the problem is writing your own scroll routines, while another is writing to the screen so scrolling is not necessary. There is, however, another solution. Turn on the full 80-column mode with PR#3 or equivalent. Now print your text to COUT in the normal manner, and do not exceed 40 characters per line--the 80-column firmware should scroll everything properly. When you are ready to display text, send a Control-Q sequence through COUT to toggle to 40-columns and send a Control-R sequence to return to double high-resolution mode. These control characters toggle the display modes, but leave the 80-column firmware active.

When switching between modes, you may experience a momentary glitch. If you send the Control-Q sequence to COUT while still in graphics mode, the screen will first switch to the normal high-resolution mode before finally switching to text mode. If you switch to text mode first, the text will be in 80-column mode (with 40 columns displayed on the left of the screen) before ultimately switching to 40-column mode). This same potential glitch may occur when switching back to double high-resolution mode, and it may be only momentary and not present any problems for your application. If, however, it does present a problem, you may wish to make your switch coincide with the video's vertical blanking interval (see the Apple IIc Technical Reference Manual).

Further Reference
o   Apple IIc Technical Reference Manual

END OF FILE TN.AIIc.002

##
FILE: TN.AIIc.003
##

Apple II
Technical Notes

Developer Technical Support

Apple IIc
#3: Foreign Language Keyboard Layouts

Revised by: Matt Deatherage November 1988
Revised by: Cindy Roberts January 1985

This Technical Note formerly described the keyboard layouts and ASCII codes for international versions of the Apple IIc keyboard.

The information about international keyboard layouts and key codes which this Note formerly covered is now documented in all current versions of the Apple IIc Technical Reference Manual.

Further Reference
o Apple IIc Technical Reference Manual

END OF FILE TN.AIIc.003

```
#####################################################################
### FILE: TN.AIIc.004
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIc
#4:   Dvorak Keyboard Layout

Revised by: Matt Deatherage   November 1988
Revised by: Cameron Birse   February 1986

This Technical Note discusses the Dvorak keyboard layout on the Apple IIc.

The old, red version of the Apple IIc Reference Manual incorrectly illustrated the Dvorak keyboard layout, however, the current Apple IIc Technical Reference Manual contains a corrected diagram on page 370.

The diagram in the current manual shows the Dvorak Simplified Keyboard (DSK) as it appears and functions on the Apple IIc today. This layout is the ANSI standard for the Dvorak keyboard layout, which was not available when the original IIc keyboard ROM was created. Previous IIc computers had a DSK layout as follows:

```
|    | !  | @  | #  | $  | %  | ^  | &  | *  | (  | )  | {  | }  |      |
|esc | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 0  | [  | ]  |delete|

|      | ?  | <  | >  | P  | Y  | F  | G  | C  | R  | L  | :  | +  | |  |
|tab   | /  | ,  | .  | p  | y  | f  | g  | c  | r  | l  | ;  | =  | \  |

|       | A  | O  | E  | U  | I  | D  | H  | T  | N  | S  | _  |       |
|control| a  | o  | e  | u  | i  | d  | h  | t  | n  | s  | -  |return |

|          | "  | Q  | J  | K  | X  | B  | N  | W  | V  | Z  |           |
|shift     | '  | q  | j  | k  | x  | b  | n  | w  | v  | z  |shift      |

|caps| ~  |  |    |                    |    |    |    |||  |/\  |
|lock| `  |  | OA |     space bar      | CA |<== |==> |\/  |||  |
```

Figure 1-Dvorak DSK Layout on Early IIc Computers

Due to service part changes and other manufacturing considerations, it is not possible to identify which IIc units have which keyboard ROM by looking at identification bytes. If a program requires knowledge of this information (i.e., a typing program which draws the Dvorak keyboard), it must ask the user for input.

One possible way to accomplish this would be for a program to draw a blank keyboard layout (except for shift, tab, control, and other keys which do not move between Dvorak and Sholes layouts) and ask the user to press the key to the right of the left shift key, while the drawing on screen highlights the correct key to press. If the key is a Z, the layout is a standard Sholes layout. If the key is an apostrophe or quotation mark, the layout is the DSK layout shown above. If the key is a semicolon or colon, the layout is the ANSI DSK layout on new IIc models. Since such a program must already ask the user if the keyboard switch is depressed (indicating a Dvorak layout), making this type of inquiry instead will do the trick.

The IIc manual has another DSK diagram in the front, on page 7. This diagram correctly shades those symbols which are in different places in the two DSK layouts.

Further Reference
o   Apple IIc Technical Reference Manual

END OF FILE TN.AIIc.004

##
FILE: TN.AIIc.005
##

Apple II
Technical Notes

Developer Technical Support

Apple IIc
#5: Memory Expansion on the Apple IIc

Revised by: Matt Deatherage November 1988
Written by: Cameron Birse October 1986

This Technical Note describes some important differences in the "memory-expandable" Apple IIc which you should take into account to ensure compatibility.

Beginning with the third Apple IIc, which was announced in September 1986, all new IIc models differ significantly from their predecessors. The most notable of these differences is the addition of a memory expansion capability. The memory expansion card for the IIc is functionally identical to the card for the IIe, but the IIc card "lives" in slot 4 and the firmware is included in the ROM on the IIc motherboard. This architecture means that you cannot depend upon the firmware ID bytes to tell if a card is installed, since unlike other "peripheral cards" in the IIc, the memory expansion card is not necessarily present. For this particular case, you need to interrogate the card and see how many blocks of memory are available. If there are no available blocks, there is no card.

SmartPort

Do a STATUS call with a statcode = $03 to get the Device Information Block (DIB). This call returns a value of $000000 in the device size fields if there is no RAM card.

In version 3 of the IIc ROM, the value resulting from a status call to device 0 implies that there is always a real card connected; the ROM version 4 returns device connected only when there is RAM card present.

ProDOS

When you do an ON_LINE call to the ProDOS MLI and there is no RAM on the Memory Expansion Card, you get an error $2D. This error is not a ProDOS error, rather it is a SmartPort error. The error is BADBLOCK, and basically tells you that the block requested was not available. If you try to catalog the RAM disk from BASIC, you will get a PATH NOT FOUND error.

Pascal

Formatting the RAM disk (unit #9) with no memory on the card returns no error.

Doing a UnitStatus call will return zero blocks available, and trying to read the volume directory will result in an IORESULT of 8, which means no room is available on the volume. Doing the V(ols command from the F(iler will result in a <no dir> and # of blocks = 0.

DOS 3.3

If there is no memory on the card and you initialize it with an IN#4 (which returns a slash, appearing to have successfully initialized the RAM disk), you will get an I/O error (ONERR code = 8) if you try to read from or write to the RAM disk.

Important: Another significant ramification of the memory expansion capability is that the mouse firmware has been moved to slot 7. This change means that programs should scan the slots just as they would on a IIe to find what peripherals are installed. Since most programs have a scan routine in them for the IIe, it should be a relatively minor change to call this routine for whatever machine you are on. In fact, we strongly recommend that programs always scan the slots for peripheral devices regardless of the machine on which they are running.

The firmware ID bytes for this version of the machine are:

```
Original Expandable IIc
    $FBB3--$06         $FBC0--$00         $FBBF--$03

Revised Expandable IIc
    $FBB3--$06         $FBC0--$00         $FBBF--$04

Apple IIc Plus
    $FBB3--$06         $FBC0--$00         $FBBF--$05
```

Further Reference

- o Apple IIc Technical Note #6, Buffering Blues
- o Apple IIc Technical Note #7, Existing Versions
- o Apple II Miscellaneous Technical Note #2, Apple II Family Identification Routines 2.1
- o Apple II Miscellaneous Technical Note #7, Apple II Family Identification
- o Apple II Miscellaneous Technical Note #8, Pascal 1.1 Firmware Protocol ID Bytes

END OF FILE TN.AIIc.005

```
#####################################################################
### FILE: TN.AIIc.006
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIc
#6: Buffering Blues

Revised by: Mike Askins — November 1988
Written by: Guillermo Ortiz — January 1987

This Technical Note describes changes on the memory-expandable IIc which affect the procedures for enabling keyboard and serial input buffering.

When the IIc firmware was reorganized to accommodate the memory expansion card in slot 4, the mouse moved to slot 7, thus causing some screen holes to be reassigned. This change may software which uses keyboard or serial input buffering to crash.

The following list shows the changes in the locations which are used for enabling keyboard and serial input buffering:

Name	Original & 3.5 ROM	Expandable IIc	Comment
typhed	$5FA	$5FA	;buffer the keyboard? NO CHANGE
twkey	$5FF	$5FC	;storage pointer for type-ahead buffer
trkey	$6FF	$6FC *	;retrieve pointer for type-ahead buffer
aciabuf	$4FF	$4FC	;Owner of serial buffer, if any
twser	$57F	$57C	;storage pointer for serial buffer
trser	$67F	$67C	;retrieve pointer for serial buffer

* In the version 3 ROM (original "memory-expandable" IIc) this pointer is still $6FF which causes, among other things, the Terminal Mode to be inoperative. Revision 4 of the IIc firmware fixes this bug.

We can not emphasize enough the need for carefully checking the version of the machine on which a program is running. It is also important to pay attention to the now obvious fact that even in the Apple IIc things can (and most probably will) move around, making any hard-coded slot assignment a sure source of incompatibility. To ensure compatibility, scan the slots.

The Apple IIc Technical Reference Manual describes how to enable buffering. Using serial buffering as an example, the pertinent instructions in the manual should now be understood as meaning:

Using Serial Buffering Transparently

```
 If (machineID = "Memory Expandable IIc") then
   begin
     aciabuf = $04FC;     {Newest IIc with Expanded Memory Capabilities,}
```

```
     twser = $057C;       {ROM versions 3 and 4.}
     trser = $067C
   end
 else
    begin
      aciabuf = $04FF;    {Original IIc and 3.5 ROM IIc}
      twser = $057F;
      trser = $067F
    end;

 Set_Location aciabuf to $C1 or $C2.
 Set_Locations twser and trser to $0.
```

Using Serial Interrupts Through Firmware

 Set_Location aciabuf to a value other than $C1 or $C2

Further Reference

- o   Apple IIc Technical Reference Manual
- o   Apple IIc Technical Note #7, Existing Versions
- o   Apple II Miscellaneous Technical Note #7, Apple II Family Identification

END OF FILE TN.AIIc.006

###
FILE: TN.AIIc.007
###

Apple II
Technical Notes

Developer Technical Support

Apple IIc
#7: Existing Versions

Revised by: Matt Deatherage November 1988
Written by: Guillermo Ortiz November 1987

This Technical Note describes the main differences between the five different IIc ROM versions which encompass the original IIc and four revisions.

Original IIc ($FBBF = $FF)

- o Can use the IIc external drive only
- o No AppleTalk firmware
- o PR#7 boots the second drive
- o Mouse firmware maps to slot 4
- o Serial firmware does not mask incoming linefeed characters
- o Serial firmware does not support XON/XOFF protocol

3.5 ROM IIc ($FBBF = $00)

- o Can use the IIc external drive and the UniDisk 3.5 drive
- o AppleTalk firmware maps to slot 7
- o PR#7 returns the message "AppleTalk Off Line"
- o Mouse firmware maps to slot 4
- o Serial firmware defaults to mask all incoming linefeed characters
- o Serial firmware supports XON/XOFF protocol

Original "Memory-Expandable" IIc ($FBBF = $03)

- o Can use the IIc external drive, the UniDisk 3.5 drive, and the IIc Memory Expansion Card
- o Mouse firmware maps to slot 7
- o No AppleTalk firmware
- o PR#7 kills the system
- o Serial firmware defaults to mask all incoming linefeed characters
- o Serial firmware supports XON/XOFF protocol

Revised "Memory-Expandable" IIc ($FBBF = $04)

Same as Original Memory-Expandable, plus:
- o Keyboard buffering firmware bug fixed
- o Firmware returns correct information when the Memory Expansion Card is not present

Apple IIc Plus ($FBBF = $05)

- o Can use the external IIc drive, the UniDisk 3.5 drive, the Apple 3.5 drives, but not the original IIc Memory Expansion Card.
- o Contains a Memory Expansion Card connector
- o 3.5" internal drive replaces 5.25" internal drive
- o Mouse maps to slot 7
- o PR#7 kills the system
- o 4 MHz 65C02 microprocessor
- o Accelerator chip and static RAM cache permit operation up to 4 MHz
- o Keyboard replaced with Apple Standard Keyboard (minus numeric keypad)
- o Internal power supply
- o Internal modem connector
- o Serial ports refitted with mini-DIN 8 connectors
- o Headphone jack has been removed
- o Volume control relocated above the keyboard
- o 40/80 column switch replaced by keyboard (Sholes/Dvorak) switch

Further Reference

- o Apple IIc Technical Reference Manual
- o Apple IIc Technical Note #5, Memory Expansion on the Apple IIc
- o Apple IIc Technical Note #6, Buffering Blues
- o Apple II Miscellaneous Technical Note #2, Apple II Family Identification Routines 2.1
- o Apple II Miscellaneous Technical Note #7, Apple II Family Identification
- o Apple II Miscellaneous Technical Note #8, Pascal 1.1 Firmware Protocol ID Bytes

END OF FILE TN.AIIc.007

\###
\### FILE: TN.AIIc.008
\###

Apple II
Technical Notes

Developer Technical Support

Apple IIc
#8: Single-Sided 3.5” Media and the Apple IIc Plus

Written by: Llew Roberts May 1989

This Technical Note describes a media limitation on the internal drive of the Apple IIc Plus.

With the exception of the internal drive on the Apple IIc Plus, single-sided 3.5" disks are supported on all Apple 3.5" drives, including external disk drives connected to an Apple IIc Plus. The IIc Plus internal disk drive assumes that all disks have an 800K capacity, so it returns valid reads on blocks which occur on the formatted side and I/O errors on blocks which occur on the unformatted side. A disk may appear to work when the disk-reading algorithm has read blocks only from the formatted side.

For these reasons, we suggest that you do not ship programs on single-sided media.

Further Reference

- o Apple IIc Technical Reference Manual

\### END OF FILE TN.AIIc.008

##
FILE: TN.AIIe.001
##

Apple II
Technical Notes

Developer Technical Support

Apple IIe
#1: Overview of the Apple IIe

Revised by: Matt Deatherage November 1988
Revised by: Cameron Birse October 1985

This Technical Note formerly presented an overview of the Apple IIe.

This Note formerly presented an overview of the Apple IIe and its differences from the Apple][and][+. The Apple IIe Technical Reference Manual now documents this information, as well as differences with other members of the Apple II family.

Further Reference

- Apple IIe Technical Reference Manual

END OF FILE TN.AIIe.001

##
FILE: TN.AIIe.002
##

Apple II
Technical Notes

Developer Technical Support

Apple IIe
#2: Hardware Protocol for Doing DMA

Revised by: Glenn A. Baxter & Rob Moore November 1988
Written by: Peter Baum January 1983

This Technical Note explains the hardware protocol for doing direct memory access (DMA) on the Apple IIe and Apple][and is meant as a guideline for developing peripherals which do DMA on these machines, not as a specification for future Apple products or revisions.

This Note covers the timing differences between the Apple][and IIe and also gives tips on how to design a peripheral card that will work in both systems. The reader should be very familiar with either the Apple][+ or the Apple IIe, especially the timing on the data and address buses in relation to the 6502.

DMA is used by peripheral cards in the Apple II family to transfer data directly into memory without benefit of the processor. Transfer of data from a peripheral device into RAM can normally be handled one byte at a time under control of the processor. By using DMA, you can achieve greater data transfer rates than the 6502 can handle in software. This transfer rate can approach the full-cycle time of the memory. This technique can also be used to transfer single data bytes into memory without requiring the CPU to process an interrupt, which can be very time consuming.

The DMA process entails five steps: turn the processor off, gain access to the R/W* line and both address and data buses, complete the data transfer, release the data and address buses, and finally, allow the microprocessor to restart. This Note covers each of these steps in detail.

At this point, I should caution the prospective developer that DMA on an Apple][+ or Apple IIe can only be done under certain circumstances. Because DMA turns off the processor, any program with a software timing loop will not work properly. These programs assume that each instruction will take a fixed amount of time, which is not true when the processor stops in the middle of an instruction. This assumption means that the Apple II disk drives will not work since they require a timing loop to read a disk. (Co-processor cards work with DMA because they initiate the disk access and know that DMA cannot be used until the disk is finished).

Another problem is that because of the mapping scheme used on the Apple IIe extended 80-column (64K) card, a peripheral card cannot tell which memory bank is being used without a complicated detection scheme. This problem means that if a DMA device writes to a certain memory space, it might not be able to read the same data back.

```
 _______
                           INTERNAL DATA BUS (MD 0-7)
|       |<---------+-----+-----------+---------+------------+---------+
|       |          ^     ^           ^         ^            |         |
|       |          |  ___|______     |         |       _____v______   |
|       |          | | Keyboard |  __v__   ____v____  | DATA LATCH |  |
|       |          | |__________| | 64K | | 80 COL. | |____________|  |
|       |       ___|___      ^    | RAM | | AUX RAM |          |      |
|       |      | EO-FF |     |KDB |_____| |_________|--------->+      |
| 6502A |      |  ROM  |     |         ^          ^  VIDEO DATA|      |
|       |      |_______|     |         |          |            |      |
|       |        ^    ^      |      +--+----------+-----+      |   ___v_____
|       |        |    |ROMEN |      |RAM ADDR BUS (0-7) |      |  |  DATA    |
|       |        |   _|______|______|__                 |      |  |  LATCH   |
|       |        |  |      MMU         |------------------------->| 74LS245 |
|       |        |->|__________________|  MD IN/OUT     |      |  |_________|
|       |        |       ^                              |      |      ^
|       |        |       |                            __|__    |      |
|       |        |       |                           | IOU |   |      |
|       |        +---------------------------------->|_____|<--+      |
|       |        |       |                            ^   |    |      |
|       |        |       |TIMING AND CONTROL SIGNALS  |   |    |      |
|       |<---------------+----------------------------+   |    |      |
|       |        |       |                              __v____v__    |
|       |   ___  |       +--------------+              |  VIDEO   |   |
|       |  | 7 | |       |              |              | CIRCUITS |   |
|       |  | 4 | |   ____|_____    _____v______        |__________|   |
|       |  | L | |  |  TIMING  |  | PERIPHERAL |                      |
|       |  | S | |  | CIRCUITS |  |   SLOTS    |<---------------------+
|       |  | 2 | |  |__________|  |____________|   EXTERNAL DATA BUS
|       |  | 4 | |                      ^
|       |  | 4 | |ADDRESS BUS (A0-A15)  |
|_______|->|___|-+----------------------+
```

Figure 1 - Apple IIe Functional Block Diagram

Though the differences between the Apple IIe and Apple][+ architecture appear to be significant to a device which uses DMA, this should not affect the design in most cases. A good rule of thumb is that if a device is designed to work on the Apple IIe, then it will be backward compatible and also run on the Apple][+. The converse is not true; cards that use DMA on the Apple][+ might not work on the Apple IIe, hence, most of the descriptions in this Note refer to the Apple IIe with occasional references to the Apple][+. For example, the timing specifications listed are calculated from the Apple IIe timing paths unless otherwise noted.

Occasionally the descriptions refer to a chip on the motherboard of the Apple IIe, so a set of Apple IIe schematics should be nearby. The corresponding parts on the Apple][+ will be specified when applicable.

The following paragraphs describe and define some of the terms that are used throughout this Note. The Apple IIe block diagram on the previous page may be helpful when reading about the buses.

01 (phase one) time	The time when the 01 system timing signal is high. During this time the data bus, address bus, and RAM are used for video refreshing. This time is also called the video cycle or video phase.
0o (phase zero) time	The time when 0o clock is high. 0o is the inverse of

	01. During this time the microprocessor uses the data and address buses. This time is also known as the CPU cycle or CPU phase.
IOU and MMU	Two MOS custom chips inside the Apple IIe. See chapter 7 of the Apple IIe Technical Reference Manual for more details on the custom chips.
Data bus	The microprocessor, ROM, and RAM are connected to this bus. On the IIe this bus generally has MOS components connected to it rather than TTL and is sometimes called the MOS data bus. A 74LS245 bidirectional bus transceiver (location B2 on the original motherboard) connects this internal bus to an external bus that the outside world sees through the peripheral slots. The data bus connected to the peripheral slots is called the external data bus. The Apple][does not have these two data buses. Instead, the peripheral slots are connected to the ROM, CPU data buffers, and RAM data inputs. The RAM data outputs are multiplexed with the keyboard data onto this bus.
Address bus	There are three different sections to the address bus on the Apple IIe. The first section consists of the addresses from the 6502A into a pair of 74LS244s (locations B1,B3). Part two connects the other side of the '244 to the address bus that the peripheral slots see. Also connected on this bus are the MMU, the ROM, and the chips that decode I/O SELECT, DEVICE SELECT, and I/O STROBE. The third address bus is generated by the custom chips and is only used to access the RAM. The MMU and IOU automatically multiplex this bus with the high byte and low byte of an address during any RAM access, whether it be for video refresh or for a microprocessor instruction fetch. This third bus is called the RAM address bus. The Apple][also has these three buses, but uses 8T97s and discrete logic instead of the 74LS244 and custom chips.
6502 microprocessor	In the Apple IIe a 6502A, a 2 MHz part is used instead of the 1 MHz 6502 used in the Apple][+. Since the custom chips in the Apple IIe are MOS and slower than the TTL in the Apple][+, the faster 6502A was used to guarantee better margins. For example, the 6502A sets up the address bus faster on the Apple IIe than the 6502 does in the Apple][+.

On the IIe, all the timing signals are generated by the PAL timing chip, except for the 7 M signal which is generated from an 74S109 or 74109 (early versions of the IIe). Although both the PAL and the 74S019 use the 14 M signal for a clock, there will be some skew between edges of the 7 M clock and the timing signals from the PAL, such as the edges of 0o or 01. This skew means the 7 M clock edge may rise as much as 20 ns before, or 5 ns after, the 0o falling edge. The clock signals of the Apple][+ should be tighter than this (probably within 5 ns of each other) since 7 M, 0o, and 01 are all generated from the same chip, a 74S175. Take this skew into account whenever using the 7 M signal in a design.

Getting on the Bus (Exact Change Only)

1. Pull DMA low during 01 time.

On the Apple IIe, the DMA line controls the direction of the 74LS245, which enables the internal data bus outwards to the peripheral slots or enables external data onto the internal bus. Changing the state of the DMA line during 0o could cause the '245 to change directions, forcing the internal data bus to go tri-state during a microprocessor read. The 6502 would read garbage and the computer might go belly up by jumping to a random memory location.

On the Apple][, pulling the DMA line always forces the CPU data bus buffer to point inward and drive toward the 6502. Pulling the DMA line low during 0o of a write cycle would result in garbage being written to memory, since the data bus to the RAM would suddenly go tri-state.

2. Wait 30 ns, then assert address bus and R/W* line.

Before driving the address bus and R/W* line, the system must process the transition on the DMA line and release the bus. This requires:

```
  25 ns    'LS244 output disable from low level
  +5 ns    'S02 low to high level output transition
  30 ns    delay from DMA negative edge before driving address
           bus
```

The 30 ns wait will also work on the Apple][, since it only needs 27 ns ('LS04 and 8T97).

```
                        ________________________________
Ph1           _______|
                     |<----------------------->|
                             213 ns (#3)
              ___________________  ___________  ______
Address Bus    High Impedence  \/           \/ Valid
from slots    ___________________/\___________/\______

                   ->|30 ns (#2)|<-
DMA           ________
                      |______________________________
```

Figure 2 - Getting On The Bus

3. Address and R/W* line must be valid within 213 ns of 01 positive edge.

This constraint is needed to meet the setup requirements of the IOU, MMU, and RAM. This time can be derived from the 6502A (2 MHz) setup requirements. The Apple][can wait for 300 ns before data must be valid, because it uses the 1 MHz 6502 which has a longer setup time.

Warning: This specification (the address setup time) is the major cause of failure for cards which use DMA in the Apple IIe. Many DMA cards which were originally designed for the Apple][+ do not meet this specification.

During DMA (Keep Your Hands Inside the Bus at all Times)

1. Don't drive the data bus during 01 time.

   On an Apple][+, it is safe to drive the data bus 35 ns after asserting the R/W* line low, regardless of the point in the timing cycle. When the R/W* line goes low, the 74LS257s at locations B6 and B7 tri-state the data bus, even in the middle of 0o or 01. This action prevents a bus fight from occurring between a DMA device and the system.

   At first glance of the Apple IIe logic schematics, it appears that a bus fight cannot occur on the data bus. During the 01 half of a write cycle, the 74LS245 tri-states the data bus within 30 ns of the R/W* line being pulled low. While this does preclude a fight from occurring on the data bus during 01, it doesn't prevent a bus crash from occurring at the beginning of 0o. At the beginning of 0o, the 74LS245 is enabled and will drive the external data bus. If the peripheral card also drives the data bus, there could be a horrendous bus fight, since the 74LS245 can source 15 ma and sink 24 ma per line. This might cause a spike on the ground plane, which could cause a processor to reset on a co-processor card.

   Let us take a look at the problem by stepping thru Figure 3, a timing diagram.

   The diagram starts with the video cycle of a read operation. During the video cycle, the video refresh data is read from the RAM and put on the data bus. This video data will appear on the peripheral slot (external) data bus because the 74LS245, as can be seen from Table 1, drives outward during 01 of a read cycle.

   Typically, the address bus and R/W* line would be setup by the 6502A during 01 for the next CPU cycle, but instead, a peripheral card pulls the DMA line low. As explained earlier, the peripheral device should wait at least 30 ns before driving the address bus and R/W* line. In this first DMA cycle, the peripheral card wants to read a byte from RAM, so it keeps the R/W* line high.

   At this point we must switch over and use the Apple][+ to explain the timing required to read the data from RAM. The rule of thumb, that designing a DMA card for the Apple IIe will be backward compatible and run on the Apple][+, will not hold here. On the Apple][+ data is valid on the peripheral connector a minimum of 468 ns from the 0o rising edge and holds to at least the falling edge of 0o at 490 ns. The hold time is actually the minimum propagation delay from the falling edge of 0o thru the following chips: 74LS257 at J1, 74LS139 at F2, 74LS20 at D2, 74LS00 at A2, and finally to the enable of the 74LS257s at B5 and B6. On the Apple IIe a byte from RAM becomes valid at most 345 ns after the rising edge of 0o and stays valid until the 0o falling edge.

```
7M
_   _   _   _   _   _   _   _   _   _   _   _   _   _   _   _   _   _   _   _
 |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_|

PH0
```

```
___               _____________               _____________               _____
   |_____________|             |_____________|             |_____________|

PH1
    _____________               _____________               _____________
___|             |_____________|             |_____________|             |_____

R/W                          ->||<-(40-213 ns)
________________________________                              _________________
                                |_____________________________|
                            (5 ns Max)   (5 ns Max)    (5 ns Max)
OE 74LS245                    ->||<-       ->||<-        ->||<-
                                  _____________              _
_________________________________|             |_____________| |_______________
                                                         (5 ns Max)
/DMA                                                       ->||<-
___
   |___________________________________________________________________________

MD IN/OUT                                   ->|  |<-(130 ns Max)
                                                  _____________
_________________________________________________|             |_______________

DATA           ->|            |<-(468 ns*)   ->|  |<-(170 ns Max)
                            _____            ___ _____
___________________________/     \__________/   \/     \_________________________
                           \_____/          \___/\_____/
                            valid                valid
  ->||<-(30 ns Min)
ADDR_  _____________________________  _____________________________  __________
___/ \/          valid              \/          valid              \/    valid
   \_/\_____________________________/\_____________________________/\__________

 ->|  |<-(40-213 ns)                       * This is an Apple ][+ specification
```

Figure 3 - Timing Diagram

In the second DMA cycle, the timing diagram shows the peripheral card writing a byte to memory. In the first phase of the cycle, the video phase, the address bus and R/W* line are setup by the peripheral card within the timing specifications described earlier, 213 ns. Though the direction of the 74LS245 still points toward the slots, the '245 is disabled when the R/W* line is pulled low by the peripheral device (see Table 1). This will tri-state the external data bus. All the signals stay unchanged through the rest of the video phase, until the CPU cycle starts with the rising edge of 0o.

Most bus fights occur at the beginning of the CPU cycle. The CPU cycle begins with address bus and R/W* line setup already and the data bus tri-stated. The signal MD IN/OUT, which drives the 74LS245 direction control, is generated by the MMU and is always low during 01, so the 74LS245 drives toward the slots. MD IN/OUT uses the 0o rising edge to clock itself high during a DMA write cycle, but because the MMU is a MOS chip the delay before MD IN/OUT finally rises can be as long as 130 ns from the 0o rising edge. Hence, at the beginning of 0o the 74LS245 is in tri-state mode, but with the direction set to drive toward the peripheral slots.

PHO	R/W		Stable State of 74LS245
1	0	(Write to RAM)	High impedance
1	1	(Read from RAM)	Outward driving external data bus (slots)
0	0	(Write to RAM)	Inward driving into RAM
0	1	(Read from RAM)	Outward driving external data bus (slots)

Table 1-Stable State of 74LS245

Within 5 ns after 0o goes high, the chip enable to the 74LS245 goes low, enabling data onto the external data bus. The 74LS245 specification guarantees that the data will be valid within 40 ns from the chip enable. If the peripheral device was also driving the bus, there would be a bus crash. To prevent this bus crash, the data bus cannot be driven during 01, unless the data is pulled off the bus before 0o goes high. This means that the rising edge of 0o cannot be used to gate data on and off the bus. The bus fight will occur before the peripheral card can tri-state the data bus.

Data can only be enabled onto the bus after the 74LS245 has changed directions and is driving the internal data bus. The DMA device must allow 130 ns for the MD IN/OUT line to change, plus the delay for the 74LS245 to change directions which takes 25 ns, for a total of 155 ns.

After this 155 ns, the data must be valid on the bus within 55 ns, because the RAM requires data be setup at the CAS falling edge, which occurs 210 ns into 0o. This does not leave any time to spare, since, for example, a 74LS245 has a 40 ns enable time. This timing criteria will also work for the Apple][+ since the setup time for 16K RAM is the same as the 64K RAM, and CAS also falls at 210 ns. The data hold time of 55 ns after CAS falls is also the same for both the Apple IIe and the Apple][+.

Here is a scenario for a DMA write. Set up the address bus and R/W* line within the required 213 ns, then wait for the first 7 M (pin 36 on slot) falling edge after 0o goes high before enabling your buffer onto the data bus. This edge will occur at 140 ns into 0o, and when the gate delay is added, should guarantee the buffer will not be driving the bus in the first 155 ns. I don't advocate depending on a minimum gate delay as standard design practice (my college professor thinks public whipping would be a justifiable punishment) but this is the real world (I'm not getting graded anyway). The data bus is valid by the time CAS falls, and should be stable for at least another 55 ns or until 0o falls.

2. The processor can be held off for a total of 10 microseconds. (10 0o clock cycles).

   This is true if a Rockwell 6502 is being used. (A Synertek part can be held off for as long as 40 µsec.) This time is the maximum cycle time of the 6502 and if there are no clock transitions within this time, it could result in internal registers (A,X,Y) losing data. This maximum time varies from manufacturer to manufacturer of the 6502.

3. MMU and IOU multiplex address bus

The custom chips automatically handle the multiplexing required of the RAM address bus. The external device doing DMA must set up the address bus and R/W* line within 213 ns of the rising edge of 01 just like the 6502A does. The custom chips will automatically generate the addresses to the RAM for the video refresh cycle during 01, but use the addresses from the address bus to set up for the next instruction cycle. Hence, the only consideration on the address bus during DMA is to meet the 213 ns setup time requirement.

The 213 ns setup time will also work with the Apple][since it can take as long as 300 ns to set up the address bus and R/W* line.

Getting Off the Bus

1. Don't release DMA during 0o.

This is analogous to step 1 of Getting on the Bus. If the DMA line is released during 0o the microprocessor will try to execute a cycle during this time without the data or address bus set up properly. This random instruction fetch will probably cause the system to crash.

2. Tri-state address bus drivers on peripheral slots

The DMA line is holding off the addresses from the 6502 onto the internal address bus by tri-stating the two 74LS244s on the Apple IIe bus and the 8T97s on the Apple][+ bus. The address bus and R/W* line from the external device in the peripheral slots should be tri-stated before releasing the DMA line or a bus fight will occur between the internal bus buffers and the peripheral slot drivers.

3. Release DMA line

These last two steps are the opposite of the first two steps required to get on the bus. Both of these steps, releasing the address and R/W* lines then the DMA line, should be done within 178 ns of 01 going high. This allows time for the 6502A to set up the address and R/W* lines properly for the next instruction cycle.

```
  213 ns    address set up requirements
    5 ns    'S02 output high-to-low transition
  -30 ns    'LS244 out enable time
  178 ns    to release DMA line and allow 6502 to set up address bus
```

Again, the Apple][can take longer, up to 260 ns, before releasing the DMA line.

Further Reference

- Apple IIe Technical Reference Manual

END OF FILE TN.AIIe.002

##
FILE: TN.AIIe.003
##

Apple II
Technical Notes

Developer Technical Support

Apple IIe
#3: Double High-Resolution Graphics

Revised by: Matt Deatherage, Glenn A. Baxter & Cameron Birse November 1988
Written by: Peter Baum September 1983

This Technical Note is a tutorial on double high-resolution (hi-res) graphics, a feature available on 128K Apple IIe, IIc, and IIGS computers.

Introduction

This Note was originally written in the early days of double high-resolution graphics. At that time, there was no Apple IIc or IIGS, therefore, some of the things originally said may seem a little strange today, five years later.

For example, this Note talks a fair amount about being sure that you have a Revision B Apple IIe with the jumper installed. All Apple IIe computers shipped since about mid-1983 have a Revision B motherboard, so this is not that big a concern anymore; furthermore, nearly every IIe out there has the aforementioned jumper already installed (it is not even an option on some third-party 80-column cards for the IIe).

Also, the IIc and IIGS are functionally equivalent (for the purposes of this article) to a Revision B IIe with the properly-jumpered 80-column card installed, and most of the references made to the Apple IIe apply equally to the IIc and IIGS. We have tried to update most of the references to avoid confusion.

Considering the myriad of programming utilities, games, graphics programs, and other software that now uses double high-resolution graphics, it is probable that this Note will not be as vital as it once was. If you are writing in AppleSoft BASIC, you will probably find it easier to purchase a commercial double hi-res BASIC utility package to add double hi-res commands to AppleSoft, rather than writing your own routines. Similarly, those who want double hi-res art will find a double hi-res art application much easier than trying to draw it from the monitor or machine language.

However, if you have the insatiable curiosity about these things that Apple II owners and developers so often are blessed (cursed?) with, this Note will show you how double high-resolution works, as well as giving a few type-along examples in the monitor to get your feet wet.

This article describes the double high-resolution display mode which is available in the Apple IIc, IIGS, and the Apple IIe (with an extended 80-column card). Double hi-res graphics provides twice the horizontal resolution and more colors than the standard high-resolution mode. On a monochrome monitor, double hi-res displays 560 horizontal by 192 vertical pixels, while on a color monitor, it allows the use of 16 colors.

Double High-Resolution on the Apple II Series

What is It?

The double high-resolution display mode that is available for the Apple IIe provides twice the horizontal resolution of the standard high-resolution mode. On a standard black-and-white video monitor, standard hi-res displays 280 columns and 192 rows of picture elements (pixels); the double hi-res mode displays 560 x 192 pixels. On a color monitor, the standard hi-res mode displays up to 140 columns of colors, each color being selected from the group of six colors available, with certain limitations. Double hi-res displays 140 columns of color, for which all 16 of the low-resolution colors are available.

	Black/White	Color
Standard Hi-Res	280 x 192 pixels	140 columns 6 colors
Double Hi-Res	560 x 192 pixels	140 columns 16 colors

Table 1-Comparison of Standard and Double Hi-Res Graphics

How Do I Install It?

Installation of the double hi-res mode on your Apple IIe depends on the following three conditions, discussed in detail below:

1. Presence of a Revision B motherboard
2. Installation of an extended 80-column text card with jumper
3. A video monitor with a bandwidth of at least 14 MHz

First, your Apple IIe must have a Revision B (Rev-B) motherboard. To find out whether your computer's motherboard is a Rev-B, check the part number on the edge of the board nearest the back panel, above the slots. If the board is a Rev-B, the part number will be 820-0064-B. (Double hi-res does not work on systems containing a Rev-A motherboard.) If your computer's motherboard is not a Rev-B, and if you want to obtain one, contact your local Apple dealer.

The second condition for installing double hi-res on your IIe is that it must have an extended 80-column text card installed. This card must be installed with a jumper connecting the two Molex-type pins on the board.

Warning: If your IIe has a Rev-A motherboard, do not use an extended 80-column card with the jumper connection mentioned above; the system will not work at all if you do.

The last requirement for operation in double hi-res mode is that your video monitor must have a bandwidth of at least 14 MHz. This bandwidth is necessary because a television set that requires a modulator will not reproduce some characters or graphic elements clearly, due to the high speed at which the computer sends out dots in this mode. Because most of the video monitors having a bandwidth of up to 14 MHz are black-and-white, the working examples in this article do not apply to color monitors. If you have a video monitor, please use it--instead of a television set--to display the following examples. The AppleColor composite monitors will work just fine.

Your Turn to be Creative (Volunteers, Anyone?)

The tutorial that occupies the rest of this Note assumes you are working at your Apple II as you read. The second part of the lesson demonstrates the double hi-res mode; therefore, before embarking on the second part, you should install a jumpered extended 80-column card in your Rev-B Apple IIe (or use any Apple IIc or IIGS).

Hands-On Practice with Standard Hi-Res

The Apple II hi-res graphics display is bit-mapped. In other words, each dot on the screen corresponds to a bit in the computer's memory. For a real-life example of bit-mapping, perform the following procedure, according to the instructions given below. (The symbol <cr> indicates a carriage return.)

1. Boot the system.
2. Engage the Caps Lock key, and type HGR<cr>. (This instruction should clear the top of the screen.)
3. Type CALL -151 <cr>. (The system is now in the monitor mode, and the prompt should appear as an asterisk (*).)
4. Type 2100:1 <cr>. One single dot should appear in the upper left-hand corner of the screen.

Congratulations! You have just plotted your first hi-res pixel. (Not an astonishing feat, but you have to start somewhere.)

With a black-and-white monitor, the bits in memory have a simple correspondence with the dots (pixels) on the screen. A dot of light appears if the corresponding bit is set (has a value of 1), but remains invisible if the bit is off (has a value of zero). (The dot appears white on a black-and-white monitor, and green on a green-screen monitor, such as Apple's Monitor /// or Monitor II. For simplicity, we shall refer to an invisible dot as a black dot or pixel.) Two visible dots located next to each other appear as a single wide dot, and many adjacent dots appear as a line. To obtain a display of another dot and a line, follow the steps listed below:

1. Type 2080:40 <cr>. A dot should appear above and to the right of the dot you produced in the last exercise.
2. Type 2180:7F <cr>. A small horizontal line should appear below the first dot you produced.

From Bits and Bytes to Pixels

The seven low-order bits in each display byte control seven adjacent dots in a row. A group of 40 consecutive bytes in memory controls a row of 280 dots (7 dots per byte, multiplied by 40 bytes). In the screen display, the least-significant bit of each byte appears as the leftmost pixel in a group of 7 pixels. The second least-significant bit corresponds to the pixel directly to the right of the pixel previously displayed, and so on. To watch this procedure in action, follow the steps listed below. The dots will appear in the middle of your screen.

1. Type 2028:1 <cr>.
2. Type 2828:2 <cr>.
3. Type 3028:4 <cr>.

The three bits you specified in this exercise correspond to three pixels that are displayed one after another, from left to right.

The most-significant bit in each byte does not correspond to a pixel. Instead, this bit is used to shift the positions of the other seven bits in the byte. For a demonstration of this feature, follow the steps listed below:

1. Type 2050:8 <cr>.
2. Type 2850:8 <cr>.
3. Type 3050:8 <cr>.

You will notice that the dots align themselves vertically. Now do the following:

4. Type 2450:88 <cr>.

The new dot (that is, the one that corresponds to the bit you just specified) does not line up with the dots you displayed earlier. Instead, it appears to be shifted one "half-dot" to the right.

5. To demonstrate that this dot really is a new dot, and not just the old dot shifted by one dot position, type 2050:18 <cr>, 2850:18 <cr>.

You will notice that the dot mentioned under step 4 (the dot that was not aligned with the other seven dots) is straddled by the dots above and below it. (The use of magnifying lenses is permitted.)

Shifting the pixel one half-dot, by setting the high, most-significant bit is most often used for color displays. When the high bit of a byte is set to generate this shifted dot (which is also called the half-dot shift), then all the dots for that byte will be shifted one half dot. The half-dot shift does not exist in the double hi-res mode.

The Figure 1 shows the memory map for the standard hi-res graphics mode.

```
                        Horizontal Offset
Base | $00 | $01 | $02 | $03         $24 | $25 | $26 | $27 |
_____|_____|_____|_____|_____       _____|_____|_____|_____|
$2000|     |     |     |     |     |     |     |     |     |
$2080|     |     |     |     |     |     |     |     |     |
$2100|     |     |     |     |     |     |     |     |     |
$2180|     |     |     |     |     |     |     |     |     |
$2200|     |     |     |     |     |     |     |     |     |
$2280|     |     |     |     |     |     |     |     |     |
$2300|     |     |     |     |     |     |     |     |     |
$2380|     |     |     |     |     |     |     |     |     |
$2028|     |     |     |      \     \      |     |     |     |
$20A8|     |     |     |       \     \     |     |     |     |
$2128|     |     |     |       /     /     |     |     |     |
$21A8|     |     |     |      /     /      |     |     |     |
$2228|     |     |     |     /     /       |     |     |     |
$22A8|     |     |     |    /     /        |     |     |     |
$2328|     |     |     |   /     /         |     |     |     |
$23A8|     |     |     |   \     \         |     |     |     |
```

```
     $2050|     |     |     |    \     \      |     |     |     |
     $20D0|     |     |     |     |     |     |     |     |     |
     $2150|     |     |     |     |     |     |     |     |     |
     $21D0|     |     |     |     |     |     |     |     |     |
     $2250|     |     |     |     |     |     |     |     |     |
     $22D0|     |     |     |     |     |     |     |     |     |
     $2350|     |     |     |     |     |     |     |     |     |
     $23D0|     |     |     |     |     |     |     |     |     |
```

Figure 1 - Standard Hi-Res Memory Map

Note: This memory map is in Chapter 2 of the Apple IIe Technical Reference,
First Printing, January 1987.

Figure 2 shows the box subdivisions for the memory map in Figure 1.

```
               Offset              Bit
               from                                  LSB
               base    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
               ________|___|___|___|___|___|___|___|
               +$0000 |   |   |   |   |   |   |   |
               +$0400 |   |   |   |   |   |   |   |
               +$0800 |   |   |   |   |   |   |   |
               +$0C00 |   |   |   |   |   |   |   |
               +$1000 |   |   |   |   |   |   |   |
               +$1400 |   |   |   |   |   |   |   |
               +$1800 |   |   |   |   |   |   |   |
               +$1C00 |   |   |   |   |   |   |   |
```

Figure 2 - Box Subdivisions of the Standard Memory Map

Note: This figure is the inset of the hi-res graphics display map in
Chapter 2 of the Apple IIe Technical Reference, First Printing, January
1987.

For example, the first memory address of each screen line for the first few lines is as follows:

$2000, $2400, $2800, $2C00, $3000, $3400, $3800, $3C00, $2080, $2480, etc.

Each of the 24 boxes contains 8 screen lines for a total of 192 vertical lines per screen. Each of the 40 boxes per line contains 7 pixels for a total of 280 pixels horizontally across each line.

The Intricacies of Double Hi-Res

Because the double high-resolution graphics mode provides twice the horizontal dot density as standard hi-res graphics does, double hi-res requires twice as much memory as does standard hi-res. If you spent many hours committing the standard hi-res memory map to memory, don't despair; double hi-res still uses the hi-res graphics page (but only to represent half the picture, so to speak). In the double hi-res mode, the hi-res graphics page is compressed to fit into half of the display. The other half of the display is stored in memory (called the auxiliary (aux) memory) on the extended 80-column card. (This article refers to the standard hi-res graphics page, which resides in main memory, as the motherboard (main) memory.)

The auxiliary memory uses the same addresses used by the standard hi-res graphics page (page 1, $2000 through $3FFF). The hi-res graphics page stored in auxiliary memory is known as hi-res page 1X. The graphics pages in auxiliary memory are bank-switched memory, which you can switch in by activating some of the soft switches. (Adventurous readers may want to skip ahead to Using the Auxiliary Memory, which appears later in this Note.)

The memory mapping for the hi-res graphics display is analogous to the technique used for the 80-column display. The double hi-res display interleaves bytes from the two different memory pages (auxiliary and motherboard). Seven bits from a byte in the auxiliary memory bank are displayed first, followed by seven bits from the corresponding byte on the motherboard. The bits are shifted out the same way as in standard hi-res (least-significant bit first). In double hi-res, the most significant bit of each byte is ignored; thus, no half-dot shift can occur. (This feature is important, as you will see when we examine double hi-res in color.)

The memory map for double hi-res appears in Figure 3.

```
                        Horizontal Offset
       $00   $01   $02   $03         $24   $25   $26   $27
         M     M     M     M           M     M     M     M
      A  a  A  a  A  a  A  a        A  a  A  a  A  a  A  a
     |u  i |u  i |u  i |u  i        u  i |u  i |u  i |u  i |
Base |x |n |x |n |x |n |x |n        x |n |x |n |x |n |x |n |
_____|__|__|__|__|__|__|__|__       __|__|__|__|__|__|__|__|
$2000|     |     |     |     |     |     |     |     |     |
$2080|     |     |     |     |     |     |     |     |     |
$2100|     |     |     |     |     |     |     |     |     |
$2180|     |     |     |     |     |     |     |     |     |
$2200|     |     |     |     |     |     |     |     |     |
$2280|     |     |     |     |     |     |     |     |     |
$2300|     |     |     |     |     |     |     |     |     |
$2380|     |     |     |     |     |     |     |     |     |
$2028|     |     |     |      \     \    |     |     |     |
$20A8|     |     |     |       \     \   |     |     |     |
$2128|     |     |     |       /     /   |     |     |     |
$21A8|     |     |     |      /     /    |     |     |     |
$2228|     |     |     |     /     /     |     |     |     |
$22A8|     |     |     |    /     /      |     |     |     |
$2328|     |     |     |   /     /       |     |     |     |
$23A8|     |     |     |   \     \       |     |     |     |
$2050|     |     |     |    \     \      |     |     |     |
$20D0|     |     |     |     |     |     |     |     |     |
$2150|     |     |     |     |     |     |     |     |     |
$21D0|     |     |     |     |     |     |     |     |     |
$2250|     |     |     |     |     |     |     |     |     |
$22D0|     |     |     |     |     |     |     |     |     |
$2350|     |     |     |     |     |     |     |     |     |
$23D0|     |     |     |     |     |     |     |     |     |
```

Figure 3 - Double Hi-Res Memory Map

Note: This memory map is in Chapter 2 of the Apple IIe Technical Reference, First Printing, January 1987.

Each box is subdivided exactly the same way it is in the standard hi-res mode.

Obtaining a Double-Hi-Res Display

To display the double hi-res mode, set the following soft switches:

	In the monitor	In AppleSoft
	Read	PEEK
HI-RES	$C057	49239
GR	$C050	49232
AN3	$C05E	49246
MIXED	$C053	49235

	In the monitor	In AppleSoft
	Write	POKE
80COL	$C00D	49165,0

Annunciator 3 (AN3) must be turned off to get into double hi-res mode. You turn it off by reading location 49246 ($C05E). Note that whenever you press Control-Reset, AN3 is turned on; therefore, each time you press Control-Reset, you must turn AN3 off again.

If you are using MIXED mode, then the bottom four lines on the screen will display text. If you have not turned on the 80-column card, then every second character in the bottom four lines of text will be a random character. (The reason is that although the hardware displays 80 columns of characters, the firmware only updates the 40-column screen, which consists of the characters in the odd-numbered columns. The characters in even-numbered columns then consist of random characters taken from text page 1X in the auxiliary memory.)

To remove the even characters from the bottom four lines on the screen, type PR#3<CR> from AppleSoft (type 3^P in the monitor). This procedure clears the memory locations on page 1X.

Using the Auxiliary Memory

The auxiliary memory consists of several different sections, which you can select by using the soft switches listed below. A pair of memory locations is dedicated to each switch. (One location turns the switch on; the other turns it off.) You activate a switch by writing to the appropriate memory location. The write instruction itself is what activates the switch; therefore, it does not matter what data you write to the memory location. The soft switches are as follows:

		In the monitor	In AppleSoft
		Write	POKE
80STORE	off	$C000	49152,0
	on	$C001	49153,0
RAMRD	off	$C002	49154,0
	on	$C003	49155,0
RAMWRT	off	$C004	49156,0
	on	$C005	49157,0
PAGE2	off	$C054	49236,0
	on	$C055	49237,0
HIRES	off	$C056	49238,0
	on	$C057	49239,0

A routine called AUXMOVE ($C311), located in the 80-column firmware, is also very handy, as we will see below.

Accessing memory on the auxiliary card with the soft switches has the following characteristics. Memory maps, which help clarify the descriptions, are in Figures 4, 5, and 6.

1. To activate the PAGE2 and HIRES switches, you need only read (PEEK) from the corresponding memory locations (instead of writing to them, as you do for the other three switches).
2. The PAGE2 switch normally selects the display page, in either graphics or text mode, from either page 1 or page 2 of the motherboard memory. However, it does so only when the 80STORE switch is off.
3. If the 80STORE switch is on, then the function of the PAGE2 switch changes. When 80STORE is on, then PAGE2 switches in the text page, locations $400-7FF, from auxiliary memory (text page 1X), instead of switching the display screen to the alternate video page (page 2 on the motherboard). When 80STORE is on, the PAGE2 switch determines which memory bank (auxiliary or motherboard) is used during any access to addresses $400 through 7FF. When the 80STORE switch is on, it has priority over all other switches.
4. If the 80STORE switch is on, then the PAGE2 switch only switches in the graphics page 1X from the auxiliary memory if the HIRES switch is also on. (Note that this circumstance is slightly different from that described in item 3.) When 80STORE is on, and if the HIRES switch is also on, then the PAGE2 switch selects the memory bank (auxiliary or motherboard) for accesses to a memory location within the range $2000 through 3FFF. If the HIRES switch is off, then any access to an memory location within the range $2000 through 3FFF uses the motherboard memory, regardless of the state of the PAGE2 switch.
5. If the 80STORE switch is off, and if the RAMRD and RAMWRT switches are on, then any reading from or writing to address space $200-$BFFF gains access to the auxiliary memory. If only one of the switches, RAMRD, for example, is set, then only the appropriate operation (in this case a read) will be performed on the auxiliary memory. If only RAMWRT is set, then all write operations access the auxiliary memory. When the 80STORE switch is on, it has higher priority than the RAMRD and RAMWRT switches.

```
       Main Memory                      Auxiliary Memory
       ___________  _________________  ___________
$FFFF |///////////|                   |           |
      |///Bank////|                   |   Bank    |
      |/Switched//|                   | Switched  |
      |//Memory///|                   |  Memory   |
      |///////////|                   |           |
      |___________|___________  _____ |___________|___________
$DFFF |///////////|///////////|       |           |           |
      |//Bank 1///|//Bank 2///|       |  Bank 1   |  Bank 2   |
$D000 |///////////|///////////|       |           |           |
      |___________|___________| _____ |___________|___________|
       ___________  _________________  ___________
$BFFF |///////////|                   |           |
      |///////////|                   |           |
      |///////////|                   |           |
      |///////////|                   |           |
      |///////////|                   |           |
```

```
       |___________|  ________________  |___________|
$5FFF  |///////////|                    |           |
       |//Hi-Res///|                    |  Hi-Res   |
       |/Graphics//|                    | Graphics  |
       |//Page 2///|                    |  Page 2X  |
$4000  |///////////|                    |           |
       |___________|  ________________  |___________|
$3FFF  |///////////|                    |           |
       |//Hi-Res///|                    |  Hi-Res   |
       |/Graphics//|                    | Graphics  |
       |//Page 1///|                    |  Page 1X  |
$2000  |///////////|                    |           |
       |___________|  ________________  |___________|
       |///////////|                    |           |
       |///////////|                    |           |
       |///////////|                    |           |
       |___________|  ________________  |___________|
$0BFF  |///////////|                    |           |
       |///Text////|                    |   Text    |
       |//Page 2///|                    |  Page 2X  |
$0800  |///////////|                    |           |
       |___________|  ________________  |___________|
$07FF  |///////////|                    |           |
       |///Text////|                    |   Text    |
       |//Page 1///|                    |  Page 1X  |
$0400  |///////////|                    |           |
       |___________|  ________________  |___________|
       |///////////|                    |           |
       |///////////|                    |           |
       |///////////|                    |           |
       |___________|  ________________  |___________|
$01FF  |///////////|                    |           |
       |/Stack and/|                    | Alternate |
       |/Zero Page/|                    | Stack and |
       |///////////|                    | Zero Page |
$0000  |///////////|                    |           |
       |___________|  ________________  |___________|

  _________________________    ______
 | 80STORE       | OFF | ON  |  |//////|
 |_______________|_____|_____|  |//////|
 | PAGE2         |  X  | OFF |  |//////|
 |_______________|_____|_____|  |______|
 | HIRES         |  X  |  X  |   Active
 |_______________|_____|_____|   Memory
 | RAMRD/RAMWRT  | OFF | OFF |
 |_______________|_____|_____|
```

Figure 4A - Memory Map One-A

```
        Main Memory                      Auxiliary Memory
        ___________   ________________   ___________
$FFFF  |///////////|                    |           |
       |///Bank////|                    |   Bank    |
       |/Switched//|                    | Switched  |
       |//Memory///|                    |  Memory   |
       |///////////|                    |           |
       |___________|__________   _____  |___________|__________
```

```
$DFFF |///////////|///////////|        |           |           |
      |//Bank 1///|//Bank 2///|        |  Bank 1   |  Bank 2   |
$D000 |///////////|///////////|        |           |           |
      |___________|___________| ______ |___________|___________|

       ___________   _________________   ___________
$BFFF |           |                     |///////////|
      |           |                     |///////////|
      |           |                     |///////////|
      |           |                     |///////////|
      |           |                     |///////////|
      |___________|  _________________  |___________|
$5FFF |           |                     |///////////|
      |  Hi-Res   |                     |//Hi-Res///|
      | Graphics  |                     |/Graphics//|
      |  Page 2   |                     |//Page 2X//|
$4000 |           |                     |///////////|
      |___________|  _________________  |___________|
$3FFF |           |                     |///////////|
      |  Hi-Res   |                     |//Hi-Res///|
      | Graphics  |                     |/Graphics//|
      |  Page 1   |                     |//Page 1X//|
$2000 |           |                     |///////////|
      |___________|  _________________  |___________|
      |           |                     |///////////|
      |           |                     |///////////|
      |           |                     |///////////|
      |___________|  _________________  |___________|
$0BFF |           |                     |///////////|
      |   Text    |                     |///Text////|
      |  Page 2   |                     |//Page 2X//|
$0800 |           |                     |///////////|
      |___________|  _________________  |___________|
$07FF |           |                     |///////////|
      |   Text    |                     |///Text////|
      |  Page 1   |                     |//Page 1X//|
$0400 |           |                     |///////////|
      |___________|  _________________  |___________|
      |           |                     |///////////|
      |           |                     |///////////|
      |           |                     |///////////|
      |___________|  _________________  |___________|
$01FF |///////////|                     |           |
      |/Stack and/|                     | Alternate |
      |/Zero Page/|                     | Stack and |
      |///////////|                     | Zero Page |
$0000 |///////////|                     |           |
      |___________|  _________________  |___________|

 ___________________________     ______
| 80STORE       | OFF | ON  |   |//////|
|_______________|_____|_____|   |//////|
| PAGE2         |  X  | ON  |   |//////|
|_______________|_____|_____|   |______|
| HIRES         |  X  |  X  |    Active
|_______________|_____|_____|    Memory
| RAMRD/RAMWRT  | ON  | ON  |
|_______________|_____|_____|
```

```
                   Figure 4B - Memory Map One-B

       Main Memory                      Auxiliary Memory
       ___________  _________________  ___________
$FFFF |///////////|                   |           |
      |///Bank////|                   |   Bank    |
      |/Switched//|                   | Switched  |
      |//Memory///|                   |  Memory   |
      |///////////|                   |           |
      |___________|___________   _____|___________|___________
$DFFF |///////////|///////////|       |           |           |
      |//Bank 1///|//Bank 2///|       |  Bank 1   |  Bank 2   |
$D000 |///////////|///////////|       |           |           |
      |___________|___________| _____ |___________|___________|
       ___________  _________________  ___________
$BFFF |///////////|                   |           |
      |///////////|                   |           |
      |///////////|                   |           |
      |///////////|                   |           |
      |///////////|                   |           |
      |___________| _________________ |___________|
$5FFF |///////////|                   |           |
      |//Hi-Res///|                   |  Hi-Res   |
      |/Graphics//|                   | Graphics  |
      |//Page 2///|                   |  Page 2X  |
$4000 |///////////|                   |           |
      |___________| _________________ |___________|
$3FFF |///////////|                   |           |
      |//Hi-Res///|                   |  Hi-Res   |
      |/Graphics//|                   | Graphics  |
      |//Page 1///|                   |  Page 1X  |
$2000 |///////////|                   |           |
      |___________| _________________ |___________|
      |///////////|                   |           |
      |///////////|                   |           |
      |///////////|                   |           |
      |___________| _________________ |___________|
$0BFF |///////////|                   |           |
      |///Text////|                   |   Text    |
      |//Page 2///|                   |  Page 2X  |
$0800 |///////////|                   |           |
      |___________| _________________ |___________|
$07FF |           |                   |///////////|
      |   Text    |                   |///Text////|
      |  Page 1   |                   |//Page 1X//|
$0400 |           |                   |///////////|
      |___________| _________________ |___________|
      |///////////|                   |           |
      |///////////|                   |           |
      |///////////|                   |           |
      |___________| _________________ |___________|
$01FF |///////////|                   |           |
      |/Stack and/|                   | Alternate |
      |/Zero Page/|                   | Stack and |
      |///////////|                   | Zero Page |
$0000 |///////////|                   |           |
      |___________| _________________ |___________|
```

```
 __________________________    ______
| 80STORE       | ON  |     |  |//////|
|_______________|_____|_____|  |//////|
| PAGE2         | ON  |     |  |//////|
|_______________|_____|_____|  |______|
| HIRES         | OFF |     |   Active
|_______________|_____|_____|   Memory
| RAMRD/RAMWRT  | OFF |     |
|_______________|_____|_____|
```

Figure 5A - Memory Map Two-A

```
       Main Memory                      Auxiliary Memory
       ___________  _________________  ___________
$FFFF |///////////|                   |           |
      |///Bank////|                   |   Bank    |
      |/Switched//|                   | Switched  |
      |//Memory///|                   |  Memory   |
      |///////////|                   |           |
      |___________|___________  _____ |___________|___________
$DFFF |///////////|///////////|       |           |           |
      |//Bank 1///|//Bank 2///|       |  Bank 1   |  Bank 2   |
$D000 |///////////|///////////|       |           |           |
      |___________|___________| _____ |___________|___________|
       ___________  _________________  ___________
$BFFF |///////////|                   |           |
      |///////////|                   |           |
      |///////////|                   |           |
      |///////////|                   |           |
      |///////////|                   |           |
      |___________| _________________ |___________|
$5FFF |///////////|                   |           |
      |//Hi-Res///|                   |  Hi-Res   |
      |/Graphics//|                   | Graphics  |
      |//Page 2///|                   |  Page 2X  |
$4000 |///////////|                   |           |
      |___________| _________________ |___________|
$3FFF |           |                   |///////////|
      |  Hi-Res   |                   |//Hi-Res///|
      | Graphics  |                   |/Graphics//|
      |  Page 1   |                   |//Page 1X//|
$2000 |           |                   |///////////|
      |___________| _________________ |___________|
      |///////////|                   |           |
      |///////////|                   |           |
      |///////////|                   |           |
      |___________| _________________ |___________|
$0BFF |///////////|                   |           |
      |///Text////|                   |   Text    |
      |//Page 2///|                   |  Page 2X  |
$0800 |///////////|                   |           |
      |___________| _________________ |___________|
$07FF |           |                   |///////////|
      |   Text    |                   |///Text////|
      |  Page 1   |                   |//Page 1X//|
$0400 |           |                   |///////////|
      |___________| _________________ |___________|
      |///////////|                   |           |
```

```
      |///////////|                    |           |
      |///////////|                    |           |
      |___________|  _________________  |___________|
$01FF |///////////|                    |           |
      |/Stack and/|                    | Alternate |
      |/Zero Page/|                    | Stack and |
      |///////////|                    | Zero Page |
$0000 |///////////|                    |           |
      |___________|  _________________  |___________|

 ____________________________    ______
| 80STORE      | ON  |     |  |//////|
|______________|_____|_____|  |//////|
| PAGE2        | ON  |     |  |//////|
|______________|_____|_____|  |______|
| HIRES        | ON  |     |   Active
|______________|_____|_____|   Memory
| RAMRD/RAMWRT | OFF |     |
|______________|_____|_____|

                    Figure 5B - Memory Map Two-B

       Main Memory                      Auxiliary Memory
       ___________  _________________   ___________
$FFFF |///////////|                    |           |
      |///Bank////|                    |  Bank     |
      |/Switched//|                    | Switched  |
      |//Memory///|                    | Memory    |
      |///////////|                    |           |
      |___________|___________   _____ |___________|___________
$DFFF |///////////|///////////|        |           |           |
      |//Bank 1///|//Bank 2///|        |  Bank 1   |  Bank 2   |
$D000 |///////////|///////////|        |           |           |
      |___________|___________|  _____ |___________|___________|
       ___________  _________________   ___________
$BFFF |           |                    |///////////|
      |           |                    |///////////|
      |           |                    |///////////|
      |           |                    |///////////|
      |           |                    |///////////|
      |___________|  _________________  |___________|
$5FFF |           |                    |///////////|
      |  Hi-Res   |                    |//Hi-Res///|
      | Graphics  |                    |/Graphics//|
      |  Page 2   |                    |//Page 2X//|
$4000 |           |                    |///////////|
      |___________|  _________________  |___________|
$3FFF |           |                    |///////////|
      |  Hi-Res   |                    |//Hi-Res///|
      | Graphics  |                    |/Graphics//|
      |  Page 1   |                    |//Page 1X//|
$2000 |           |                    |///////////|
      |___________|  _________________  |___________|
      |           |                    |///////////|
      |           |                    |///////////|
      |           |                    |///////////|
      |___________|  _________________  |___________|
$0BFF |           |                    |///////////|
```

```
      |   Text    |                   |///Text////|
      |  Page 2   |                   |//Page 2X//|
$0800 |           |                   |///////////|
      |___________| _________________ |___________|
$07FF |///////////|                   |           |
      |///Text////|                   |   Text    |
      |//Page 1///|                   |  Page 1X  |
$0400 |///////////|                   |           |
      |___________| _________________ |___________|
      |           |                   |///////////|
      |           |                   |///////////|
      |           |                   |///////////|
      |___________| _________________ |___________|
$01FF |///////////|                   |           |
      |/Stack and/|                   | Alternate |
      |/Zero Page/|                   | Stack and |
      |///////////|                   | Zero Page |
$0000 |///////////|                   |           |
      |___________| _________________ |___________|


 __________________________    ______
| 80STORE      | ON  |     |  |//////|
|______________|_____|_____|  |//////|
| PAGE2        | OFF |     |  |//////|
|______________|_____|_____|  |______|
| HIRES        | OFF |     |   Active
|______________|_____|_____|   Memory
| RAMRD/RAMWRT | ON  |     |
|______________|_____|_____|
```

Figure 6A - Memory Map Three-A

```
       Main Memory                      Auxiliary Memory
       ___________  _________________  ___________
$FFFF |///////////|                   |           |
      |///Bank////|                   |   Bank    |
      |/Switched//|                   | Switched  |
      |//Memory///|                   |  Memory   |
      |///////////|                   |           |
      |___________|___________  _____ |___________|___________
$DFFF |///////////|///////////|       |           |           |
      |//Bank 1///|//Bank 2///|       |  Bank 1   |  Bank 2   |
$D000 |///////////|///////////|       |           |           |
      |___________|___________| _____ |___________|___________|
       ___________  _________________  ___________
$BFFF |           |                   |///////////|
      |           |                   |///////////|
      |           |                   |///////////|
      |           |                   |///////////|
      |           |                   |///////////|
      |___________| _________________ |___________|
$5FFF |           |                   |///////////|
      |  Hi-Res   |                   |//Hi-Res///|
      | Graphics  |                   |/Graphics//|
      |  Page 2   |                   |//Page 2X//|
$4000 |           |                   |///////////|
      |___________| _________________ |___________|
$3FFF |///////////|                   |           |
```

```
      |//Hi-Res///|                   |  Hi-Res   |
      |/Graphics//|                   | Graphics  |
      |//Page 1///|                   |  Page 1X  |
$2000 |///////////|                   |           |
      |___________| _________________ |___________|
      |           |                   |///////////|
      |           |                   |///////////|
      |           |                   |///////////|
      |___________| _________________ |___________|
$0BFF |           |                   |///////////|
      |   Text    |                   |///Text////|
      |  Page 2   |                   |//Page 2X//|
$0800 |           |                   |///////////|
      |___________| _________________ |___________|
$07FF |///////////|                   |           |
      |///Text////|                   |   Text    |
      |//Page 1///|                   |  Page 1X  |
$0400 |///////////|                   |           |
      |___________| _________________ |___________|
      |           |                   |///////////|
      |           |                   |///////////|
      |           |                   |///////////|
      |___________| _________________ |___________|
$01FF |///////////|                   |           |
      |/Stack and/|                   | Alternate |
      |/Zero Page/|                   | Stack and |
      |///////////|                   | Zero Page |
$0000 |///////////|                   |           |
      |___________| _________________ |___________|

 __________________________    ______
| 80STORE      | ON  |     |  |//////|
|______________|_____|_____|  |//////|
| PAGE2        | OFF |     |  |//////|
|______________|_____|_____|  |______|
| HIRES        | ON  |     |   Active
|______________|_____|_____|   Memory
| RAMRD/RAMWRT | ON  |     |
|______________|_____|_____|
```

Figure 6B - Memory Map Three-B

Shortcuts: Writing to Auxiliary Memory from the Keyboard

Press Control-Reset, then type CALL -151 <cr> (to enter the monitor). Now type the following hexadecimal addresses to turn on the double hi-res mode:

```
     C057      (for hi-res)
     C050      (for graphics)
     C053      (for mixed mode)
     C05E      Turns off AN3 for double hi-res
     C00D:0    Turns on the 80COL switch
```

This procedure usually causes the display of a random dot pattern at the top of the screen, while the bottom four lines on the screen contain text. To clear the screen, follow the steps listed below:

1. Type 3D0G <cr> to return to BASIC.
2. Type HGR <cr> to clear half of the screen. (The characters you type will probably appear in alternating columns. This is not a cause for alarm; as noted above, the firmware simply thinks you are working with a 40-column display.) Remember that hi-res graphics commands do not know about the half of the screen stored on page 1X in the auxiliary memory. Therefore, only page 1 (that is, the first half) of the graphics page on the motherboard is cleared. As a result, in the the screen display, only alternate 7-bit columns appear cleared.

   On the other hand, if all of the screen columns were cleared after the HGR command, then chances are good that you are not in double hi-res mode. If your screen was cleared then to determine which mode you are in, type the following instructions:

   ```
   CALL -151
   2000:FF
   2001<2000.2027M
   ```

   If a solid line appears across the top of the screen, you are not in double hi-res mode. (The line that appears should be a dashed or intermittent line: - - - - - across the screen.) If you are not in double hi-res mode, then make sure that you do have a Rev. B motherboard, and that the two Molex-type pins on the extended 80-column card are shorted together with the jumper block. Then re-enter the instructions listed above.

If you are staring at a half-cleared screen, you can clear the non-blank columns by writing zeros to addresses $2000 through $3FFF on graphics page 1X of auxiliary memory. To do so, simply turn on the 80STORE switch, turn on the PAGE2 switch, then write to locations $2000, $2001, $2002, and so on up through $3FFF. However, this procedure will not work if you try it from the monitor. The reason is that each time you invoke a monitor routine, the routine sets the PAGE2 switch back to page 1 so that it can display the most recent command that you entered. When you try to write to $2000, etc. on the auxiliary card, instead it will write to the motherboard memory.

Another way to obtain the desired result is to use the monitor's USER command, which forces a jump to memory location $3F8. You can place a JMP instruction starting at this memory location, so the program will jump to a routine that writes into hi-res page 1X. Fortunately, the system already contains such a routine: AUXMOVE.

Using AUXMOVE

You use the AUXMOVE routine to move data blocks between main and auxiliary memory, but the task still remains of setting up the routine so that it knows which data to write, and where to write it. To use this routine, some byte pairs in the zero page must be setup with the data block addresses, and the carry bit must be fixed to indicate the direction of the move. You may not be surprised to learn that the byte pairs in the zero page used by AUXMOVE are also the scratch-pad registers used by the monitor during instruction execution. The result is that while you type the addresses for the monitor's move command, those addresses are being stored in the byte pairs used by AUXMOVE. Thereafter, you can call the AUXMOVE command directly, using the USER (Control-Y) command.

In practice, then, enter the following instructions:

```
     C00A:0               (turns on the 80-column ROM, which
                          contains the AUXMOVE routine)
     C000:0               (reason explained below)
     3F8: 4C 11 C3        (the jump to AUXMOVE)
     2000<2000.3FFF ^Y    (where ^Y indicates that you should type
                          Control-Y)
```

The syntax for this USER (Control-Y) command is:

```
                    {AUXdest}<{MBstart}.{MBend}^Y
```

The command copies the values in the range MBstart to MBend in the motherboard memory into the auxiliary memory beginning at AUXdest. This command is analogous to the MOVE command.

You can use this procedure to transfer any block of data from the motherboard memory to hi-res page 1X. Working directly from the keyboard, you can use a data block transferred this way to fill in any part of a double hi-res screen image. The image to be stored in hi-res page 1X (i.e., the image that will be displayed in the even-numbered columns of the double hi-res picture) must first be stored in the motherboard memory. You can then use the Control-Y command to transfer the image to hi-res page 1X.

The AUXMOVE routine uses the RAMRD and RAMWRT switches to transfer the data blocks. Because the 80STORE switch overrides the RAMRD and RAMWRT switches, the 80STORE switch must be turned off--otherwise it would keep the transfer from occurring properly (hence the write to $C000 above).

If the 80STORE and HIRES switches are on and PAGE2 is off, when you execute AUXMOVE, any access to an address located within the range from $2000 to $3FFF inclusive would use the motherboard memory, regardless of how RAMRD and RAMWRT are set. Entering the command C000:0 <cr> turns off 80STORE, thus letting the RAMRD and RAMWRT switches control the memory banking.

The Control-Y trick described above only works for transferring data blocks from the main (motherboard) memory to auxiliary memory (because the monitor always enters the AUXMOVE routine with the carry bit set). To move data blocks from the auxiliary memory to the main memory, you must enter AUXMOVE with the carry bit clear. You can use the following routine to transfer data blocks in either direction:

```
     301:AD 0 3      (loads the contents of address $300 into the
                     accumulator)
     304:2A          (rotates the most-significant bit into the carry
                     flag)
     305:4C 11 C3    (jump to $C311 (AUXMOVE))
     3F8:4C 1 3      (sets the Control-Y command to jump to address
                     $301)
```

Before using this routine, you must modify memory location $300, depending on the direction in which you want to transfer the data blocks. If the transfer is from the auxiliary memory to the motherboard, you must clear location $300 to zero. If the transfer is from the motherboard to the auxiliary memory, you must set location $300 to $FF.

Two Double Hi-Res Pages

So far, we have only discussed using graphics pages 1 and 1X to display double hi-res pictures. But--analogous to the standard hi-res pages 1 and 2--two double hi-res pages exist: pages 1 and 1X, at locations $2000 through 3FFF, and pages 2 and 2X, at locations $4000 through 5FFF. The only trick in displaying the second double hi-res page is that you must turn off the 80STORE switch. If the 80STORE switch is on, then only the first page (1 and 1X) is displayed. Go ahead and try it:

```
C000:0    to turn off the 80STORE switch
C055      to turn on the PAGE2 switch
```

The screen will fill with another display of random bits. Clear the screen using the instructions listed above (in the Using AUXMOVE section). However, this time, use addresses $4000 through 5FFF instead. (Don't be alarmed by the fact that the figures you are typing are not displayed on the screen. They are being "displayed" on text page 1.)

```
4000:0
4001<4000.5FFFM
4000<4000.5FFF ^Y
```

You will be delighted to learn that you can also use this trick to display two 80-column text screens. The only problem here is that the 80-column firmware continually turns on the 80STORE switch, which prevents the display of the second 80-column screen. However, if you write your own 80-column display driver, then you can use both of the 80-column screens.

Color Madness

It should come as no surprise that color-display techniques in double hires are different from color-display techniques in standard hi-res. This difference is because the half-dot shift does not exist in double hi-res mode.

Instead of going into a dissertation on how a television set decodes and displays a color signal, I'll simply explain how to generate color in double hi-res mode. In the following examples, the term color monitor refers to either an NTSC monitor or a color television set. Both work; however, the displays will be much harder to see on the color television. The generation of color in double hi-res demands sacrifices. A 560 x 192 dot display is not possible in color. Instead, the horizontal resolution decreases by a factor of four (140 dots across the screen). Just as with a black-and-white monitor, a simple correspondence exists between memory and the pixels on the screen. The difference is that four bits are required to determine each color pixel. These four bits represent 16 different combinations: one for each of the colors available in double hi-res. (These are the same colors that are available in the low-resolution mode.)

Let's start by exploring the pattern that must be stored in memory to draw a single colored line across the screen. Use a color demonstration program (such as COLOR.TEST from older DOS 3.3 System Master disks) to adjust the colors displayed by your monitor. After you have adjusted the colors, exit from the color demonstration program.

The instructions that appear below are divided into groups separated by blank lines. Because it is very difficult (and, on a television set, almost impossible) to read the characters you are typing as they appear on the screen, you will probably make typing errors. If the instructions appear not to work, then start again from the beginning of a group of instructions.

```
     CALL -151            (to get into the monitor routine/program)
     C050                 (This set of instructions puts the computer
     CO57                 into double hi-res model.
     C05E
     C00D:0
     2000:0               (This set of instructions clears first one half
     2001<2000.3FFFM      of the screen, and then the other half of
     3F8: 4C 11 C3        the screen.)
     2000<2000.3FFF^Y

     2100:11 4            (Two red dots appear on top left of
                          screen)
     2102<2100.2126M      (A dashed red line appears across screen)

     2150:8 22            (Two green dots appear near bottom left)
     2152<2150.2175M      (Dashed green line appears across screen)

     2100<2150.2177^Y     (Fills in the red line)
```

In contrast to conditions in standard hi-res, no half-dot shift occurs, and the most-significant bit of each byte is not used.

As noted above, four bits determine a color. You can paint a one-color line across the screen simply by repeating a four-bit pattern across the screen, but it is much easier to write a whole byte rather than just change four bits at a time. Since only seven bits of each byte are displayed (as noted earlier in our discussion of black-and-white double hi-res) and the pattern is four bits wide, it repeats itself every 28 bits or four bytes. Use the instructions listed below to draw a line of any color across the screen by repeating a four byte pattern for the color as shown in Table 2.

```
     2200: main1 main2    (Colored dots appear at the left edge)
     2202<2200.2226M      (A dashed, colored line appears)

     2250: aux1 aux2
     2250<2250.2276M

     2200<2250.2276^Y     (Fills in line, using the selected color)
```

Color	aux1	main1	aux2	main2	Repeated Binary Pattern
Black	00	00	00	00	0000
Magenta	08	11	22	44	0001
Brown	44	08	11	22	0010
Orange	4C	19	33	66	0011
Dark Green	22	44	08	11	0100
Grey1	2A	55	2A	55	0101
Green	66	4C	19	33	0110
Yellow	6E	5D	3B	77	0111
Dark Blue	11	22	44	08	1000
Violet	19	33	66	4C	1001
Grey2	55	2A	55	2A	1010

Pink	5D	3B	77	6E	1011
Medium Blue	33	66	4C	19	1100
Light Blue	3B	77	6E	5D	1101
Aqua	77	6E	5D	3B	1110
White	7F	7F	7F	7F	1111

Table 2-The Sixteen Colors

In Table 2, the heading aux1 indicates the first, fifth, ninth, thirteenth, etc. byte of each line (i.e., every fourth byte, starting with the first byte). The heading main1 indicates the second, sixth, tenth, fourteenth, etc. byte of each line (i.e., every fourth byte, starting with the second byte). The aux2 and main2 headings indicate every fourth byte, starting with the third and fourth bytes of each line, respectively. Aux1 and aux2 are always stored in auxiliary memory, while main1 and main2 are always stored in the motherboard memory.

As you will infer from Table 2, the absolute position of a byte also determines the color displayed. If you write an 8 into the first byte at the far left side of the screen (i.e., in the aux1 column), then a red dot is displayed. But if you write an 8 into the third byte at the left side of the screen (the aux2 column), then a dark green dot is displayed. Remember, the color monitor decides which color to display based on the relative position of the bits on each line (i.e., on how far the bits are from the left edge of the screen).

So far, so good. But suppose you want to display more than one color on a single line. It's easy: just change the four-bit pattern that is stored in memory. For example, if you want the left half of the line to be red, and the right half to be purple, then store the red pattern (8, 11, 22, 44) in the first 40 bytes of the line, then store the purple pattern (19, 33, 66, 4C) in the second 40 bytes of the line. Table 2 is a useful reference tool for switching from one color to another, provided you make the change on a byte boundary. In other words, you must start a new color at the same point in the pattern at which the old color ended. For example, if the old color stops after you write a byte from the main1 column, then you should start the new color by storing the next byte in memory with a byte from the aux2 column. This procedure is illustrated below:

```
2028:11 44 11 44 11 44 11 77 5D 77 5D 77 5D    (creates a dashed line
2128: 8 22 8 22 8 22 8 22 6E 3B 6E 3B 6E       that is red then yellow)

2028<2128.2134^Y                               (fills in the rest of
                                               the colors)
```

Switching Colors in Mid-Byte

If you want a line to change color in the middle of a byte, you will have to recalculate the column, based on the information in Table 2. Suppose you want to divide the screen into three vertical sections, each a different color. The leftmost third of the screen ends in the middle of the 27th character from the left edge-that is, in an aux2 column of the color table. (Dividing 27 by 4 gives a remainder of 3, which indicates the third column, or aux2.) Your pattern should change from the first color to the second color after the 5th bit of the 27th byte. You can change the color in the middle of a byte by selecting the appropriate bytes from the aux2 column of Table 2 and concatenating two bits for the second color with five bits for the first color.

However, because the bits from each byte are shifted out in order from least significant to most significant, the two most significant bits (in this case bits 5 and 6, because bit 7 is unused) for the second color are concatenated with the five least significant bits for the first color. For instance, if you want the color to change from orange (the first color) to green (the second color), then you must append the two most significant bits (5 and 6) of green to the five least significant bits (0-4) of orange. In Table 2, the aux2 column byte for green is 19, and the two most significant bits are both clear. The aux2 column byte for orange is 33, and the five least significant bits are equal to 10011. The new byte calculated from appending green (00) to orange (10011) yields 13 (0010011). Therefore, the first 26 bytes of the line come from the table values for orange; the 27th byte is 13, and the next 26 bytes come from the table values for green.

```
2300: 19 66                      (puts an orange line on the
                                 screen)
2302<2300.2310M
2350: 4C 33
2352<2350.2360M
2300<2350.2360^Y

230D: 33 4C 33 4C 33 4C 33 4C    (puts a green line next to it)
235D: 13 66 19 66 19 66 19 66    (note the first byte)
230D<235D.2363^Y
```

There you have it: a basic explanation of how double hi-res works--except for one or two anomalies. The first anomaly is that NTSC monitors have a limited display range. The second anomaly shows one of the features of double hi-res versus a limitation of standard hi-res.

An NTSC color monitor decides what color to display based on its view of four bit windows in each line, starting from the left edge of the screen. The monitor looks at the first four bits, determines which color is called for, then shifts one bit to the right and determines the color for this new four-bit window. But remember, the color depends not only on the pattern, but also the position of the pattern. To compensate for relative position from the left edge of the screen, the monitor keeps track of where on each line each of these windows start. (For those of you of the technical persuasion, this is done through the use of the color burst signal, which is a 3.58 MHz. clock).

Try this example:

```
2000:0              Clears the screen
2001<2000.3FFFM
2000<2000.3FFF^Y

2001:66             Draws an orange box in the upper left
2401:66
2801:66
2C01:66
3001:66

2050:33             Draws a blue box below and to the right
3402<2050.2050^Y    of the orange box
3802<2050.2050^Y
3C02<2050.2050^Y
```

Notice that if the blue box was drawn at the top of the screen, next to the orange box, they would overlap. Yet, the boxes were drawn on two different columns, orange on main2 and blue on aux1. This can be explained by the previous paragraph, and the sliding windows. The monitor will detect the pattern for orange slightly after the main2 column, while the pattern for blue shows up before column aux1.

The orange pattern is as follows:

```
     0000000 | 0110011 |0000000    look at four-bit windows and you will see
        aux2 |   main2 |   aux1    an orange pattern overlaps on both sides
```

If a pattern is repeated on a line, this overlap does not cause a problem, since the same color just overlaps itself. But watch what happens when a new pattern is started next to a different pattern:

```
     3002<2050.2050^Y    Puts a blue pattern next to the orange one
     2C02<2050.2050^Y
     2802<2050.2050^Y
```

Where the blue overlaps the orange, you will see a white dot. This effect is because one of the four-bit windows the monitor sees is all 1s. If two colors are placed right next to each other, the monitor will sometimes display a third color, or fringe, at the boundary. This fringe effect is especially noticeable when there are a lot of narrow columns of different colors next to each other. (Next time you run COLOR TEST take a look at the boundaries between the colors).

The orange and blue patter is as follows:

```
     0000000 |0110011 | 11001100    note the four 1s in a row at the boundary
        aux2 |  main2 |     aux1    between orange and blue
```

Conclusion

Now you have the tools and the rules to the double hi-res mode. As you can see, double hi-res has more color with higher resolution than standard hi-res. You can even develop games that do fancy animation or scroll orange objects across green backgrounds. You can develop word processing programs which use different fonts or proportional character sets in black and white. Have fun playing with his new mode.

Further Reference

- Apple IIe Technical Reference Manual
- Apple IIc Technical Reference Manual
- Apple IIGS Hardware Reference

END OF FILE TN.AIIe.003

###
FILE: TN.AIIe.004
###

Apple II
Technical Notes

Developer Technical Support

Apple IIe
#4: RDY Line

Revised by: Glenn A. Baxter November 1988
Written by: Peter Baum July 1984

This Technical Note describes an input signal to the 6502 microprocessor called the RDY line.

Using the RDY Line on the Apple IIe and Apple][+

Though the 6502 was one of the first commercially successful microprocessors sold, the designers had foresight to include some very useful functions. Because many early peripherals products were very slow devices, a microprocessor could not read from the device directly. To connect these slow devices onto the Apple peripheral bus so the 6502 can read data from them requires either buffering the device or slowing down the processor. Though most people would try to buffer the device, sometimes it is not feasible. When buffering is not possible, a peripheral device can pull the RDY line to slow down the processor long enough to read a byte. This technique can be used by slow devices to communicate with the 6502.

The RDY line allows a peripheral card to halt the microprocessor during read operations (opcode, operand, or data fetches--reads) with the output address lines reflecting the current address being fetched. If a peripheral device cannot get data on the bus fast enough to meet the setup time of the 6502, then the peripheral card can pull the RDY line low and tell the 6502 to wait. This cannot be done during a 6502 write cycle because the 6502 does not wait during writes.

For the 6502 to read a valid data byte from a peripheral card, the card has about 800 ns from the time the addresses are valid to put the data on the bus. The data must be setup on the bus within approximately 400 ns from the time that the I/O STROBE, I/O SELECT, or DEVICE SELECT signal on the peripheral slot goes true. If a device pulls the RDY line low for one clock cycle, the device will have approximately 1.4 µs, instead of the 400 ns, to put out valid data. The RDY line can be pulled low for more than one cycle--in fact, there is no limit. A device that takes 100 µs to send data can just hold the RDY line low for 100 cycles. Hence, this technique will allow any slower device to get on the bus and send data to the 6502.

This is a bit different than DMA on the Apple IIs. DMA actually prevents the CPU from receiving a clock signal, whereas the RDY line is actually a function of the processor. In Apple II DMA, the 6502 CPU will die after approximately 15 clocks because it depends on the clock to refresh its internal registers.

(The 6502 is dynamic, whereas the 65C02 is static, and therefore not affected by the absence of clock information). In the case of the RDY line, the CPU is internally told to just not complete its bus cycle until RDY is de-asserted. This is a similar concept to DTACK on the Motorola 68000 series CPUs.

The RDY line is typically pulled low during PH1, but the specification sheets for the 6502 show that it can be pulled anytime before the last 200 ns of PH2. The PH2 line is not used by the Apple II and is an unused output from the 6502. It is basically the same as the PH0 line with a little delay. Before I explain when to use (or not use, in some cases) the RDY line, let us first look at some timing diagrams of the Apple system.

Figure 1 shows the relationship between the 6502 and Apple IIe and Apple][+. The timing specifications have been adjusted to reflect the signals as they are seen from the peripheral slots. For example the 6502 (1 MHz) specification guarantees that the address bus will be valid within 225 ns from PH2 out. But the peripheral slots do not see these address lines directly. Instead, the address lines go through a buffer and then out to the peripheral slots. This routing adds a maximum delay of 13 ns in the Apple][and 18 ns in the Apple IIe. The timing diagrams will show, in the case of an Apple][, that the address bus will be valid to the peripheral slots within 238 ns (225+13) of the PH2 falling edge.

The major differences in timing between the Apple][+ and the Apple IIe are due to the processor. The Apple][uses a 1 MHz 6502, while the Apple IIe uses a 6502A, which is a 2 MHz part. This does not mean that the system clock in the Apple IIe runs any faster, only that the 6502A is capable of running faster. This difference results in better timing margins. For example, the address and data buses are set up faster in the Apple IIe by the 6502A than the 6502 sets them up in the Apple][. (This was done because the custom chips in the Apple IIe are slower than the discrete logic in the Apple][, and the 6502A was needed to compensate).

A peripheral card which uses the RDY line can only be used under certain circumstances. Because pulling the RDY line low halts the processor, any program with a software timing loop may not work properly. These programs assume that each instruction will take a fixed amount of time, which is not true when the processor stops in the middle of an instruction. An Apple II Disk is an example of a peripheral which requires timing loops and may not run properly if the RDY line is used.

	Apple][1 MHz 6502		Apple IIe 2 MHz 6502A	
Symbol	Minimum	Maximum	Minimum	Maximum
T02- *	15	50+20 (LS08)	15	50+5 (S02)
T02+ *	30	80+15 (LS08)	30	80+5 (S02)
Tads		225+13 (8T97)		140+18 (LS244)
Trwh	30		30	
Tdevsel-		96 (3 x LS138)		65 (LS154+LS138)
Tiosel-		64 (2 x LS138)		38 (LS138)
Tiostb-		32 (LS138)		15 (LS10)
Tdevsel+		18 (LS138)		30 (LS154)
Tiosel+		36 (2 x LS138)		18 (LS138)
Tiostb+		18 (LS138)		15 (LS10)
Tdsu	100+17 (8T28)**		50+12 (LS245)	
Thr	10		10	
Trs ***	200		200	

(All times are given in nanoseconds (ns).)

```
*      load = 100 pf.
**     The RFI versions of the Apple ][+, revisions A through D
       motherboards, use an 8304 instead an 8T28.
***    The RDY line must never change states within Trs to end of 02.

              Table 1-Timing Specifications for Figure 1

___________                          ________________________
Apple 0o   |________________________|       CPU phase        |_______________


Apple 01    ________________________                          _______________
___________|     Video phase        |________________________|


Q3          _______________          _______________          _______________
___________|               |________|               |________|

           |TO2- |                  |TO2+ |                  |TO2- |

_________________                          ________________________
02 out of 6502*  |________________________|                        |_________

                 |  Tads     |                              Trwh ->|  |

___________________________  __________________________________________  ____
R/W & ADDR                 \/          valid addresses                 \/
___________________________/\__________________________________________/\____
as seen from slots
                                    |Tdevsel- |              |Tdevsel+|

DEVICE SELECT
______________________________________________                          _____
as seen from slots                            |________________________|

                                    |Tiosel-|                |Tiosel+|

I/O SELECT
____________________________________________                          _______
as seen from slots                          |________________________|

                                    |Tiostb-|                |Tiostb+|

I/O STROBE
____________________________________________                          _______
as seen from slots                          |________________________|

                                                      Tdsu ->|      ||<- Thr

DATA                                                         ________
____________________________________________________________/ valid  \_______
from slots                                                  \________/

                                                 |     Trs          |

_______________________________________________  ____________________  ______
RDY                                            \/ don't change state \/
_______________________________________________/\____________________/\______
```

* - 02 is an output signal from the 6502 which is not used by the Apple.
    It is a delayed 0o.

Figure 1 - Timing Signals As Seen From the Peripheral Slots

Table 1 lists three different type of numbers. If a number is by itself, then it is just the corresponding 6502 or 6502A specification. If a number is followed by parenthesis, then it represents the delay, produced by TTL gates, between the 6502 and the peripheral slots. The characters in the parenthesis denote the part number(s) of the part(s) which generated the delay. These parts are typically 74' series TTL except for the 8T28 and 8T97. If there are two numbers in a column with a plus sign (+) then the first number signifies the 6502 specification and the second the TTL delay, with the corresponding part number. Most of the TTL delay times are from the Texas Instrument data books. The 6502 specifications are from the Synertek 6502 data sheet and from Synertek application note AN2 - SY6500.

When the RDY Line Can be Changed and When It Cannot

As can be seen from these figures, the RDY line should not be gated with the PH0 trailing edge since this happens around the same time as the falling edge of PH2. This would violate the TRS specification and probably force the 6502 to perform erratically. Gating the RDY line with the trailing edge of Q3 during PH0 might work, but this could leave as little as 25 ns for the signal to be valid. In other words, Q3 must enable the RDY line low within 25 ns of Q3 changing states. If this output cannot be guaranteed stable, then the RDY line might violate the TRS specification.

The safest time to pull the RDY line is using the PH0 rising edge, but this edge occurs before I/O SELECT, I/O STROBE, or DEVICE SELECT are enabled. Therefore, this scheme will not work if any of these three enables is used by the peripheral card. For example, many peripheral cards use memory mapped I/O to transfer data with the cards registers designed to reside in the DEVICE SELECT memory space. Location $C0n0 (where n = 8 + slot number of peripheral card) might hold the status of the card, and location $C0n1 might be used to read a device such as a disk or an A/D converter. The card uses the DEVICE SELECT signal, pin 41 on the slot, and the four low-order address lines to determine if the 6502 wants to read the status register or read from the A/D converter. Typically, the status register can put its data on the bus within 200 ns, easily meeting the setup requirements of the 6502. But the A/D converter might take at least 100 µs before it can respond with data. The RDY line must be pulled low to allow time for the A/D converter to set up the data bus. Notice that the peripheral card does not know that it should pull the RDY line low until after the DEVICE SELECT signal has gone low. This signal does not go low until after PH0 goes high, so the PH0 rising edge cannot be used to enable the RDY line for this peripheral card.

There are a few ways around this problem. One solution would be to decode the $C0n0 address on the peripheral card and not use DEVICE SELECT. This solution also requires either putting user-selectable switches on the card for setting the slot number, or making the card slot dependent. Another solution is to pull the RDY line low using one of the first three edges, trailing or leading, of the 7 M clock. These edges occur at 70, 140, and 210 ns into PH0 and are trailing, leading, then trailing edges, respectively. The best solution is to use the DEVICE SELECT signal to enable the RDY line. Figure 2 should help.

```
7M
_   _   _   _   _   _   _   _   _   _   _   _   _   _   _   _   _   _
 |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |

0o
    _____________              _____________              _____________
___|             |____________|             |____________|             |____

Q3
    ______        ______        ______        ______        ______        ____
___|      |______|      |______|      |______|      |______|      |______|

DEVICE SELECT
____             _____________                _____________                ___
    |___________|             |_____________|              |_____________|

     write cycle                                    6502 halts with addresses &
     don't pull RDY                                  R/W* line valid here
___________________  _____  __________________________________________________
ADDRESS R/W*       \/     \/ valid
___________________/\_____/\__________________________________________________

RDY
_____________________________________________
     |<----+----->|                          |________________________________
           |                                                |<----+----->|
           |                                                      |
           |______________________________________________________|
               Do NOT change RDY line at these times
```

Figure 2 - Timing Diagram

Do Not Pull RDY During Write Cycles

Because there is no acknowledge response from the 6502, the peripheral card must do some of its own housekeeping and check if a write cycle is taking place. On write cycles, the 6502 will not halt, but continue running until the next read cycle. After a slow peripheral pulls the RDY line and before it tries to get on the bus, it must make sure the 6502 is not in the middle of a write cycle. Otherwise there will be a bus crash, as both the peripheral card and 6502 try to drive the bus. One simple way to prevent this bus crash from occurring is to make sure the peripheral card does not pull the RDY line low during a write cycle. You can guarantee this will not happen by checking the R/W line when PH0 goes high or DEVICE SELECT goes low. The R/W line is guaranteed to be stable by this time.

Releasing the RDY Line

When the RDY line is released, the 6502 will continue the cycle that was originally halted and allow the 6502 to read the data bus. Data will be read on the next trailing edge of PH2 by the 6502, as long as RDY does not change within TRS of the end of PH2. When the peripheral device has set the data bus up with the correct data, it can release the RDY line to complete the read cycle. Releasing the RDY line has exactly the same constraints as pulling the line; do not change RDY within 200 ns of the end of PH2.

The RDY line can be released before data has been set up, if the data will be valid within specification. This means that RDY can be released in the middle of PH1 if the data bus will be valid 117 ns before PH2 trailing edge, for the Apple][(62 ns for the Apple IIe).

Slow Writes

Since the 6502 cannot be halted during write cycles, if a device requires longer than one cycle to receive data then the data must be buffered. Here is an example of how to accomplish this:

```
                             __________
                            |          |
                    _____|\ |     7    |_____|\
          Data bus |       \|     4    |      \ to slow peripheral
                   |_____  /|     L    |_____  /
                         |/ |     S    |     |/
                            |     3    |
                            |     7    |
          _____________     |     4    |
          DEVICE SELECT     |  /\      |
          or                |_/__\_____|
          __________           |    O     ______
          I/O SELECT  _________|    |___  INXFER
          or
          __________
          I/O STROBE
```

Figure 3 - Buffering Data

Note: It is very easy to overrun the slow peripheral using this scheme, since it only buffers one byte at a time. Do not send data twice to the buffer within the maximum allowed time between slow peripheral reads.

Further Reference

- o Apple IIe Technical Reference Manual

END OF FILE TN.AIIe.004

##
FILE: TN.AIIe.005
##

Apple II
Technical Notes

Developer Technical Support

Apple IIe
#5: /INH Line

Revised by: Glenn A. Baxter November 1988
Written by: Peter Baum May 1984

This Technical Note describes how a peripheral card on the Apple IIe should use the inhibit (/INH) line. This information is true for the Apple IIe only.

Using the /INH Line on the Apple IIe

Overview

One of the new features of the Apple IIe is the ability to add more memory or override existing memory from a peripheral card. This feature, which uses the /INH line on the peripheral slots, has been expanded from its original purpose on the Apple][+ of disabling the on-board ROM and allowing the language card (RAM) to reside in the same address space. The Apple IIe allows any part of memory to be replaced by memory on a peripheral card. This Note explains how a peripheral card should use the /INH line.

Uses

Presently, only a few peripheral devices use the /INH line in the IIe for memory expansion. One type of card uses /INH for RAM expansion by switching in extra language cards, while another class of cards uses it to extend the built-in 80-column ROM code by replacing it with their own ROM code. Other cards use /INH so that they can have more than one stack and zero page. Future peripheral cards can take advantage of the /INH line to do even fancier memory expansion, such as keeping multiple programs running in memory at the same time.

More memory, either ROM or RAM, can be added by mapping the memory into the same address space as existing memory. The processor can then select which memory, the on-board or the additional, it wants to use by setting a register (or soft switch). This technique of switching different blocks of memory into the same address space is called bank switching. An example of this technique for extending memory is found in the Apple][+ language card and in the bank switched memory on the IIe.

How It Works

When the /INH line, pin 32 in slots 1-7, is pulled low, all memory on the motherboard and in the auxiliary slot is disabled (including memory on the 80-column and extended 80-column cards). This action allows a peripheral card in slots 1-7 to enable its memory onto the bus.

When the 6502 reads a byte from memory, the data typically comes from one of three places: motherboard ROM, motherboard RAM, or RAM on one of the 80-column cards in the auxiliary slot. When the /INH line is pulled low, all of the above mentioned ROM and RAM is disabled and will not drive the data bus. This disabling allows the peripheral slots to drive the bus by enabling data onto it. The 6502 will then read data from the peripheral card instead of a location on the motherboard or auxiliary slot.

During a 6502 write cycle, if the /INH line is pulled low, then motherboard and auxiliary card RAM are both disabled. A peripheral card can then read a byte off the data bus and store it.

Implementation

Because pulling the /INH line low disables all of memory, the peripheral card must be very careful when it does this. If only a small piece of memory is to be banked into a specific address space, then the /INH line should only be pulled on memory references to that address space. Otherwise the motherboard memory will be disabled and the processor will read or write to the wrong memory and the program will not work properly. For example, if a peripheral card wants to replace the zero page with memory on the card, then the /INH line should be pulled low only on references to the address space between $0 and $FF. If the /INH line is pulled during a processor instruction fetch from the monitor ROM at $F800, the 6502 will read the wrong instruction (or a floating bus) and probably crash the program.

Pulling the /INH line at specific addresses is called select decoding. The hardware on the peripheral card does this by checking the address bus of the 6502, and if the address falls in the correct range, the card pulls the /INH line low. In the earlier example of a new zero page, if the address bus was in the range $0-$FF the card would pull /INH low.

Differences: IIe vs.][+

On the Apple][+, select decoding was not necessarily needed because the /INH line only affected the ROM and not the RAM. If the Apple][+ peripheral card wanted to bank in extra language cards at the addresses $D000-$FFFF, then it could pull the /INH line and keep it low during any memory access. This action would disable the on-board ROM and not any other memory accesses such as zero page or stack. This same card would not work in the IIe, since the next instruction fetch to RAM after pulling /INH low would read a floating bus because all the memory would be disabled.

Another Feature

For those of you who love to muck around in the guts of the Apple IIe, one more feature has been added to the /INH function. The /INH line will also override DMA accesses to memory on the motherboard. This override means that if a peripheral card uses DMA to read or write to memory, another peripheral card could pull the /INH line and process the DMA access. An example of this would be a co-processor card using the memory on a RAM card in another slot. Rather than have the co-processor write to the memory on the motherboard then have the 6502 write to the RAM card, the co-processor can write to an address that the RAM card recognizes. The RAM card could then pull the /INH line and it would be free to read or write the data bus. This technique could also be used by a co-processor to write directly to a printer card in another slot.

Timing

The peripheral card must wait for the address bus to settle, which occurs a maximum of 190 ns after the falling edge of 0o, before pulling the /INH line. (The 6502A maximum address setup time is 140 ns from 02, with a worst case 6502A skew of 50 ns from 0o to 02.) To guarantee that the RAM is disabled and a write does not accidentally take place to the motherboard, the /INH line must be pulled low within 330 ns of 0o.

```
01
            ___________________________                             __________
___________|        Video phase        |___________________________|

0o
___________                             ___________________________
           |___________________________|         CPU phase         |__________

Q3
            _______________    210 ns   _______________             __________
___________|    280 ns     |___________|               |___________|

___________________  ___________________________________________________________
ADDR               \/
___________________/\___________________________________________________________

         | 190 ns   |
                    | 140 ns |
_________           _________   _________________________________
INH      \_________/         \/              valid                \_____________
_________/         \_________/\_________________________________/
              1         2
```

1. The INH line can be pulled high at this time.
2. The INH line can be pulled low (or high) after the addresses are valid at 190 ns, but before 300 ns (from 0o).

Figure 1 - INH Line Timing Signals

Circuits

Figure 2 illustrates a simple example of a circuit that can be used to implement the /INH function.

```
                ________
               |        |
       A15 ---|        |
               | Select |
       A14 ---| Decode |
               | Logic  |                    _______
       __/\/--|        |                   |       |
               |        |------------|D  7  Q|--- INH* J[1-7]32
       A0  ---|        |  PULLINH*  |   4   |
               |________|            |   L   |
                                     |   S   |
```

```
            |\                    |   3   |
    Q3  ----| >o------------------|>  7   |
            |/                    |_  9   |
    0o  --------------------------|G      |
                                  |_______|
```

Figure 2 - Circuit Implementing /INH Function

An Application

The circuit in Figure 3 can be used to replace the code in the monitor ROM, from location $FC00 to $FFFF, with custom code. Anytime the address space between $FC00-$FFFF is accessed, the /INH line is pulled low, the motherboard memory is disabled, and the circuit's 1K RAM is enabled instead. Part of this feature can be disabled and the motherboard memory can be read by keeping the switch connected to +5 volts (READDIS). Whenever the system writes to any location in the address space $FC00-$FFFF, the circuit will disable any RAM on the motherboard and instead write into the 1K RAM.

Here is a series of commands that can be used with the circuit to replace the reset vector at $FFFC and $FFFD. A new reset routine can be written that will print the screen or save the status of all the registers whenever the Reset key is pressed.

Start the system with the circuit's switch connected to +5 (READDIS). Doing so will enable the system to read the monitor ROM during startup, before the 1K RAM has been initialized.

Get into the monitor by typing CALL -151. The system prompt should now be an asterisk (*). Copy the monitor ROM into the 1K RAM with the command FC00<FC00.FFFFM. Change the reset vector so that it jumps to location $300 with this command, FFFC:0, then copy your new reset routine into memory starting at location $300. Now, set the switch to ground (READEN) so all future read accesses to $FC00-$FFFF will read the 1K RAM.

For example, if these instructions are stored in memory starting at location $300, then the system will clear the screen and continue execution in the monitor when the Reset key is pressed.

```
    $300:20 58 FC    JSR HOME   (clears screen)
    $303:4C 65 FF    JMP to MON (resume execution in monitor)
```

One of the problems with this circuit is that it also overrides any accesses to the language card, therefore any program that uses the language card will not work with it. The circuit does not keep track of which memory is enabled, ROM or language card RAM, in the $FC00-$FFFF space.

```
Q3  --------------------------------------------------------+
       _____                                               _|_
A15 ---|     \                                             \ / <--00
A14 ---| 11   )--+                                          O
A13 ---|_____/   |   _____                                  |
                 +--|     \                ______       ____|___
A12 ----------------| 11   )-----------------|      \     |  \/   _|    ___
A11 ----------------|_____/  +---------------| 11    )---|D      Q|--- INH
                             |            +--|______/    |        |
A10 -------------------------+   _____    |              |        |
R/W -----+---+-----------------|     \    |              |        |
```

```
        _|_  |                 | 00   )O--+             |  379  |
00  --> \ /  |              +--|_____/                  |       |
         O   |              |                           |       |
        _|___|_____         |            _____          |_      |
A9  ---| __        |        |           /     |--+------|G     Q|---+
A8  ---| OE  WE  CE|------------------O(   00 |  |      |_______|   |
A7  ---|           |--- D7  |           \_____|-----+               |
A6  ---|   2159    |--- D6  |                    |  +---------------+
A5  ---|  1K RAM   |--- D5  |                    |
A4  ---|           |--- D4  |                    |    ^ +5V
A3  ---|           |--- D3  |                    |    |
A2  ---|           |--- D2  |                    |    o READDIS
A1  ---|           |--- D1  |                    |   /
A0  ---|___________|--- D0  +----------------------o/
                                                 |
PH0 ---------------------------------------------+    o READDEN
                                                    __|__
                                                     ---
                                                      -
```

Figure 3 - Circuit to Replace Monitor ROM Code

Further Reference
- o Apple IIe Technical Reference Manual

END OF FILE TN.AIIe.005

```
#######################################################################
### FILE: TN.AIIe.006
#######################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIe
#6: The Apple II Paddle Circuits

Revised by: Glenn A. Baxter — November 1988
Written by: Peter Baum — May 1984

This Technical Note describes the paddle circuit used in the Apple II family of computers.

Caveats

Since Apple has introduced machines with internal clock speeds which may not be exactly 1.023 MHz, it is best to use the PREAD firmware call to read paddle data. This Note assumes that the clock speed of the system is exactly 1.023 MHz. If you want to insure accuracy in reading paddle data, you should make sure the system is first running at the correct speed. Enough information is provided so that you can write your own PREAD routine, although this is discouraged. If the program runs on an Apple IIGS or some future machine, your custom paddle reading routine will fail to give the correct results.

Circuit Description

The value of the Apple paddles (or joystick) is determined by a software timing loop reading a change of state in a timing circuit. The paddles consist of a variable resistor (from 0-150k ohms) which makes up part of the timing circuit. There is a routine in the monitor ROM, called PREAD, which counts the time until a state change occurs in the paddle circuit. This time is translated into a value between 0 and 255.

The block diagrams in Figures 1 and 2 show the paddle circuit for the Apple][+, Apple IIc, and the Apple IIe. The large block on the left illustrates part of the circuitry inside the 558 timer chip. The 558 chip consists of four of these blocks, with all four paddle triggers lines shorted together on the motherboard and activated by the soft switch at $C070. The outputs of the 558 chip run into a multiplexer, which places the appropriate signal onto the high bit of the data bus when a paddle soft switch address in the range $C064 $C067 is read. The Apple IIc uses a 556 timer rather than the 558 chip and only supports two paddles, 0 and 1.

The 100 ohm resistor and .022 microfarad capacitor are on the motherboard, with the variable resistor in the paddle. Each of the four paddle inputs have their own capacitor and resistor. Since these components can vary by as much as five percent from Apple to Apple, this circuit is not a very exact analog to digital converter. If a paddle is moved from one Apple to another without changing the resistance (turning the knob), the paddle read routine will probably calculate a different value for each machine. About the only feature of the paddle read routine that a programmer can depend on is that the value returned will rise if the paddle resistance increases (or fall if the resistance decreases).

The paddle timing circuit on the Apple][+ and Apple IIc is slightly different than the one on the Apple IIe. On the Apple IIe, the 100 ohm fixed resistor is between the transistor and the capacitor, while the variable resistor in the paddle is connected directly to the capacitor. On the Apple][+ and IIc, the capacitor is connected directly to the transistor and the fixed resistor is in series with paddle resistor.

```
 _______________________________________________
|                   556 Timer                   |
|    ^ VCC                                      |               _____________
|    |                                          |              |             |
|    \                                          |              |   Paddle    |
|    /                                          |              |      ^      |
| 5K \                                          |              |      |  ^   |
|    /    +-----------------------------------------O---+      |      \ /    |
|    \    |                                     |       |      |      //     |
|    |    |            ___________              |       |      |      X      |
|    +------- + |\    |           |             |       |      |     //      |
|    |    |     | >---|RESET      |             |       |      |    / \      |
|    \    +-- - |/    |           |             |       |      |______|______|
|    /                |           |             |       |             |
| 5K \  Comparator    |           |             |       |    100 ohm  |
|    /                | FLIP FLOP |         +-------O---+--+-\/\/\/\--+
|    \                |           |         |   |          |
|    |                |           |        /    |          |
|    +------- + |\    |          _|      |/     |         _|_
|    |          | >---|SET       Q|---+--|\     |         ___ .022uF
|    \    +-- - |/    |___________|   |  | \>   |          |
|    /    |                           |     |   |          |
| 5K \    |                           |     +---|---O------+
|    /    |                           |   __|__ |
|    \    |                           |    ---  |
|  __|__  |                           |     -   |
|   ---   |                           |  |\     |
|    -    |                           +--| >O---|---O Output
|         |                              |/     |     ($C06x)
|_________|_____________________________________|
          |
          O Trigger
            ($C070)
```

Figure 1 - Apple][+ and IIc Paddle Circuit

An Example of a Typical Paddle Read Routine

The timing circuit works by discharging a capacitor through a transistor, then shutting the transistor off and letting the paddle charge the capacitor by supplying current through the variable resistor. The rate at which the capacitor charges is a function of the variable resistance; the lower the paddle resistance, the greater the current and the faster the capacitor charges. When the capacitor reaches a predetermined value it changes the state of a flip flop. The paddle read routine counts the time it takes for the capacitor to rise and change the flip flop.

Let's step through an example of a typical paddle read operation. For now we will assume the capacitor has already been discharged and in a few pages I will explain when this assumption can be made and when it cannot.

The software starts by reading the soft switch at location $C070, which strobes the trigger lines on the 558 timer. This action causes two events to occur, the output signal (which is read at $C064-$C067 for paddle 0-3, respectively) goes high and the transistor turns off.

The software, after initially strobing the trigger line, executes a timing loop which reads the state of the output signal. When the output signal changes from high to low the software jumps out of the timing loop and returns a value indicating the time. The monitor PREAD routine consists of a 11 µsec. loop and will return a value between 0 and 255. (Note: The firmware listing is wrong and says the loop is 12 µsec.) The timing loop returns 255 if the circuit takes longer than 2.82 ms for the state change to occur.

```
 _______________________________________________
|                   558 Timer                   |
|    ^ VCC                                      |               _____________
|    |                                          |              |             |
|    \                                          |              |   Paddle    |
|    /                                          |              |      ^      |
| 5K \                                          |              |      |  ^   |
|    /    +---------------------------------+   |              |      \ /    |
|    \    |                                 |   |              |      //     |
|    |    |            ___________          |   |              |      X      |
|    +------- + |\     |           |        |   |              |     //      |
|    |    |     | >---|RESET      |         |   |              |    / \      |
|    \    +-- - |/     |           |        |   |              |______|______|
|    /                 |           |        |   |                     |
| 5K \  Comparator     |           |        |   |      100 ohm        |
|    /                 | FLIP FLOP |        +-------O--\/\/\/\--+-----+
|    \                 |           |        |   |               |
|    |                 |           |       /    |               |
|    +------- + |\     |          _|      |/    |              _|_
|    |          | >---|SET       Q|---+--|\    |              ___ .022uF
|    \    +-- - |/     |___________|   |  | \>  |               |
|    /    |                            |     |  |               |
| 5K \    |                            |     +---|---O-----------+
|    /    |                            |   __|__ |
|    \    |                            |    ---  |
|  __|__  |                            |     -   |
|   ---   |                            |  |\     |
|    -    |                            +--| >O---|---O Output
|         |                               |/     |     ($C06x)
|_________|______________________________________|
          |
          O Trigger
            ($C070)
```

Figure 2 - Apple IIe Paddle Circuit

```
* PADDLE READ ROUTINE
```

```
* ENTER WITH PADDLE NUMBER (0-3) IN X-REG

FB1E:AD 70 C0  PREAD   4  LDA  PTRIG     ;TRIGGER PADDLES
FB21:A0 00             2  LDY  #0        ;INIT COUNTER
FB23:EA                2  NOP            ;COMPENSATE FOR 1ST COUNT
FB24:EA                2  NOP
FB25:BD 64 C0  PREAD2  4  LDA  PADDL0,X  ;COUNT EVERY 11 µSEC.
FB28:10 04             2  BPL  RTS2D     ;BRANCH WHEN TIMED OUT
FB2A:C8                2  INY            ;INCREMENT COUNTER
FB2B:D0 F8             3  BNE  PREAD2    ;CONTINUE COUNTING
FB2D:88                   DEY            ;COUNTER OVERFLOWED
FB2E:60        RTS2D      RTS            ;RETURN W/VALUE 0-255
```

Inside the 558 timer chip, when the trigger is strobed low, the comparator that feeds the set input of the flip flop is triggered, which in turn sets the output of the 558 timer. At the same time, the transistor, which has held the capacitor near ground by sinking current from it, is shut off. The capacitor can now charge using the current supplied by the paddle. The smaller the paddle's resistance, the more current the paddle will supply and the faster the capacitor charges. After some time, the capacitor will charge to the threshold value of 3.3 volts, which is set by the voltage divider network in the 558 timer, and the comparator that feeds the reset input on the flip flop will trigger. This trigger sets the output signal ($C06x) of the 558 timer low, which indicates to the software that the circuit has timed out.

```
              _____       ___________________________________________________
Trigger $C070      |      |
                   |______|

                             ________________________________
                           _|                                |<------ Threshold
Feedback to              _|                                  |_
Reset Comp _____________|                                      |_______________

                         ____________________________________
                        |                                    |
Output _________________|                                    |_________________
                                    Timing Value
                        |<---------------------------------->|
                                0 - 2.82 milliseconds
```

Figure 3 - Paddle Circuit Recharge Timing

Resetting the flip flop turns the transistor on, which discharges the capacitor very quickly (normally less than 250 ns). That paddle can then be read again.

A Closer Look at the Hardware

The First Anomaly

Notice that the last sentence states that the paddle can be read again and not the paddles. If another paddle is read immediately after the first, it may yield the wrong value. To demonstrate this, I will step through an example of reading a second paddle immediately after finishing the first.

In this example I will assume that the first paddle has been set with a very low resistance, while the second paddle has a high resistance. The first paddle will time out very quickly and return with a small value, while the second paddle will take longer and yield a larger value.

We start reading the paddles by testing the paddle outputs to see if they are low, which indicates that the capacitor has been discharged. Assuming that the outputs are low, the next step is to trigger the 558 timer ($C070), which turns off the transistor and allows the capacitors to charge. Since all of the trigger input lines are shorted together, all four of the capacitors will charge, but at different rates since the paddle resistances have been set to different values. The voltage on the capacitor for the first paddle will reach the threshold voltage very quickly since the paddle resistance has been set low, therefore the timing loop will time out quickly.

At this point the capacitor for the second paddle is still charging and has not yet reached the threshold since the paddle resistance was set to a high value. The transistor for the second paddle is still turned off due to the initial trigger used for reading paddle one. This means that the capacitor for the second paddle has not been discharged.

Any attempts at reading the second paddle now will only yield false results. The capacitor is partly charged and therefore will reach the threshold value much faster than if the capacitor had been completely discharged. If the timing loop is used, it will return with a smaller value than it would if the capacitor had been completely discharged. Notice that retriggering (reading location $C070) the 558 timer will not help, since that only keeps the transistor turned off and does not help discharge the capacitor. The only way for the capacitor to discharge is to let the circuit time out completely by letting the capacitor charge until it resets the flip flop.

To read the second paddle, the capacitor must first be discharged, which is only done when the threshold value is reached and the 558 timer flip flop is reset. The only way to guarantee that the capacitor is discharged is if the transistor is on. This condition is met when the paddle output is low. Therefore, start every paddle read either by waiting for at least 3 ms before strobing the trigger input or testing to make sure that the paddle output is low.

If after 4 ms the paddle output is not low, then there is a good chance that there is no paddle connected. This result may also indicate that a peripheral with a larger maximum value resistor than the 150k ohms used by the Apple paddles is attached. Some peripheral devices use this technique of a larger variable resistor so that more than 256 points of resolution can be determined. Of course, this requires a custom software driver and the monitor PREAD routine cannot be used.

Apple IIe Anomalies

The problem with Apple IIe paddle input is that the capacitor may not be discharged by the transistor. Typically, the transistor will discharge the capacitor in less than 250 ns on the Apple][+. But on the Apple IIe, if the paddle resistance is very low then the paddle may supply enough current to always keep the capacitor charged.

Because the fixed resistor (100 ohms) on the Apple IIe motherboard is between the capacitor and the transistor, there will be a voltage drop across the resistor if the capacitor stays charged. When the transistor is shut off by the trigger strobe, this voltage drop will disappear and the capacitor, which may be near the threshold voltage, will trigger the reset comparator earlier than it would if the capacitor had been discharged completely. The net affect of this is that the paddles will read zero on the Apple IIe when they would read a small value on the Apple][+ or IIc.

Other circuits which expect the capacitor to discharge completely may not work properly. A circuit which attempts to simulate a paddle through active components such as a digital to analog converter may be able to source enough current that the capacitor never discharges and the paddle always reads zero.

It should also be noted that due to electromagnetic interference, later model IIe computers actually have an extra capacitor attached between the BUTTON inputs and ground. This essentially slows the response time of the input, making a fully digital input appear a bit more analog (no pun intended). Care should be taken in designing system which depend on a certain repetition rate of the button inputs. Careful engineering and testing across systems should prevent any problems. As an example, adding a transistor output stage to drive the button inputs to the appropriate states might be a good idea for a serializing A/D. A joystick would not require this kind of circuit because the user input is too slow to be affected by the capacitors. For more information on the changes in later model IIe computers, refer to Apple IIe Technical Note #9, Switch Input Changes.

Conclusion

Hopefully, this Note has given the reader a good feel for the paddle circuitry and the routines which determine the paddle values. To reinforce the material covered, you should try writing your own paddle read routine. For example, you could write a read routine that would read two paddles at once. The software loop will not have the 11 µsec. resolution of the PREAD routine, but you will find it still works just fine. Happy programming.

Further Reference

- Apple IIe Technical Reference Manual

END OF FILE TN.AIIe.006

##
FILE: TN.AIIe.007
##

Apple II
Technical Notes

Developer Technical Support

Apple IIe
#7: Interfaces--Serial, Parallel, and IEEE-488

Revised by: Matt Deatherage November 1988
Written by: Peter Baum April 1984

This Technical Note describes the pin configurations of three difference interface types offered on the Apple II family of computers.

Serial

Currently, Apple sells a card, called the Super Serial Card (SSC), that can be used to connect an Apple printer to an Apple II. The SSC replaces both the Communications Card and the Hi-Speed Serial Card. The SSC supports the firmware (Pascal 1.1) protocol except for the optional control and interrupt handling routines.

The SSC has a 10-pin header on it, but comes with a cable which connects the header to a female DB-25 connector. The SSC can be configured as either a modem (DCE) or as a terminal (DTE) using a jumper block (in the latter case the jumper block acts as a modem eliminator). Though the pin configuration of the DB-25 connector is well defined, there is no standard use of the handshake signals. Different printers will use the handshake lines for different functions. Table 1 shows the pin configuration for the DB-25 on the SSC. Consult your printer manual for more specific information on which signals are used.

10-pin Header	Signal Name		Female DB-25 Terminal	Female DB-25 Modem
1	Frame Ground	(FRMGND)	1	1
2	Transmit Data	(TxD)	3	2
3	Receive Data	(RxD)	2	3
4	Request To Send	(RTS)	8	4
5	Clear To Send	(CTS)	8	5
6	Data Set Ready	(DSR)	20	6
8	Signal Ground	(SGLGND)	7	7
10	Data Carrier Detect	(DCD)	4,5	*8
7	Secondary Clear to Send	(SCTS)	19	**19
9	Data Terminal Ready	(DTR)	6	20

* Only if SW1-7 is closed (on) with SSC.
** Only if SW2-7 is closed (on) with SSC.

Table 1-Pin Configuration for SSC DB-25 Connector

Parallel

Apple formerly shipped a parallel card, called the Parallel Interface Card (PIC), which can be used to connect a parallel printer to an Apple II. The PIC replaced the Parallel Printer Interface Card and the Centronics Interface Card. The PIC does not support the firmware protocol, so Pascal identifies the card as a printer card (described in Pascal protocols).

Most commonly used printers operate properly if the switches on the PIC are set as in Figure 2.

	1	2	3	4	5	6	7
on				x	x		
off	x	x	x			x	x

Figure 2-PIC Switch Configuration

This setting prepares the parallel interface to transfer data using a 1 microsecond strobe pulse of negative polarity when sending data, while receiving a negative acknowledge signal, with interrupts disabled.

The PIC has a 26-pin header, but it comes with a cable which connects the header to a female DB-25. The Parallel Printer Card and the Centronics Card used a 20-pin header. Most parallel printers (90%) use a "microribbon 36" as the connector. The pin configuration varies from printer to printer, but Table 2 covers most printers (Apple DMP, Epson). For other printers, refer to page 7 of the Parallel Interface Card Manual.

PIC Function	Printer Function	26-Pin	DB-25	36-Pin	20-Pin
Ground	Ground	3	2	19	1
Ground	Ground	22	24	16	20
Ground	Ground	7	4		
Ground	Ground	14	20		
ACK	Acknowledge	6	16	10	2
Strobe	Strobe	4	15	1	8
DO 0	Data 1	9	5	2	10
DO 1	Data 2	11	6	3	11
DO 2	Data 3	15	8	4	12
DO 3	Data 4	18	22	5	13
DO 4	Data 5	20	23	6	14
DO 5	Data 6	21	11	7	15
DO 6	Data 7	23	12	8	16
DO 7	Data 8 *	25	13	9	17
DI 3	Fault	24	25	32	6
DI 4	Busy	2	14	11	7
DI 5	Paper out	12	19	12	9
DI 6	Select	16	21	13	8
DI 7	Enable	10	18	35	19
	**		7		

* This may be assigned a "hard" value for some printers to distinguish between graphics and normal character sets.

** Pin 7 is blocked on the female DB-25 connector and omitted on the mail DB-25 connector to prevent the insertion of serial connectors into parallel ports.

IEEE-488

The IEEE-488 bus standard is a well defined eight-bit parallel, byte serial, asynchronous data transfer interface. The standard has been thoroughly documented with the most complete description available from the Institute of Electrical and Electronic Engineers (IEEE) in New York. Standard cables are manufactured by many companies and usually advertised as either IEEE-488, General Purpose Interface Bus (GPIB), or Hewlett-Packard Interface Bus (HPIB) cables.

IEEE-488 cards do not support Apple firmware protocols, so an assembly language driver must be used to access the cards from high level languages (see Appendix F of the IEEE-488 Interface User's Guide).

Further Reference

- Apple IIe Technical Reference Manual
- Parallel Interface Card Manual
- IEEE-488 Card Manual

END OF FILE TN.AIIe.007

###
FILE: TN.AIIe.008
###

Apple II
Technical Notes

Developer Technical Support

Apple IIe
#8: Known Anomalies of Enhanced IIe ROMs

Revised by: Matt Deatherage November 1988
Revised by: Cameron Birse February 1986

This Technical Note describes three problems with the Enhanced IIe ROMs and some suggested solutions.

The following three anomalies are known to occur when the Enhanced IIe ROMs are present:

1. Some Apple II peripheral cards do not handle interrupts well since Apple II family members before the IIc and Enhanced IIe did not handle them very well either. If a card that cannot handle interrupts is used on the Enhanced Apple IIe, any interrupt is very likely to crash the system. A common example of this would be older, non-interruptible printer cards used with a Mouse card in the system. You can often work around this problem by disabling interrupts before printing to such a printer card.

2. There may be some problems when using the ROMs with communications packages. These problems are due to the way the 80-column firmware switches into 40-column mode. By sending a Control-Q through COUT, the firmware switches into 40-column mode. A simple solution to this would be to send an Escape-Control-D sequence, which disables the control functions. This solution will remain in effect until either the 80-column card is re-initialized by PR#3 or an Escape-Control-E sequence is sent through COUT. Another solution would be to simply not allow Control-Q sequences to get through to COUT by filtering them before they get there.

3. Many developers using double high-resolution graphics may wish to use 40-column text displays so the text can be read on a television set. There are a couple of possibilities:

   A. You can define your own double high-resolution character set with any size characters you desire, then plot them on the double high-resolution screen.

   B. You can print text to the Apple IIe text screen and toggle the screen on to display it.

      Note: There is no way to display four lines of 40-column text at the bottom of the double high-resolution screen in mixed mode since the 80-column hardware must be active while double high-resolution graphics are being displayed.

To use the second method, however, does require some special considerations.

The Apple IIe scroll routine continues to use the window parameters when scrolling, but uses the 80COL soft switch to determine if it should scroll the 80- or 40-column screen. Since the firmware has initialized a 40-column window, the scroll routines will move only the first 40 columns. But, the 80COL flag has been turned on for double high-resolution, therefore, the scrolling routine takes every even column from auxiliary memory and every odd column from main memory. As a result, only the first 40 columns get scrolled, 20 columns from auxiliary memory and 20 columns from main memory.

One possible solution to the problem is to write your own scroll routines. Another might be to write to the screen so that scrolling will not occur. But there is yet another solution. Turn on the full 80-column mode with a PR#3 or the equivalent. Now print your text to COUT in the normal manner, being careful not to exceed 40 characters per line. The 80 column firmware will scroll everything properly. When you are ready to display text, send a Control-Q sequence through COUT to switch to 40 columns. When you are ready to return to double high-resolution mode, send a Control-R sequence to COUT.

When switching modes, a momentary glitch may occur. If you send the Control-Q sequence to COUT while still in graphics mode, the screen will go to regular single high-resolution mode before finally going to text mode. If you switch to text mode first, the text will be in 80-column mode (with 40 columns displayed on the left half of the screen) before ultimately going to 40-column mode. The same potential glitch may occur going back to double high-resolution mode. The glitch will be only momentary and may not present any problem for your application.

Further Reference

- o Apple IIe Technical Reference Manual

END OF FILE TN.AIIe.008

##
FILE: TN.AIIe.009
##

Apple II
Technical Notes

Developer Technical Support

Apple IIe
#9: Switch Input Changes

Revised by: Glenn A. Baxter November 1988
Written by: Earl Edwards May 1988

This Technical Note describes three changes which have been made to the switch circuitry of Apple IIe revision C and later motherboards.

The latest Apple IIe logic board has some changes in its switch circuitry. Logic boards with part numbers 820-0087-C and later differ from earlier boards as follows:

- SW2 has been connected to the Shift keys on the keyboard by closing the X6 jumper.
- 12k ohm pullup resistors have been added to SW0 and SW1.
- A 0.1 microfarad capacitor to ground has been added to all three switch inputs: SW0 (PB0, Open-Apple, OAPL), SW1 (PB1, Option, Closed-Apple, CAPL), and SW2 (PB2).

Note: Schematics showing the differences are available in Chapter 7 of the Apple IIe Technical Reference, First Printing, January 1987.

The X6 jumper was closed to allow the Shift key to be read directly, facilitating the shift-click mouse selection feature in software products. Note that this change connects SW2 to +5V through a 1k ohm resistor, and when a shift key is depressed, SW2 is at ground potential.

The 12k ohm resistors were added to ensure that the self-diagnostic test would run when the keyboard is disconnected. The resistors have negligible influence when the keyboard is connected.

The capacitors were added to reduce radiated emissions. This reduction was required because of changes in the memory configuration. As a result of the addition, the functional bandwidth of the inputs has been reduced; however, the input requirements of the 74LS251 have not changed. This addition may cause improper operation with peripheral devices that rely on high push button repetition rates.

The minimum V(IH) to the 74LS251 remains 2.0V, but for improved noise margin, a minimum V(IH) of 2.4V is recommended. This requires a drive of about 6 ma to overcome the 470-ohm 5 percent resistor on SW0 and SW1.

The maximum V(IL) is 0.8V, and here again you should allow for some noise margin. The low level is ensured by the 470-ohm keyboard pulldown resistor alone, but additional current sink will speed up the transition time.

Further Reference
- o Apple IIe Technical Reference Manual

END OF FILE TN.AIIe.009

```
#####################################################################
### FILE: TN.ATLK.001
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

AppleTalk
#1: Identifying AppleTalk

Revised by: Dan Strnad July 1989
Written by: Dan Strnad November 1988

This Technical Note describes the correct methods for identifying AppleTalk under ProDOS 8 and GS/OS, as the ATLK ROM signature is no longer used. Changes since November 1988: Updated for System Software 5.0 and added references for determining if an application has been launched over the network and identifying AppleTalk and its associated slot.

To determine if an application has been launched over the network, refer to the NetLaunch code fragment found in the AppleShare Programmer's Guide for the Apple IIGS.

Under ProDOS, to identify both AppleTalk and the slot with which it is associated for printing, refer to Apple II AppleTalk Technical Note #4, Printing Through the Firmware.

To identify AppleTalk under ProDOS 8:

1. Issue an AppleShare GetInfo call.
2. If there is no error result, AppleTalk is installed.

```
InfoParams    DB $00             ;Synchronous only
              DB $02             ;GetInfo call number
InfoResult    DS 13              ;<- results returned here

CheckATalk    JSR $BF00
              DB $42             ;$42 command # for AppleTalk calls
              DW InfoParams      ;Parameter list address
              BCS NoATalk        ;handle the error
IsATalk.      ...                ;AppleTalk installed when here

NoATalk       ...                ;AppleTalk not installed when here
```

To identify AppleTalk protocols and AppleShare file system under System Software 5.0:

1. Set up the parameter block for a GS/OS GetFSTInfo call using fstNum = 1.
2. Issue the GetFSTInfo call.
3. If the fileSysID is $0D the AppleShare FST and AppleShare are present.
4. If a parameter out of range error ($53) results, the AppleShare

file system is not present.
5. Otherwise, if steps 3 and 4 are inconclusive, increment the fstNum and loop back to step 2.

To identify AppleTalk protocols, including LAP through PFI but excluding the file system, under System Software 5.0:

1. Set up the parameter block for a GS/OS DInfo call using device number one.
2. Issue the DInfo call.
3. If the deviceID is $1D, the AppleTalk main driver and AppleTalk are present.
4. If a parameter out of range error ($53) results, the AppleTalk protocols are not present.
5. Otherwise, if steps 3 and 4 are inconclusive, increment the device number and loop back to step 2.

To identify AppleTalk protocols, including LAP through ASP but excluding the file system, under System Software 4.0:

1. Issue an an SPGetStatus call
2. If the call returns without error, AppleTalk is present.

Note: With the release of System Software 5.0, earlier versions are not supported.

Further Reference

- Inside AppleTalk
- AppleShare Programmer's Guide for the Apple IIGS
- GS/OS Reference
- Apple II AppleTalk Technical Note #4, Printing Through the Firmware

END OF FILE TN.ATLK.001

###
FILE: TN.ATLK.002
###

Apple II
Technical Notes

Developer Technical Support

AppleTalk
#2: ProDOS 8 Compatibility on the IIe and IIGS

Written by: Mark Day November 1988

This Technical Note describes areas which could cause an application to run under the AppleShare Apple IIe workstation software, but fail under the Apple IIGS workstation software.

- o If code is running in auxiliary memory in emulation mode (e.g., ProDOS 8 programs that run code from auxiliary memory), make sure $0100 in auxiliary memory is set to the normal stack pointer and $0101 in auxiliary memory is set to the auxiliary (alternate) stack pointer. (See page 93 of the Apple IIe Technical Reference Manual.)
- o Make sure ProDOS calls are not made from auxiliary memory; Apple has never recommended doing this, and it is not supported.
- o Make sure interrupts are enabled when making ProDOS 8 calls.
- o Make sure interrupts are not disabled for long periods of time, nor for a high percentage of time. Whenever interrupts are disabled, there is a chance that an AppleTalk packet will be missed (which could cause AppleShare volumes to be unmounted). The more interrupts are disabled, the more likely that packets will be missed. This risk is inherent for any application that disables interrupts (directly or indirectly), therefore, interrupts should be disabled with discretion and only when absolutely necessary.
- o Make sure programs get the completion routine return address from the GetInfo call when they are started.
- o Make sure to identify AppleTalk by calling GetInfo and checking for an invalid call number error (which means AppleTalk is not present). Do not use the ATLK signature bytes for identification. See Apple II AppleTalk Technical Note #1, Identifying AppleTalk.

Further Reference
- o Apple IIe Technical Reference Manual
- o Apple II AppleTalk Technical Note #1, Identifying AppleTalk

END OF FILE TN.ATLK.002

###
FILE: TN.ATLK.003
###

Apple II
Technical Notes

Developer Technical Support

AppleTalk
#3: Avoiding Remote Printer Time-Outs

Revised by: Jim Luther September 1989
Written by: Jim Luther May 1989

This Technical Note discusses how to avoid time-outs when printing to remote printers.
Changes since May 1989: Updated to reflect System Software 5.0 changes and to clarify the results of changing the time-out interval.

The Apple II AppleTalk firmware's Remote Print Manager (RPM), which supports AppleTalk's Super Serial Card (SSC) entry points, maintains a time-out interval value. The time-out interval is usually set to 30 seconds. When an application quits writing to the AppleTalk firmware, the RPM waits this time interval before sending the last block of data to the printer and closing the Printer Access Protocol (PAP) connection.

What does this mean? If an application waits longer than the time-out interval (e.g., 30 seconds) between any write accesses to the AppleTalk firmware (i.e., a pause between initialization and printing or a pause during printing), the PAP connection closes, the current page may be ejected from the printer (this is printer dependent--the ImageWriter II and ImageWriter LQ do not automatically eject the page, the Apple LaserWriter does), and the rest of the application's output to the printer is lost. If you initialize the AppleTalk SSC firmware, you must print immediately or a time-out may occur and reinitialization is necessary to print again. Applications should not initialize the firmware and expect it still to be initialized at a later point in time.

What You Can Do

The RPM's PMSetPrinter call may be used to change the time-out interval to a different value. However, the time-out interval should be kept as short as possible because other users cannot open another PAP connection with the printer until your machine has timed-out. In other words, if you set the time-out interval for five minutes, the RPM keeps the PAP connection open with the printer for five minutes after the last character is written to the RPM, thus blocking other machines from using that printer for five extra minutes; this delay is unacceptable in a shared printer environment.

With an Apple IIGS using System Software 5.0, the RPM's PMSetPrinter call may be used to set the time-out interval to zero. When the time-out interval is set to zero, the session never times out and must be closed with the Apple IIGS-specific PMCloseSession RPM call.

Further Reference

- o AppleShare Programmer's Guide for the Apple IIGS

END OF FILE TN.ATLK.003

###
FILE: TN.ATLK.004
###

Apple II
Technical Notes

Developer Technical Support

AppleTalk
#4: Printing Through the Firmware

Revised by: Jim Luther September 1989
Written by: Matt Deatherage & Jim Luther July 1989

This Technical Note discusses considerations of printing through the AppleTalk firmware in transparent mode.
Changes since July 1989: Updated to reflect ROM 03 changes to AppleTalk firmware.

The AppleShare Programmer's Guide to the Apple IIGS states that printing in transparent mode (through Super Serial Card emulation for older applications which don't know about AppleTalk) is initiated when you do a PR#7 command.

This statement is pretty short-sighted. It's much like saying printing through an ImageWriter II is initiated when you do a PR#1 command--it's only true if what you want is where you think it is--and usually it isn't.

An Apple IIe Workstation Card, although recommended for slot 7, can work in almost any slot (just like an ImageWriter II can be connected to nearly any slot, except maybe slot 3 when the 80-column firmware is active). An Apple IIGS with ROM versions 00 or 01 may only have AppleTalk firmware located in slot 7. An Apple IIGS with ROM version 03 may only have AppleTalk firmware located in either slot 1 or 2.

Before printing through the Super Serial Card emulation to AppleTalk, take the same precautions you would take before printing to any slot--check to make sure you see the requested slot as a Pascal device before using Pascal entry points, and try to look for the signature bytes that indicate the features you want are present. In general, avoid hard-coding slot numbers for anything.

If your application wants to print over the network, you are already identifying AppleTalk as described in AppleTalk Technical Note #1, Identifying AppleTalk. Apple has defined a convention to allow applications to know, when possible, which slot or port the network connection should use for transparent printing. If the AppleTalk completion routine pointer points to an address in slot ROM space, then that slot contains the transparent network printing firmware. In other words, if the completion routine points to $0000CnXX, where n is between 1 and 7, then n is the slot to be used for transparent printing. If the completion routine pointer does not point to slot ROM, then the application cannot determine what slot to print through and must ask the user. (This situation will not happen on current Apple II computers, but it could happen in the future.)

Note: This convention returns a slot number between 1 and 7, which is not fully compatible with the Slot Arbiter. When using GS/OS, do not pass this number directly to the Slot Arbiter. Refer to Apple IIGS Technical Note #69, The Ins and Outs of Slot Arbitration.

This technique applies only to ProDOS 8 programs. Apple IIGS applications running under GS/OS should do text printing over the network through the Remote Print Manager driver, which can be identified by a deviceID of $001F as returned from DInfo.

The following 6502 code sample illustrates this technique:

```
;
;    This routine will identify AppleTalk and the slot AppleTalk is
;    associated with (if possible).
;

CheckATalk equ    *

;     Check for AppleTalk (see AppleTalk Technical Note #1)

           jsr    $BF00               ; ProDOS 8 MLI
           dfb    $42                 ; $42 command for network calls
           dw     InfoParams          ; Parameter list address
           bcs    NoATalk             ; no AppleTalk; handle the error

;    AppleTalk installed when here, so find the slot it uses

           lda    ComReturn+2         ; bank $00?
           ora    ComReturn+3         ; high byte = 0?
           bne    AskForSlot          ; no, so slot can't be determined
           lda    ComReturn+1         ; get the address page
           cmp    #$C8
           bge    AskForSlot          ; greater or equal to $C8 is bad
           cmp    #$C1
           blt    AskForSlot          ; less than $C1 is bad
           and    #$0F                ; $Cn = $0n
           sta    ATalkSlot

HaveSlot   equ    *                   ; AppleTalk is installed and
                                      ; is in slot # ATalkSlot

AskForSlot equ    *                   ; AppleTalk is installed but slot
                                      ; can't be determined

NoATalk    equ    *                   ; AppleTalk is not installed

           rts                        ; so this sample returns

AtalkSlot  dfb    $00                 ; Slot to use for transparent printing

InfoParams dfb    $00                 ; Synchronous only
           dfb    $02                 ; GetInfo call number
           ds     2                   ; result code
ComReturn  ds     4                   ; completion return address
           ds     8                   ; space for other result info
```

Further Reference

- o AppleShare Programmer's Guide for the Apple IIGS
- o Apple IIGS Technical Note #69, The Ins and Outs of Slot Arbitration
- o Apple II AppleTalk Technical Note #1, Identifying AppleTalk
- o Apple II Miscellaneous Technical Note #8, Pascal 1.1 Identification Bytes

END OF FILE TN.ATLK.004

##
FILE: TN.ATLK.005
##

Apple II
Technical Notes

Developer Technical Support

AppleTalk
#5: SPCommand Calls and Error $0702

Written by: Mark Day July 1989

The system now uses SPCommand calls asynchronously. Applications that have AppleShare volumes mounted under System Software 5.0 and also make SPCommand calls themselves should now handle the "Too many ASP calls" error, $0702.

AppleShare uses a protocol called AppleTalk Session Protocol (ASP) to maintain a connection (session) with all servers that you are logged on to. All commands and data transfer to the server are sent using ASP.

The implementation of ASP on the Apple IIGS has a limit of one command outstanding (waiting to complete) per session. This means that if one command has been sent, its reply must be received before you can send the next command. Remember, the SPCommand call is used to send commands over a session. If you try to issue an SPCommand before another (asynchronous) SPCommand on the same session has completed, your call will return with a "Too many ASP calls" error, $0702.

Before System Software 5.0 on the Apple IIGS, no system software made asynchronous SPCommand calls, and therefore this error would only occur if the developer was making the asynchronous calls. As of System Software 5.0, the AppleShare FST uses asynchronous calls to help prevent the loss of a connection with servers and to assist the Finder in dynamically updating windows when a change is made to a network volume. Therefore, this error may be returned even though the developer is not making asynchronous calls.

The error is easy to handle if you are making synchronous SPCommand calls. Simply make the call, and if it completes with error $0702, loop back and make the call again until you can do so without error $0702. This technique forces your program to wait until ASP is free again to make the call.

If you are making asynchronous SPCommand calls, and you receive the $0702 error, you might want to install a short (i.e., 1/4 second) timer using the InstallTimer call, and make the SPCommand call again when the timer completes. Remember, the InstallTimer has to be asynchronous, since you are making it from the completion routine of an asynchronous call.

The SPWrite call also has a limit of one outstanding call per session. System software does not currently use asynchronous SPWrite calls, but looping until ASP returns something other than $0702 would be a good precaution for SPWrite, too.

Note: When using the AppleShare FST under GS/OS, there is little reason to make SPCommand calls yourself, since most of the calls you can make are available through the FST as normal file system calls or as FST-specific calls.

Further Reference

- AppleShare Programmer's Guide for the Apple IIGS
- Inside AppleTalk
- System Software 5.0 documentation (APDA)

END OF FILE TN.ATLK.005

##
FILE: TN.ATLK.006
##

Apple II
Technical Notes

Developer Technical Support

AppleTalk
#6: Apple IIe Workstation Card Anomalies

Written by: Dan Strnad July 1989

This Technical Note describes known anomalies when using the Apple IIe Workstation Card.

- Pascal Protocol Serial STATUS call returns incorrect results.
  When using the Workstation card, the Pascal STATUS call (normally used for printing) does not properly indicate whether the card is ready to receive characters. Applications should avoid this call, as the Pascal WRITE call in the firmware will perform this function automatically.

- ProDOS 8 invisible bit is not respected.
  The invisible bit in the ProDOS 8 access byte was defined after the release of the Apple IIe Workstation Card, so the ProDOS Filing Interface present on the card treats this bit as reserved.

Further Reference

- AppleShare Programmer's Guide for the Apple IIGS
- Apple IIe Technical Reference Manual

END OF FILE TN.ATLK.006

```
#####################################################################
### FILE: TN.GSOS.001
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

GS/OS
#1: Contents of System.Disk and System.Tools

Revised by: Matt Deatherage — July 1989
Written by: Matt Deatherage — November 1988

This Technical Note describes the contents of the disks System.Disk and System.Tools and the minimum files necessary to boot GS/OS starting with System Software 5.0.
Changes since January 1989: Updated to reflect System Software 5.0.

This Note gives a description of each of the files in the Apple IIGS System Software 5.0 package. The package includes three disks: System.Disk, System.Tools, and the Apple II Setup disk for AppleShare 2.01. System.Disk is bootable and includes two drivers with 5.0. System.Tools is not bootable, but with 5.0 it includes other drivers, AppleTalk files, and some utility programs. The Apple II Setup disk is an update for AppleShare File Servers (version 2.01 or later) which updates them to boot into GS/OS. Since the software on this disk is not available for licensing and will not ship with applications, this Note does not cover its contents.

Contents of System.Disk

ProDOS	The boot file for GS/OS, ProDOS, contains the code necessary to load GS/OS from any particular file system. This file will be file-system dependent. For example, the file ProDOS on a bootable disk in the ProDOS file system will be different than the file ProDOS on a bootable disk in the High Sierra file system.
System	The directory containing most of the GS/OS files.
CDevs	The directory containing all Apple IIGS Control Panel Devices (CDevs) required for minimal operation.
Alphabet	Sets translation specifications and display languages.
DirectConnect	Allows selection of direct-connected printers.
General	Allows setting of general system parameters.
Keyboard	Sets keyboard parameters.
Modem	Controls modem port settings.
Monitor	Sets 40-column or 80-column mode,

	monochrome or color mode, and the color of text, text background, and borders.
Mouse	Sets mouse parameters.
Printer	Controls printer port settings.
RAM	Controls the size of the RAM disk and the GS/OS Disk Cache.
Slots	Allows selection of slot settings and startup slot.
Sound	Sets user preference for sound pitch and volume.
Time	Sets the internal clock's time and display format.
CDev.Data	A list of internal Control Panel parameters for each CDev in the directory; the list is precalculated for speed when opening the Control Panel.
Desk.Accs	The directory containing all the classic and new desk accessory files to be loaded at boot time.
CtlPanel.NDA	The new desk accessory which allows users to control almost all system parameters and choose printers and file servers.
Drivers	The directory containing all device drivers needed by GS/OS and the Toolbox (including the Print Manager and MIDI Tools).
AppleDisk3.5	The Apple 3.5 Drive device driver for GS/OS.
AppleDisk5.25	The driver for Apple 5.25" disk drives, including Disk II drives and Apple UniDisk 5.25 drives. This driver is required for GS/OS to recognize 5.25" disk drives.
Console.Driver	The text screen and keyboard device driver for GS/OS.
ImageWriter	The ImageWriter driver for the Print Manager.
Printer	The printer port driver for the Print Manager.
Modem	The modem port driver for the Print Manager.
Printer.Setup	A file containing the default printer driver and port driver settings for the Print Manager.
Error.Msg	A compiled file containing all error messages required by GS/OS. This file is separate from the GS.OS file to provide easier support for localization.
ExpressLoad	New routines for GS/OS which load specially processed files up to four times faster than previously possible prior to System Software 5.0. GS/OS loads ExpressLoad at boot time on systems with more than 512K total memory.
Fonts	The directory containing all system fonts to be used.
Courier.10	10 point Courier font.
Courier.12	12 point Courier font.
FastFont	A preshifted version of Shaston 8 which QuickDraw II loads at QDStartUp time and

```
                            uses to draw Shaston 8 text faster than
                            could normally be accomplished.  QuickDraw
                            II does not load FastFont on systems with
                            512K total memory.
    Geneva.10               10 point Geneva font.
    Geneva.12               12 point Geneva font.
    Helvetica.10            10 point Helvetica font.
    Helvetica.12            12 point Helvetica font.
    Shaston.16              16 point Shaston font.
    Times.10                10 point Times font.
    Times.12                12 point Times font.
    Venice.14               14 point Venice font.
    Font.Lists              A file prepared by the Font Manager when
                            FMStartUp is first called.  It contains
                            information about all the fonts in the
                            Fonts directory and is only recalculated
                            if the Font Manager reasonably believes
                            the information has changed.

FSTs                        The directory containing the file system
                            translators to be loaded at boot time.
    Char.FST                The character device FST.
    Pro.FST                 The ProDOS FST.
GS.OS                       The remainder of GS/OS.
GS.OS.Dev                   The GS/OS Device Manager and associated
                            core routines.  Separate from GS.OS for
                            speed reasons.
P8                          The ProDOS 8 operating system, version 1.8.
Start                       The boot program.  If this file exists,
                            GS/OS will always launch it upon boot.  In
                            this case, as in most cases, this is the
                            Finder.  The Finder for System Software
                            5.0 is V1.3.
Start.GS.OS                 The file containing the GLoader and GQuit
                            routines.  It loads the files GS.OS and
                            GS.OS.Dev, which contain the rest of the
                            operating system.
System.Setup                The directory containing all the
                            initialization files to be executed at
                            boot time.
    CDev.Init               A file, required for the Control Panel new
                            desk accessory, which executes any
                            initialization code in any CDev that is in
                            the CDev subdirectory.
    Resource.Mgr            The Resource Manager, V1.0.  This is an
                            initialization file since the design of
                            the Resource Manager requires it to be
                            present even when an application has not
                            specifically loaded it.  If this file is
                            not present, the system will not boot.
    Sys.Resources           A file containing system resources used by
                            the tools and the Control Panel, and which
                            are available to applications.
    Tool.Setup              A required file that loads TS2, which
                            contains all the patches to tools in ROM
                            for ROM level 01.  Tool.Setup would
                            attempt to load TS1 if executed on a
                            machine with ROM level 00, but GS/OS does
```

```
                                not boot on such a machine, therefore, TS1
                                is not included.
        TS2                     All the patches to ROM tools for ROM level
                                01.
        TS3                     A required file included for future
                                compatibility.
    Tools                       The directory containing tool files for
                                all tools not in ROM.
        Tool014                 Window Manager V3.1.
        Tool015                 Menu Manager V3.1.
        Tool016                 Control Manager V3.1.
        Tool018                 QuickDraw Auxiliary V3.0.
        Tool019                 Print Manager V3.0.
        Tool020                 LineEdit V3.0.
        Tool021                 Dialog Manager V3.1.
        Tool022                 Scrap Manager V3.0.
        Tool023                 Standard File V3.0.
        Tool025                 Note Synthesizer V1.4.
        Tool026                 Note Sequencer V1.4.
        Tool027                 Font Manager V3.1.
        Tool028                 List Manager V3.1.
        Tool029                 ACE Tools V1.1.
        Tool034                 TextEdit V1.0.
Finder.Def                      Finder default settings file.  This file
                                must be present on the backup copy of
                                System.Disk you use with the Installer
                                program.  The Installer will not be able
                                to install GS/OS if this file is not
                                present on System.Disk.
Icons                           The directory containing all the icon
                                files used by the Finder.
    Finder.Icons                The core set of icons used by the
                                Finder for all system files and devices.
    Finder.Icons.X              The additional icons used by the Finder on
                                systems with more than 512K total total
                                memory.
    FType.Main                  The file type names used by the Finder on
                                all systems.
    FType.Aux                   The additional file type names used by the
                                Finder on systems with more than 512K
                                total memory.
AppleTalk                       A directory containing files to implement
                                the AppleTalk networking protocols.  On
                                this disk, this folder is empty.
BASIC.System                    The ProDOS 8 BASIC command interpreter,
                                V1.3.
BASIC.Launcher                  A short program which allows BASIC.System
                                to run AppleSoft program files which are
                                opened from the Finder.
Tutorial                        A directory containing several "empty"
                                files (files containing two carriage
                                returns) and other directories, used in
                                user-level documentation to teach the
                                concepts of a hierarchical file system.
                                These files are absolutely unnecessary to
                                the operation of the System Software.
    Budgets
        Finder.Data
```

```
        Home
            CY.1990
            CY.1991
            Finder.Data
        Office
            Finder.Data
            FY.1990
            FY.1991
    Finder.Data
    Graphics
        Ad
        Finder.Data
        Flier
        Letterhead
        Masthead
    Letters
        Finder.Data
        Mr.Merritt
        Ms.Bachtold
        To.Family
            Dad
            Finder.Data
            Mom
        TO.FRIENDS
            Darryl
            Finder.Data
            Molly
```

Contents of System.Tools

```
Icons                           Additional icons for the Finder.  This
                                folder is currently empty.
System                          A directory containing additional parts of
                                GS/OS not found on System.Disk.
    CDevs                       Directory with additional Control Panel Devices.
        AppleShare              Allows users to choose and log onto
                                AppleShare file servers.
        ATIWriter               Allows users to choose ImageWriter
                                printers on AppleTalk networks for use
                                with the Print Manager.
        ATLQIWriter             Allows users to choose ImageWriter LQ
                                printers on AppleTalk networks for use
                                with the Print Manager.
        ATLWriter               Allows users to choose LaserWriter
                                printers on AppleTalk networks for use
                                with the Print Manager.
        DirectConnect           Allows selection of direct-connected
                                printers.
    Desk.Accs                   Directory with additional desk
                                accessories.
        CDRemote                An updated version of the CD Remote new
                                desk accessory which ships with the
                                AppleCD SC.  This version works with the
                                SCSI Manager in System Software 5.0.
        VideoMix.NDA            An updated version of the VideoMix new
                                desk accessory which ships with the Apple
                                II Video Overlay Card.
```

Drivers	Directory with additional device drivers for GS/OS and the Toolbox.
Apple.Midi	The Apple MIDI Interface driver for the MIDI Tools.
AppleDisk5.25	The driver for Apple 5.25" disk drives, including Disk II drives and Apple UniDisk 5.25 drives. This driver is required for GS/OS to recognize 5.25" disk drives.
AppleTalk	The AppleTalk port driver for the Print Manager. It works with either serial port when configured for AppleTalk.
AT.IW.PSetup	This file contains the same information as the file Printer.Setup for an ImageWriter printing through AppleTalk. The Installer replaces the file Printer.Setup on the destination disk with this file and renames it Printer.Setup.
AT.IWLQ.PSetup	This file contains the same information as the file Printer.Setup for an ImageWriter LQ printing through AppleTalk. The Installer replaces the file Printer.Setup on the destination disk with this file and renames it Printer.Setup.
ATalk	The main AppleTalk GS/OS driver.
ATP1.ATROM	AppleTalk protocols to patch the IIGS ROM.
ATP2.ATRAM	AppleTalk protocols not in ROM.
Card6850.MIDI	The driver for 6850-based MIDI interface cards for the MIDI Tools.
Epson	The Epson(R) printer driver for the Print Manager.
EPSON.PSetup	This file contains the same information as the file Printer.Setup for an Epson printing through the parallel card driver. The Installer replaces the file Printer.Setup on the destination disk with this file and renames it Printer.Setup.
ImageWriter	The ImageWriter driver for the Print Manager.
ImageWriter.LQ	The ImageWriter LQ driver for the Print Manager. This driver currently has no more functionality than the ImageWriter driver.
IW.PSetup	This file contains the same information as the file Printer.Setup for an ImageWriter printing through the printer port. The Installer replaces the file Printer.Setup on the destination disk with this file and renames it Printer.Setup.
IWEM	PostScript(R) program which allows a LaserWriter emulate an ImageWriter. A user can load it into the LaserWriter with the LaserWriter CDev, and it is automatically invoked when printing through the slot associated with AppleTalk.
IWLQ.PSetup	This file contains the same information as the file Printer.Setup for an ImageWriter LQ printing through the printer port. The Installer replaces the file Printer.Setup

	on the destination disk with this file and renames it Printer.Setup.
LaserWriter	The LaserWriter driver for the Print Manager. This driver works with any LaserWriter with PostScript. It does not work with the LaserWriter IISC.
LW.PSetup	This file contains the same information as the file Printer.Setup for an LaserWriter printing through AppleTalk. The Installer replaces the file Printer.Setup on the destination disk with this file and renames it Printer.Setup.
Modem	The modem port driver for the Print Manager.
Parallel.Card	A driver for some parallel printer interface cards for the Print Manager. This driver works with the Apple Parallel Interface Card, as well as several other parallel interface cards.
Printer	The printer port driver for the Print Manager.
SCC.Manager	The GS/OS supervisory driver that arbitrates hardware-level usage of the SCC in the Apple IIGS.
SCSI.Manager	The GS/OS SCSI Manager, the supervisory driver that arbitrates hardware-level usage of Apple II SCSI cards.
SCSICD.Driver	The GS/OS driver for the AppleCD SC drive. This driver is required for GS/OS to recognize CD-ROM drives.
SCSIHD.Driver	The GS/OS driver for SCSI hard disks. This driver is required for GS/OS to recognize SCSI hard disks.
UniDisk3.5	The GS/OS driver for UniDisk 3.5 drives. This driver is required for proper operation of UniDisk 3.5 drives. Using the UniDisk with GS/OS without this driver eventually corrupts media.
FSTs	Directory with additional File System Translators.
AppleShare.FST	The AppleShare FST which allows GS/OS to access AppleShare file servers.
HS.FST	The High Sierra FST which allows GS/OS to access CD-ROM discs formatted in the international standard High Sierra or ISO 9660 formats. This FST is read-only; it only performs read operations.
System.Setup	Directory with additional initialization files.
AppleIIVOC.INIT	An initialization file used by the Apple IIGS Video Overlay Card tool set.
ATInit	The AppleTalk initialization file.
ATResponder	The AppleTalk responder, used for AppleTalk network management.
Tools	Directory with additional tools.
Tool032	MIDI Tools, V1.3.
Tool033	Video Overlay Card tool set V1.1
Fonts	Directory with additional fonts. This

	directory is currently empty.
Scripts	This directory contains all the scripts for the Installer. On launch, the Installer looks in its parent directory for the Scripts directory and the scripts it contains.
ADV.DISK.UTIL	Script to install the Advanced Disk Utility program.
APPLE.MIDI	Script to install the Apple MIDI Interface driver and tool set.
APPLEDISK5.25	Script to install the 5.25" disk driver for GS/OS.
APPLESHARE	Script to install AppleShare.
Aristotle.Patch	Script to install a change to Aristotle for easier class transition.
ATIMAGEWRITER	Script to install the ImageWriter printer driver for the Print Manager, as well as the files necessary to work with AppleTalk.
ATIMAGEWRITERLQ	Script to install the ImageWriter LQ printer driver for the Print Manager, as well as the files necessary to work with AppleTalk.
CARD6850.MIDI	Script to install the 6850-based MIDI Interface card driver.
CDROM	Script to install the High Sierra FST as well as the SCSI Manager and SCSI CD-ROM driver for GS/OS.
DCIMAGEWRITER	Script to install the ImageWriter printer driver for the Print Manager, as well as the files necessary to connect it to a serial port.
DCIMAGEWRITERLQ	Script to install the ImageWriter LQ printer driver for the Print Manager, as well as the files necessary to connect it to a serial port.
EPSON	Script to install the Epson printer driver for the Print Manager, as well as the parallel card driver.
FONTS	Script to install additional fonts. No additional fonts are currently supplied.
INST.SYS.MIN	Script to install a minimal GS/OS system on a given destination volume.
INST.SYSF.NOFIN	Script to install a minimal GS/OS system,without the Finder, on a given destination volume.
INSTAL.SYS.FILE	Script to install a GS/OS system, with the Finder, on a given destination volume.
LASERWRITER	Script to install the LaserWriter printer driver for the Print Manager, as well as the files necessary to work with AppleTalk.
Local.Net.Boot	Script to create a 3.5" floppy disk with the minimum configuration necessary to boot locally but log onto an AppleShare file server.
NAMER	Script to install Namer II and related AppleTalk files.
Quick.Logoff	Script to add a quick logoff feature

```
                                to AppleShare.
    SCSI.HARD.DISK              Script to install the SCSI Manager and
                                SCSI hard disk driver for GS/OS.
    Server.Sys.File             Script to install System Software 5.0 on
                                an AppleShare File Server.
    UNIDISK3.5                  Script to install the UniDisk 3.5 driver
                                for GS/OS.
    VIDEOMIX                    Script to install the latest versions of
                                the Apple II VideoMix software and tools.
Installer                       The Apple IIGS Installer program.  This
                                program makes use of scripts found in the
                                Scripts directory on this disk to install
                                parts of the system, as well as third-
                                party applications, without the user
                                needing to copy individual files.
AppleTalk                       This directory contains additional
                                AppleTalk files and utilities for
                                AppleShare and AppleTalk.
    Boot.Driver                 A driver for AppleShare that GS/OS loads
                                before the other drivers are loaded and
                                which remains resident in memory after the
                                boot process is finished.  Installed on
                                servers by the Installer script
                                Server.Sys.File.
    Display.0
    Namer                       This directory contains the Namer II
                                application to rename AppleTalk devices.
        MtxAbs.0                MouseText code routines for by Namer II.
        Namer.II                The Namer II application (a ProDOS 8
                                program).
        Namer.0                 Additional code needed by Namer II.
    QuickLogoff                 An initialization file used to add a quick
                                logoff feature to AppleShare.
    Start                       The AppleShare startup program which is
                                installed in place of the Finder on
                                AppleShare volumes.  It allows the user to
                                log on and then launches the server
                                startup program for the user's machine.
Adv.Disk.Util                   The Advanced Disk Utility program which
                                allows for partitioning of SCSI hard
                                disks, as well as erasing, initializing,
                                and zeroing volumes or partitions.
```

Minimum GS/OS System Disk Requirements

The following files are required for GS/OS to boot. This list does not address files needed by the Finder or the IIGS Toolbox. Those files only required in certain circumstances are noted as such. Those files that may be excluded only when disk space or memory limitations make it absolutely necessary are marked with asterisks (*).

```
ProDOS
System
    Start.GS.OS
    GS.OS
    GS.OS.Dev
    Error.Msg
```

```
FSTs
    Pro.FST                 Required for ProDOS disks.
    HS.FST                  Required for High Sierra or ISO 9660
                            discs.
    Char.FST
    AppleShare.FST          Required to use AppleShare file servers
Drivers
    AppleDisk3.5            Required for Apple 3.5 Drives.
    AppleDisk5.25           Required for 5.25" drives.
    UniDisk3.5              Required for UniDisk 3.5 drives.
    SCSI.Manager            Required for SCSI devices.
    SCSIHD.Driver           Required for SCSI hard disks.
    SCSICD.Driver           Required for AppleCD SC drives.
    Console.Driver
    ATalk                   Required for AppleTalk (including
                            AppleShare).
    ATP1.ATROM              Required for AppleTalk (including
                            AppleShare).
    ATP2.ATRAM              Required for AppleTalk (including
                            AppleShare).
    SCC.Manager             Required for AppleTalk (including
                            AppleShare).
System.Setup
    CDev.INIT               Required for the Control Panel NDA.
    Tool.Setup
    TS2
    TS3
    Resource.Mgr
    Sys.Resources
CDevs
    Alphabet*
    AppleShare*             Required for selecting AppleShare file
                            servers.
    ATIWriter*              Required for choosing printers.
    ATLQIWriter*            Required for choosing printers.
    ATLWriter*              Required for choosing printers.
    DirectConnect*          Required for choosing printers.
    General*
    Keyboard*
    Modem*
    Monitor*
    Mouse*
    Printer*
    RAM*                    Should always be included if possible.
                            It provides the only way to set the size
                            of the GS/OS Disk Cache.
    Slots*
    Sound*
    Time*
    CDev.Data*              Only required if using the same CDevs that
                            ship on System.Disk.
Desk.Accs*                  Required for desk accessories; any desk
                            accessories should be installed in this
                            directory.
    CtlPanel.NDA*
ExpressLoad*                The only reason not to ship ExpressLoad is
                            a lack of disk space; it is not loaded in
                            512K systems.
```

Start	Must be present for GS/OS to boot or some other file that GS/OS can boot into must be present in its place.
Tools	Required for any of the RAM-based tools; any RAM-based tools should be installed in this directory.
Fonts	Required for the Font Manager.
FastFont*	This makes Shaston 8 text drawing much faster and should be included unless absolutely impossible.
P8	Required for ProDOS 8.
BASIC.System	Required for AppleSoft BASIC.
BASIC.Launcher	Required for AppleSoft BASIC if the user is allowed to open these programs from the Finder.

Further Reference

- o GS/OS Reference, Volumes 1 and 2

Epson is a registered trademark of Seiko Epson Corporation.
PostScript is a registered trademark of Adobe Systems, Incorporated.

END OF FILE TN.GSOS.001

##
FILE: TN.GSOS.002
##

Apple II
Technical Notes

Developer Technical Support

GS/OS
#2: GS/OS and the 80-Column Firmware

Written by: Matt Deatherage November 1988

This Technical Note discusses the changes in handling the 80-column firmware between GS/OS and ProDOS 16.

For compatibility with the Apple IIe, the Apple IIGS does not treat slot 3 like it treats other slots. Instead of using a bit in the Slot Register ($C02D) to control the mapping of ROM in slot 3 between the built-in 80-column firmware and any peripheral card physically in slot 3, the soft switches SETINTC3ROM ($C00A) and SETSLOTC3ROM ($C00B) are used instead. On the Apple IIe, these soft switches (referred to by the single label SLOTC3ROM) respectively map the ROM at $C300 to the internal 80-column firmware (which works with the auxiliary-slot 80-column card in most IIe computers) or to a peripheral card in slot 3. Note that writing to SETSLOTC3ROM on a IIe or IIGS with no card in slot 3 results in floating bus addresses in the $C300 space.

ProDOS 8 will not allow an Apple IIe or later model computer to have a card other than an 80-column card in slot 3. ProDOS 8 needs the 80-column firmware on a 128K machine for use in the /RAM driver, and the enhanced Apple IIe has some of the interrupt firmware in the $C300 space. When ProDOS 8 is loaded in an Apple IIe or later, it writes to SETSLOTC3ROM and looks at five identification bytes. If all five of these bytes do not match, ProDOS 8 will write to SETINTC3ROM to use the internal firmware. If all five bytes match, the external slot 3 ROM is left mapped in.

ProDOS 16 fell victim to a bug in ProDOS 8 versions 1.2 through 1.6 which always switched in the internal 80-column firmware, regardless of the user's Control Panel setting. GS/OS does not have this bug; a card in slot 3 of a IIGS other than an 80-column card will not be mapped out by GS/OS.

Application programmers who require the 80-column firmware should be familiar of the following points:

o If your program contains a routine to insure that the 80-column firmware is indeed available, it could be buggy. Since ProDOS 16 always made the 80-column firmware available, your routine to check that condition may never have been executed.
o If your program requires the 80-column firmware and it is not available, your program should display a message on the screen informing the user that he must set Slot 3 in the Control Panel to Built-in Text Display for your program to execute, then gracefully exit. Switching the $C300 ROM space, even with the user's permission, is not recommended. Slot 3 could contain an operating

GS/OS device, perhaps even the one your program was launched from. Remember, it is possible to boot GS/OS from slot 3.

Do not try to be clever in a situation like this. For example, do not go looking at ID bytes in slot 3 to try to determine the type of device present so that you can switch it out if you identify it as a non-disk device. Slot 3 could contain an active device being operated by a loaded GS/OS driver.

Your program should not ask the user's permission to switch ROM space between ports and slots (or in this case, the internal firmware versus the external card). That is why there is a Control Panel. Simply display a message informing the user that he must set Slot 3 in the Control Panel to Built-in Text Display for your program to execute. You may offer to change the battery RAM parameter for the user and restart the system (using the OSShutdown call), but under no circumstances should you hit the soft switch yourself, even with the user's permission.

Further Reference

- GS/OS Reference, Volume 1
- ProDOS 8 Technical Note #15, How ProDOS 8 Treats Slot 3

END OF FILE TN.GSOS.002

##
FILE: TN.GSOS.003
##

Apple II
Technical Notes

Developer Technical Support

GS/OS
#3: Pointers on Caching

Written by: Matt Deatherage November 1988

This Technical Note discusses effective use of the GS/OS cache.

Introduction

GS/OS is the first Apple II operating system to offer a sophisticated caching mechanism. However, using the cache and using it wisely are two different things. This Note presents some concepts which should lead to higher performance for your application if it uses the cache.

What's Cached Automatically?

All blocks on a GS/OS readable disk could be classified into one of two categories. "Application blocks" are all blocks on the disk contained in any file (except a directory file), while "system blocks" are other blocks on the disk. System blocks belong to the file system and include directory blocks, bitmap blocks, and other housekeeping blocks specific to the file system.

GS/OS always maintains at least a 16K cache, even if the user has set the disk cache size to 0K with the Disk Cache new desk accessory. When the system (usually an FST) goes to read a system block, the block is identified as a candidate for caching and is cached if possible. Applications define blocks as candidates for caching by using the cachePriority field of many class 1 GS/OS calls. Note that class 0 calls do not have this field, thus applications using exclusively class 0 calls will not be able to cache any application blocks.

Although this difference may seem like a limitation, it in fact improves performance. On the Macintosh, most applications that work with files (like database managers) leave the file with which they are working open while they need it; the file is only closed when the window containing it is closed. Apple II programs historically are quite different--they usually read an entire file at the beginning, modify it in memory, and write it when the save function is selected. A moment's thought will show that if GS/OS arbitrarily cached most or all application blocks, system blocks that would be used again (such as directory blocks) will be kicked out to make room for them. We will see that this is probably a bad thing to do.

How to Cache Effectively

The first tendency of many programmers is to attempt to completely cache any given file, but this usually leads to a degradation in performance, not an improvement. In small caches such strategies can slow the system to a crawl, and large caches offer no significant improvement. Remember that until the cache memory is needed, it is available to the system. The cache size for GS/OS as set by the user is the maximum to be allotted, not the minimum.

Suppose you are attempting to cache a 40K file (80 512-byte blocks). If the cache is set to less than 40K, the entire cache will be written through, kicking out all system blocks currently cached. A cache of this size slows system performance for little gain, since the entire file could not be cached anyway. Even if the cache is large enough to hold the entire file, you are needlessly taking twice the amount of memory with the same file (by reading it into memory you have obtained from the Memory Manager and by asking GS/OS to keep a copy in the cache).

It is evident that the system makes the best use of the cache automatically, freeing your application from the duty of caching system blocks, but there are certain instances where caching application data can improve system performance.

An application which does not limit document size to available memory will often only keep a portion of the document in memory at any given time. Suppose that the beginning of such an application's document file contains a header which to various parts of the document file. (These parts could be chapters for a word processor, report formats for a database manager, or individual pictures for an animation program.) This document header is probably not very long, but the application will likely need to read it quite often to quickly access various portions of the document file.

This header is a prime candidate for caching since it is a part of the file which will definitely be read many times during the life of the application. Contrast this with arbitrarily caching the entire file, which needlessly wastes both cache space and available memory to keep a duplicate copy of something that may or may not be read from disk again.

Although caching provides enormous benefits to GS/OS, indiscriminate use of the cache will waste memory and degrade overall system performance. Be prudent and limit your use of the cache to those portions of your document files which will be read from disk many times.

Further Reference
o   GS/OS Reference, Volume 1

END OF FILE TN.GSOS.003

##
FILE: TN.GSOS.004
##

Apple II
Technical Notes

Developer Technical Support

GS/OS
#4: A GS/OS State of Mind

Revised by: Dave Lyons & Matt Deatherage July 1989
Written by: Matt Deatherage January 1989

This Technical Note discusses GS/OS concepts and practices.
Changes singe January 1989: Updated for System Software 5.0.

Although GS/OS bears many similarities to ProDOS, GS/OS is a much wider-reaching operating system, working not only with multiple file systems but also with character devices. Some things which work under ProDOS cause problems under GS/OS, and application programmers need to be aware of the differences, particularly those developing text-based programs.

GS/OS Hints

Be aware of character devices. A legal GS/OS pathname, perhaps entered by a user in response to a prompt, could map to a character device, with potentially disastrous results. Error $58, Not a Block Device, can protect you against this on many calls, including Create, but you must still take precaution. DInfo tells you if a device is a character device or block device; bit seven of the characteristics word is set if the device is a block device.

Don't preprocess pathnames. A user input routine which prevents users from entering pathnames that don't follow ProDOS syntax may help prevent Illegal Pathname Syntax errors, but it also keeps users from creating files on non-ProDOS disks with anything but ProDOS pathname syntax, and it could keep them from accessing files on non-ProDOS disks which they created with another GS/OS application. Since the only FST which allowed you to write to a device under System Software 4.0 was ProDOS, you didn't see this problem right away. However, System Software 5.0 includes an AppleShare FST which, compared to ProDOS, is fast and loose with pathnames. "How about an anti-ProDOS name?" is a legal AppleShare filename. To allow compatibility with present and future non-ProDOS FSTs, Apple suggests you pass user-entered pathnames directly to GS/OS, with no application preprocessing.

Remember that under GS/OS both colons and slashes are valid separators, and colons can only be separators. In addition, all eight bits of each byte of a pathname are significant. Refer to GS/OS Reference, Volume 1 for more information on GS/OS pathname syntax. Using all eight bits of each byte may be particularly difficult for text-based applications, which have no way to force the standard Apple II character set to display characters such as sigma or the copyright symbol; they can fiddle to get characters like the sterling pound sign and an Apple. Some programs may wish to adopt special typographical conventions for these special characters while others may choose not to create files with such characters in their names. These programs could present the user with a list of existing filenames (with some substitution for the characters which are unavailable), while providing a method of choosing one, to retrieve such files. Any way around this problem for a text-based program will be less than optimal.

Avoid the Text Tools and all slot dependencies. Preliminary GS/OS documentation points to a System Service call named DYN_SLOT_ARBITER. This mechanism, which is not fully implemented in System Software 5.0, eventually will allow the operating system to use internal ports and external slots for the same "slot" in the same session, instead of requiring the user to reboot the system to safely change between ports and slots. Applications which have hard-coded slot dependencies (as the Text Tools unfortunately require) make this transition very difficult, both for GS/OS and for the applications and users. We recommend that applications use the GS/OS loaded and generated character device drivers for text output. A DInfo call will tell you what slot or port a driver controls, and whether or not it is a character device.

Avoid other file system dependencies. Many of the things ProDOS programmers are used to as facts of life just are not true any longer. For example, filenames don't have to be 15 characters or less under GS/OS. When making class one calls, GS/OS will tell you if you don't have enough room for the pathname by returning a Buffer Too Small error ($4F). Avoiding file system dependencies means handling this error intelligently: if you receive it, allocate more space for the buffer and try the call again. GS/OS will tell you how much space is needed. If you absolutely must hard code pathnames, such as volume names, be sure to use the colon as the separator, because if you do not, filenames with slashes will cause problems. Similarly, don't assume any of the following:

- There can only be 51 files in the volume directory
- All devices are named ".Dn," where n is the device number
- All blocks are 512 bytes long
- All devices are block devices
- Any other ProDOS-specific characteristics

Don't hog all of the memory. While this is never a good idea on the IIGS, it's even worse under GS/OS. To process things like pathnames, GS/OS allocates memory through the Memory Manager. If you've allocated all of available memory (i.e., for a disk copy procedure), GS/OS will be forced to return an Out of Memory error ($54). If the condition is so severe that GS/OS can no longer function, it will return a fatal GS/OS error with an ID = 2, and the user will be asked to restart the system.

(A common cause of fatal GS/OS error 2 during development is using a length byte instead of a length word on a class one string. Doing so almost always causes the first word to be greater than 8K, which is the maximum length of pathnames under GS/OS. GS/OS then dies for your enjoyment, as it is unable to allocate the memory for the pathname because it's too big, even if more than 8K is available.)

Hard code as little as possible. Even seemingly static things like device names should not be hard coded, since a new loaded driver could change the name of the same device at any time. Also, it may be possible in the future for users to rename devices.

Only ask for the access you need. If you're just going to read a file, make a call to Open the file with read permission only. In file systems where access privileges mean more than they traditionally have in ProDOS (where things are usually "Locked" or "Unlocked"), this could save some trouble. For example, AppleShare allows the same file to be opened multiple times as long as each open is with read-only access. If your program is only going to read a file, opening it with read and write access needlessly denies others on the server access to the file.

Copy all GS/OS information with files. Applications that copy files need to do more than copy the data fork of the file. If the file is extended, the resource fork of the file should be copied as well. In addition, when requested, each FST returns an option_list that contains information specific to the host file system that GS/OS does not use (i.e., AppleShare's option_list includes Finder information and access privileges). Calls to GetFileInfo and Open can return the option_list, while a call to SetFileInfo can set it. An FST will not set parameters in the option_list which should not be altered (just as SetFileInfo skips the EOF fields in GetFileInfo records). To ensure that the duplicate has as much host file system information from the original as can reasonably be transferred, always copy the option_list.

Further Reference

- o GS/OS Reference, Volumes 1 and 2

END OF FILE TN.GSOS.004

##
FILE: TN.GSOS.005
##

Apple II
Technical Notes

Developer Technical Support

GS/OS
#5: Resource Fork Formats

Revised by: Matt Deatherage July 1989
Written by: Matt Deatherage January 1989

This Technical Note discusses the resource fork format of GS/OS extended files.
Changes since January 1989: Documented the location of resource fork format information.

Due to an omission in GS/OS Reference, Volume 1, some developers are not aware that the format of the resource fork of any file is reserved by Apple Computer, Inc. With the release of System Software 5.0 for the Apple IIGS, a Resource Manager is available to manipulate discrete chunks of data stored in the resource forks of files. To prevent corruption of media, information should only be stored in any resource fork in this format.

The Resource Manager should always be used to manipulate the data in resource forks. Some utilities may find this impossible and will require direct manipulation of resources without the Resource Manager. Information on the format of the resource forks is included with the Resource Manager documentation in the System Software 5.0 documentation.

Further Reference

- GS/OS Reference, Volume 1
- System Software 5.0 documentation (APDA)

END OF FILE TN.GSOS.005

##
FILE: TN.GSOS.006
##

Apple II
Technical Notes

Developer Technical Support

GS/OS
#6: Drivers and GS/OS Direct Page

Revised by: Dave Lyons July 1989
Written by: Matt Deatherage March 1989

This Technical Note corrects an error in the preliminary GS/OS documentation and provides an alternate suggestion for developers who are writing GS/OS drivers.
Changes since March 1989: Added information about setting the D register before making system service calls and documented that the GS/OS direct page is now guaranteed to remain the same between Driver_StartUp and Driver_ShutDown calls.

Preliminary GS/OS documentation, including the beta draft of GS/OS Reference, Volume 2, incorrectly states that locations $5A through $5F are available for device drivers, and that locations $66 through $6B are shared by device drivers and supervisory drivers (and may be corrupted by either a driver or supervisory driver call).

This is not correct. The locations in question are used by GS/OS; destroying these locations can cause system failure and media corruption.

Drivers which require direct page space of their own should request it from the Memory Manager when they are started. Upon receiving a call, a driver can save the value of the D register (containing the GS/OS direct page) and switch to its own direct page. The driver may keep the value of its direct page inside the driver itself; no space on GS/OS direct page is available for this purpose. The driver must restore the D register to point to the GS/OS direct page before returning from the call, and it should also dispose of its direct page space when it shuts down.

The driver must also set the D register to point to the GS/OS direct page before making any system service call other than SET_SPEED and DYN_SLOT_ARBITER.

Note: The location of the GS/OS direct page is guaranteed to remain the same between Driver_StartUp and Driver_ShutDown calls.

Further Reference

- GS/OS Reference, Volume 2

END OF FILE TN.GSOS.006

##
FILE: TN.GSOS.007
##

Apple II
Technical Notes

Developer Technical Support

GS/OS
#7: Behavior of SET_DISKSW

Written by: Matt Deatherage July 1989

This Technical Note discusses changes to the documented behavior of SET_DISKSW in System Software 5.0. This Note is primarily of interest to device driver authors.

GS/OS Reference, Volume 2, states that the system service call SET_DISKSW ($01FC90) will remove a device's blocks from the cache and place its volumes off line.

With System Software 5.0, this behavior is slightly changed. SET_DISKSW also posts insertion and ejection notices to the GS/OS Notify Procedure queue, so that notification procedures may be called. This requires SET_DISKSW to check the current status of the device to know if the disk switched condition indicates an insertion or an ejection (by comparing the current device status against the device-dispatcher maintained status).

A GS/OS driver may have an interrupt handler present to handle interrupts generated by its device on insertion or ejection (if the hardware is capable of generating such interrupts). Such an interrupt handler will probably want to call SET_DISKSW when an insertion or ejection is detected to make the rest of the operating system aware of it. However, SET_DISKSW obtains the device's status based on the deviceNum and callNum on the GS/OS direct page.

Any driver or interrupt handler calling SET_DISKSW must first save the values for deviceNum and callNum on the GS/OS direct page, replacing callNum with the number of a driver call that accesses media (Apple suggests Driver_Read, $0002) and replacing deviceNum with the number of the device for which SET_DISKSW is being called. The caller must restore the original values after SET_DISKSW returns.

Although SET_DISKSW saves and restores the GS/OS direct page, the caller must know where the GS/OS direct page is located so it can place the proper parameters there. The value used for the GS/OS direct page should be the value of the D register when the driver receives its Driver_StartUp call. The GS/OS direct page is now guaranteed to remain constant between Driver_StartUp and Driver_ShutDown calls.

Further Reference

- GS/OS Reference, Volume 2

END OF FILE TN.GSOS.007

```
#####################################################################
### FILE: TN.GSOS.008
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

GS/OS
#8: Filenames With More Than CAPS and Numerals

Written by: Matt Deatherage July 1989

This Technical Note discusses the problems some applications may have when dealing with filenames containing lowercase letters for the first time.

With System Software 5.0, lowercase filenames enter GS/OS en masse for the first time. Lowercase filenames are inherent to the AppleShare filing system and have been added to the ProDOS filing system through the ProDOS FST. However, since Apple II filing systems never had lowercase characters in filenames before, this change undoubtedly causes problems for some applications. This Note gives general guidelines to help developers avoid such problems.

How the ProDOS FST Does It

"Wait," you say (not for any particular reason, other than a general fondness for monosyllables). "If you put lowercase characters in the ProDOS directory entry, it's going to cause all kinds of problems. What's gonna' happen on][+ machines?"

Two previously unused bytes in each file's directory entry are now used to indicate the case of a filename. The bytes are at relative locations +$1C and +$1D in each directory entry, and were previously labeled version and min_version. Since ProDOS 8 never actually used these bytes for version checking (except in one case, discussed below), they are now used to store lowercase information. (In the Volume header, bytes +$1A and +$1B are used instead.)

If version is read as a word value, bit 7 of min_version would be the highest bit (bit 15) of the word. If that bit is set, the remaining 15 bits of the word are interpreted as flags that indicate whether the corresponding character in the filename is uppercase or lowercase, with set indicating lowercase. For example, the filename Desk.Accs has a value in this word of $B9C0, or binary 1011 1001 1100 0000. The following illustration shows the relationship between the bits and the filename:

```
        Bits in WORD:                   1011100111000000
        Filename:                       Desk.Accs
        Uppercase or Lowercase:         ULLLUULLL
```

Note that the period (.) is considered an uppercase character.

What it Means

Because no lowercase ASCII characters are actually stored in the filename fields of the directory entries, all ProDOS 8 software should continue to work correctly with disks containing files with lowercase characters in the filenames. Neither ProDOS 8 nor the ProDOS FST are case sensitive when searching for filenames: ProDOS is the same file as PRODOS is the same file as prodos.

The main trouble applications have is when a filename has been "processed" by the application before passing it to GS/OS. For example, if a command shell automatically converts filenames to all uppercase characters before passing them to ProDOS 16, the chosen uppercase and lowercase combination for the filename will never be seen by the user without any apparent reason. Some developers have considered it okay to ignore lowercase considerations, thinking that they would only apply to file systems other than ProDOS (and file systems which would not be available on the Apple II for a long time, if ever). These developers were mistaken.

A more pressing problem is that of an application that is looking for a specific file, perhaps a data file or a configuration file. If the application simply passes a pathname to GS/OS and asks for that file to be opened, it will be opened if it exists. The case of the filename is irrelevant since file systems are not case sensitive. However, if the application makes GetDirEntry calls on a specific directory, looking for the filename in question, there could be trouble: the application won't find the file unless its string comparison routine is not case sensitive. If the user has renamed the file MyApp.Config, and the string comparison is looking for MYAPP.CONFIG, then the application will report that the file does not exist.

It is repeated here that when dealing with normal OS considerations, it's almost always better to ask for something and respond intelligently if it's not there than it is to go looking for it yourself. The OS already has a lot of code to look for things (or expand pathnames, or examine access privileges, etc.), and reinventing the wheel is not only tedious, it can be detrimental to future compatibility.

The One Exception

In the past, ProDOS 8 did look at the version bytes when opening a subdirectory. The code to do this has been removed from ProDOS 8 V1.8. Please be aware that earlier versions of ProDOS 8 will be unable to scan subdirectories with lowercase characters in the directory name, even to find files in those directories.

Conclusion

Most user-input routines (including the Standard File tool set) return filenames or pathnames that can be passed directly to GS/OS without preprocessing. Doing so may return "pathname syntax errors" more often than not doing so, but it also enables applications to take advantage of future versions of the System Software that loosen the restrictions on syntax (or new file systems that never had such restrictions). Under GS/OS, even ProDOS disks aren't what they used to be.

Further Reference

- o GS/OS Reference

END OF FILE TN.GSOS.008

```
#####################################################################
### FILE: TN.IIGS.001
#####################################################################
```

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 any 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.
- Under no circumstances 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.

END OF FILE TN.IIGS.001

###
FILE: TN.IIGS.002
###

Apple II
Technical Notes

Developer Technical Support

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

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

END OF FILE TN.IIGS.002

```
#####################################################################
### FILE: TN.IIGS.003
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#3: Window Information Bar Use

Revised by: Dan Oliver — November 1988
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.

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.

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 call 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. The direct page and data bank are not defined and should be considered unknown.

The Window Manager passes your InfoDefProc the following information:

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

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 the 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 Macintosh 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 currently will not allow controls it creates in an information bar. In this case, NewControl would be using a port that is not in your window's port, namely the Window Manager's port.)

Further Reference

- Apple IIGS Toolbox Reference, Volumes 1 & 2

END OF FILE TN.IIGS.003

##
FILE: TN.IIGS.004
##

Apple II
Technical Notes

Developer Technical Support

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

Revised by: Dan Oliver November 1988
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.

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 catagory. 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, I will assume we 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 will be 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 will 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.

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.

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, $0080 if in 640 mode.
;
BoundsRect    equ    8                  ;Offsets in port record from QuickDraw
document.
PortRect      equ    16
;
;
              pha                       ;Space for result.
              pha
              _FrontWindow              ;Start with the top most window, this
assumes
              bra    enter              ;there are no invisible windows ahead of the
                                        ;active window in the window list.
;
;
loop          ldy    #BoundsRect+2
              lda    [window],y         ;Get window's starting horizontal position.
              eor    #$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

END OF FILE TN.IIGS.004

###
FILE: TN.IIGS.005
###

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#5: Window and Menu Titles

Revised by: Matt Deatherage November 1988
Written by: Dan Oliver October 1986

This Technical Note discusses spacing for both window and menu titles.

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.

END OF FILE TN.IIGS.005

```
#####################################################################
### FILE: TN.IIGS.006
#####################################################################
```

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

- o Apple IIGS Toolbox Reference, Volume 2
- o System Software 5.0 documentation (APDA)

END OF FILE TN.IIGS.006

###
FILE: TN.IIGS.007
###

Apple II
Technical Notes

Developer Technical Support

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.

END OF FILE TN.IIGS.007

##
FILE: TN.IIGS.008
##

Apple II
Technical Notes

Developer Technical Support

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
o Apple Numerics Manual

END OF FILE TN.IIGS.008

##
FILE: TN.IIGS.009
##

Apple II
Technical Notes

Developer Technical Support

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 K ohms to 9 K ohms.

Pin 1, the A/D input, presents a dynamic load to the source, drawing at 10 K ohms 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 K ohms 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

END OF FILE TN.IIGS.009

##
FILE: TN.IIGS.010
##

Apple II
Technical Notes

Developer Technical Support

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
o Apple IIGS Toolbox Reference, Volume 2

END OF FILE TN.IIGS.010

##
FILE: TN.IIGS.011
##

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

END OF FILE TN.IIGS.011

```
#####################################################################
### FILE: TN.IIGS.012
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#12: Tool Set Interdependencies

Revised by: Matt Deatherage July 1989
Written by: Jim Merritt April 1987

This Technical Note lists all known interdependencies between system tool sets on the Apple IIGS.
Changes since November 1988: Added System Software 5.0.

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 System Software 5.0 documentation.

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)
    Control Manager         (#16)
    * Menu Manager          (#15)
    * Line Edit             (#20)              For AlertWindow call only
    * Font Manager          (#27)              For AlertWindow call only
    * Resource Manager      (#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)
    * Resource Manager      (#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)
    Control Manager          (#16)
    * 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.
    * Resource Manager       (#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)
    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 and install 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.
```

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.
    QuickDraw II                  (#4)
        QuickDraw II Auxiliary    (#18)
    Event Manager                 (#6)
    Window Manager                (#14)
    Control Manager               (#16)
    Menu Manager                  (#15)
    LineEdit                      (#20)
    Dialog Manager                (#21)
        either
            Sound Tools then      (#8)
            Note Synthesizer      (#25)
        or
            Note Sequencer        (#26)
        MIDI Tools                (#32)
    Standard File Operations      (#23)
    Scrap Manager                 (#22)
    Desk Manager                  (#5)
    List Manager                  (#28)
    Font Manager                  (#27)
    Print Manager                 (#19)
    Text Edit                     (#34)
```

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.

Further Reference

- o Apple IIGS Toolbox Reference, Volumes 1 & 2

END OF FILE TN.IIGS.012

##
FILE: TN.IIGS.013
##

Apple II
Technical Notes

Developer Technical Support

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.

END OF FILE TN.IIGS.013

###
FILE: TN.IIGS.014
###

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#14: Standard File Calls and GrafPort Records

Revised by: Matt Deatherage November 1988
Written by: Guillermo Ortiz June 1987

This Technical Note formerly described how Standard File 1.1 and earlier did not preserve the GrafPort around Standard File calls and recommended that you save and restore the GrafPort around Standard File calls.

Standard File 2.0 fixes a bug present in earlier versions which did not preserve the GrafPort around Standard File calls. You should not have to write a special case to handle this bug since it is reasonable to expect users to be running your program from a current System Disk (Standard File 2.0 is available in System Disks after 1.1).

You can still save and restore the GrafPort around Standard File calls as it will still work, but doing so will increase the size of your code and cause unnecessary overhead during execution.

END OF FILE TN.IIGS.014

###
FILE: TN.IIGS.015
###

Apple II
Technical Notes

Developer Technical Support

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.

The Font Manager cannot scale a font which is larger than 32K, so InstallFont will fail if scaling is required and the desired font exceeds this limit. If the call fails for this reason, it will report 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 Disk 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.

END OF FILE TN.IIGS.015

```
#####################################################################
### FILE: TN.IIGS.016
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

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
- o Apple IIGS Firmware Reference

END OF FILE TN.IIGS.016

###
FILE: TN.IIGS.017
###

Apple II
Technical Notes

Developer Technical Support

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 operating system's QUIT 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

o   Apple IIGS Toolbox Reference, Volume 1

END OF FILE TN.IIGS.017

##
FILE: TN.IIGS.018
##

Apple II
Technical Notes

Developer Technical Support

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

Revised by: Mike Askins, Matt Deatherage & Jim Mensch November 1988
Written by: Mike Askins June 1987

This Technical Note describes how to get the IIGS to pass control to you for SCC interrupt handling when you are not using the serial firmware.

The Apple IIGS serial firmware is a robust environment for almost every serial programming application; however, there may be times when you must go around the firmware to handle things yourself (despite popular belief, these times are the exception rather than the rule). If you want to handle SCC interrupts on the IIGS without using the serial firmware, there are two procedures you may follow.

The first method calls your interrupt handling routine instead of the built-in serial firmware when an interrupt is generated. This routine is always the second one called when an interrupt occurs (AppleTalk is first) and should give your routine plenty of time to handle the interrupt. You must confirm that the serial port was the source of the interrupt. If so, perform your task, clear the carry bit and perform an RTL. If not, set the carry bit and perform an RTL. You enter your routine in full-native mode with an eight-bit accumulator and registers, and you must exit the same way. Your routine must also preserve the data bank register. Implementing this method is easy. To do so, simply use GetVector and SetVector in the Miscellaneous Tools to save the current vector (always a good idea so you can replace it when your interrupt handler is removed) and replace it with a vector to your routine. The reference number for the SCC interrupt vector is $0009.

The second method calls your interrupt handler last. It simply tells one of the serial ports not to claim a generated interrupt. Control then passes through the entire interrupt handling chain to the user interrupt vector at $3FE and $3FF. The procedure for implementing this method follows:

1. Set your bank $00 handling address at the usual place in $3FE and $3FF.
2. Set a system interrupt flag byte, SerFlag. The ROM version determines the method of finding the address of SerFlag. In ROM 2.0 and later, you can get the address with a call to the Miscellaneous Tools GetAddr using a reference number of $000E. Although you should not need to write a special case for earlier ROM versions, you can do so if you wish; the address of SerFlag in ROM 1.0 is $E10104. Refer to the Apple II Miscellaneous Technical Note #7 for information on identifying Apple IIGS ROM versions.

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 will be handling interrupts. The bits for the different ports are as follows:

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

When you are finished handling interrupts from the chosen port (i.e., when you terminate), you should restore the byte to its original value.

Further Reference

o Apple IIGS Toolbox Reference Manual, Volume 1
o Apple IIGS Firmware Reference Manual
o Apple II Miscellaneous Technical Note #7, Apple II Family Identification

END OF FILE TN.IIGS.018

##
FILE: TN.IIGS.019
##

Apple II
Technical Notes

Developer Technical Support

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
o Apple IIGS Toolbox Reference, Volume 2
o Apple IIGS Toolbox Reference Update

END OF FILE TN.IIGS.019

###
FILE: TN.IIGS.020
###

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#20: Catalog of APW Language Numbers

Revised by: Matt Deatherage July 1989
Written by: Jim Merritt August 1987

This Technical Note lists all APW language numbers which Apple II Developer Technical Support has currently assigned, and it discusses a new scheme for future assignments. This Note obsoletes any publications bearing this information with earlier publication dates.
Changes since November 1988: Added Resource Description languages.

Apple II Developer Technical Support assigns and catalogues all official APW language numbers, and effective May 1988, we have a new scheme for these numbers. First, we list the APW languages which do not follow the new scheme; inclusion of a language on this list does not imply the language product exists or ever will exist under APW.

Number	Language Code	Use
$0	PRODOS	Text file (File Type $04)
$1	Text	APW text file
$2	ASM6502	6502 Assembler
$3	ASM65816	65816 Assembler
$4	BASIC	Byte Works BASIC
$5	BWPASCAL	Byte Works Pascal
$6	EXEC	Command file
$7	SMALLC	Byte Works small C
$8	BWC	Byte Works C
$9	LINKED	APW linker command language
$A	CC	APW C
$B	PASCAL	APW Pascal
$C	COMMAND	Byte Works command-processor window
$1E	TMLPASCAL	TML Pascal

Under the new scheme, we define the high byte of the APW language number as a vendor number and the low byte as a language number. To form the APW language number, combine the vendor number with the language number. The following is a list of currently defined vendors and languages; inclusion of a vendor on this list does not imply the vendor is developing, or ever will be developing, any of the language products listed for APW.

Vendor Number	Vendor Name
$0	Apple Computer

$1	The Byte Works
$2	TML Systems
$3	Zedcor
$4	RavenWare

Language Number	Language Name
$2	6502 Assembler
$3	65816 Assembler
$4	BASIC
$A	C
$B	Pascal
$C	Command-processor window
$D	Forth
$E	Small C
$F	Lisp
$10	Modula-2
$11	FORTRAN
$12	Logo
$13	Prolog
$14	COBOL
$15	Resource Description

If, as an Apple Partner or Associate, you need a new language number for a language processor not currently covered on this list or a vendor number, write to:

Apple II Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, M/S 75-3T
Cupertino, CA 95014
ATTN: APW Language Number Administration

Note: Language number assignments are considered provisional until the applicant submits proof of publication of a language processor using the assigned number. Acceptable proof must include a complete specification for the language that the processor recognizes, as well as photocopies of public notices that discuss the terms and details of publication (e.g., newspaper and magazine ads, software reviews, brochures, circulars, electronic mail solicitations, etc.). Unless a developer has made prior arrangements with Apple II Developer Technical Support, we rescind a provisional language number assignment after a period of one calendar year from the date of assignment if a developer does not submit the required proof of publication.

Further Reference

o   Apple IIGS Programmer's Workshop Reference

END OF FILE TN.IIGS.020

##
FILE: TN.IIGS.021
##

Apple II
Technical Notes

Developer Technical Support

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 design of the Fast Processor Interface (FPI) imposes 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, which contains the DMA bank address register ($C037), does 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

o Apple IIGS Hardware Reference

END OF FILE TN.IIGS.021

###
FILE: TN.IIGS.022
###

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#22: Proper Unloading of Dynamic Segments

Revised by: Lou Infeld, Matt Deatherage, Jim Mensch & Eric Soldan — November 1988
Written by: Guillermo Ortiz — October 1987

This Technical Note discusses strategies that programs may use to deal with dynamic segments.

An application which uses dynamic segments must unload them by hand when it needs the memory they occupy since the system does not automatically release this memory. This requirement poses some interesting problems for any developer using dynamic segments to ensure that an application runs within a given memory configuration, and this Technical Note presents the issues and some possible avenues for attacking the problems.

Which Segments to Unload?

Unloading One Segment at a Time

First, we have a global programming problem: what to do when the application needs more memory and cannot get it? The normal impulse, unloading arbitrary dynamic segments until we satisfy the memory need, presents another question: how does the application recognize which segments are loaded but have not participated in the current chain of events? In other words, how can the application be sure it does not have to return control to a segment it wants to unload?

Since the system provides no mechanism to track the chain of active calls, you need to program the application to allow for such bookkeeping. Some alternatives are as follows:

1. Eliminate interdependencies between dynamic segments. Thus, a subroutine in one segment can be sure that code from another, disposable, segment has not called it. For instance, you can often make initialization code fully independent of code in other segments.

2. Write dynamic segments in such a way that there is only one entry point to each one and establish a list of counters in the main static segment; every time the entry point gets control it increments the counter, and it decreases the counter every time it passes back control. If at any given moment the application needs more memory, it can check the list and unload those segments with

empty (zero) counters.

It is possible to imagine more complicated scenarios, such as how to recover if the segments you need to load fragment memory in such a way that there is not enough contiguous memory available to satisfy allocation needs. We leave these possibilities as exercises for the curious reader.

Unloading All Segments at Once

Another technique is to unload all dynamic segments in the main logic of the program. You can accomplish this with one Unload Segment By Number ($0C) call specifying zero for the load segment number. Unloading all dynamic segments is not as inefficient as it sounds because the System Loader does not reload a needed dynamic segment from the corresponding load file if the system has not yet purged it from memory.

How Best to Unload the Segments

Obtaining the Address of a Dynamic Segment

Once we know what we want to unload, we have the more specific problem of deciding which call to use in unloading the segment. One option is, very appropriately, Unload Segment ($0E). However, the APW Assembler may object to references to dynamic segments other than a JSL unless you tell it otherwise.

The APW Assembler uses the DYNCHK directive to allow access to labels or data in dynamic segments without generating errors. With DYNCHK ON, the APW assembler gives an error message if you access any label in a dynamic segment with any instruction other than JSL. Turning DYNCHK OFF allows you to make any reference to dynamic segment labels without error, although this is usually not valid (see below). DYNCHK OFF allows code like the following to assemble properly:

```
    Pushlong    #DynSegAddr    ; push the jump table address
    _UnloadSegment             ; and unload the segment!
```

It is then your responsibility to ensure that this code is not executed unless that dynamic segment is currently loaded. Note that DYNCHK OFF also causes CODECHK OFF, thus APW does not tell you if you accidentally use JSR to address a label in a different (static or dynamic) segment. Checking for this is your responsibility. Also note that there is no equivalent of DYNCHK in the MPW IIGS cross-development system; the default is the equivalent of DYNCHK OFF.

If you choose to leave DYNCHK ON, you have to use other methods to find the address of a label in a dynamic segment. Below is an example which shows a way of getting the address of a dynamic segment which will link correctly with DYNCHK ON. Suppose we have the following code in a static segment:

```
    LABEL    JSL FOO
```

If FOO is in a dynamic segment, then we can obtain the address of FOO at run time using the following code:

```
    LDA LABEL+1
    STA ADDRESS
    LDA LABEL+3
    STA ADDRESS+2
```

The address obtained in this way is the address of the jump table entry for FOO . You can use this address in the Unload Segment call since the System Loader recognizes the address as a jump table address. Note that since the address is a jump table entry, nothing other than passing control to this address is valid. Doing any other kinds of operations (like loading or storing data) at that label causes problems. LDA DynSegAddr does not give you the first byte at DynSegAddr, but rather the first byte of the jump table to DynSegAddr. Also note that good programming practice dictates that all access to dynamic segments occur via JSL instructions.

If you decide which segments to unload using the scheme described in alternative two above, you again have several options. Most of these options depend upon your preferences for the DYNCHK directive. If you choose to turn DYNCHK OFF, you can actually code a table of segment addresses into your program. The System Loader patches the address at the time of execution with jump-table addresses, and you can unload segments that way. If you leave DYNCHK ON, you must obtain the address of each dynamic segment at the time of execution yourself.

We presented an assembly language example of this earlier. The following code segment demonstrates how a function in C can return the address of its location in RAM:

```
char *Seg2Address()
{
    char *address
    asm
    {
        phk
        phk
        lda #Seg2Address
        sta address
        pla
        and 0x00FF
        sta address+2
    }
    return(address);
}
```

Note that the glue in APW C 1.0 for Unload Segment is broken and actually calls Unload Segment By Number. This is fixed in APW C 1.0.1 and later and in all versions of MPW IIGS C.

Using Unload Segment By Number

As mentioned earlier, you can also use Unload Segment By Number ($0C) to unload dynamic segments. This call takes user ID, load file number, and load segment number as parameters, and it is reasonable to assume at this time that the load file number has a value of one for most programs. The trick here is getting to know, in advance, the load segment number of any segment the application may want to dispose of at a specified moment during its execution.

The idea of a table again comes to mind since each dynamic segment could store its own number there the first time its gets control for all the other segments to see. In this case, it is also possible for the application to initialize the table with the segment numbers. An application can determine its own load segment numbers when you compile it since you can specify into which load segment you wish to place any object segment. However, you must exercise care when linking to preserve this order.

Unloading Segments Before Running Out of Memory

If all the dynamic segments are independent modules, it is possible to implement a second approach: instead of waiting until the application runs out of memory to begin unloading segments, unload each segment immediately after it returns control to the code that called it. With this approach, you cut the application's overhead and know that at any given moment, you have the maximum amount of memory available. The System Loader introduces some extra overhead of its own of course, but if the segment is still in memory, it should not be too much. The following code example illustrates this technique:

```
    JSL    DynSegAddr          ; call the dynamic segment
                               ; (it does its work before returning)
    Pushlong    #DynSegAddr    ; push the jump table address
    _UnloadSegment             ; and unload the segment!
```

Further Reference
- Apple IIGS Programmer's Workshop Assembler Reference
- ProDOS 16 Technical Reference
- GS/OS Reference, Volume 2

END OF FILE TN.IIGS.022

##
FILE: TN.IIGS.023
##

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.

END OF FILE TN.IIGS.023

```
#####################################################################
### FILE: TN.IIGS.024
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#24: Apple IIGS Toolbox Reference Updates

Revised by: Matt Deatherage November 1988
Written by: Rilla Reynolds October 1987

This Technical Note formerly documented 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.

The information formerly contained in this Note is now documented in the Apple IIGS Toolbox Reference Update beta draft (Product Number K2B005) issued in September 1988 and available from the Apple Programmer's and Developer's Association (APDA). This manual draft contains changes to the Toolbox since the original manuals were compiled, facts about the Toolbox which were not originally covered in the manuals, and corrections to the manuals. This draft includes the previously undocumented Audio Compression and Expansion, Note Synthesizer, Note Sequencer, and Midi Tools Tool Sets.

Further Reference:
- o Apple IIGS Toolbox Reference, Volumes 1 & 2
- o Apple IIGS Toolbox Reference Update

END OF FILE TN.IIGS.024

```
#####################################################################
### FILE: TN.IIGS.025
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#25: Apple IIGS Firmware Reference Updates

Revised by: Jim Luther May 1989
Written by: Rilla Reynolds October 1987

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 November 1988: Added corrections to Chapter 5, pages 94, 105, & 106, on the serial-port firmware and changed the diagram in Chapter 7, page 140, on the SmartPort firmware.

Contents

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

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(R), 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 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 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."

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.

Page 122, Statcode = $00: Change the function of bit 0 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 Apple II SmartPort Technical Note #4, SmartPort Device Types.

Page 125, SmartPort driver status: This section should read: "A status call with a unit number of $00 and a status code of $00 is a request to return the status of the SmartPort driver. This function returns the number of devices as well as the current interrupt status. Devices should return $00 in the reserved bytes and exit with a transfer count of $0008. The format of the status list returned is as follows:

```
STATLIST    Byte 0        Number of devices
            Byte 1        Reserved
            Byte 2        $00           Vendor Unknown
                          $01           Apple
                          $02...$FF     Vendor Unique
            Byte 3        Reserved
            Byte 4        Reserved
            Byte 5        Reserved
            Byte 6        Reserved
            Byte 7        Reserved
```

The number of devices field is a 1-byte field indicating the total number of devices connected to the slot or port. This number will always be in the range 0 to 127.

Vendors must request a Vendor ID Assignment from Apple II Developer Technical Support before using a specific value in byte two.

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      |F|D|A|9|T|S|S|F|A|D|A|F|1       |F|D|A|A|S|699     |4|D|A|F|
|5-Nibble|F|5|A|6|r|e|i|o|d|E|A|F|5-Nibble|F|5|A|D|e|GCR     | |E|A|F|
|SelfSync| | | | |a|c|d|r|r| | | |SelfSync| | | | |c|Nibbles |C| | | |
|Fields  | | | | |c|t|e|m|s| | | |Fields  | | | | |t|Fields  |h| | | |
|        | | | | |k|o| |a|L| | | |        | | | | |o|        |e| | | |
|        | | | | | |r| |t|R| | | |        | | | | |r|        |c| | | |
|        | | | | | | | | |C| | | |        | | | | | |        |k| | | |
|        | | | | | | | | | | | | |        | | | | | |        |s| | | |
|        | | | | | | | | | | | | |        | | | | | |        |u| | | |
|        | | | | | | | | | | | | |        | | | | | |        |m| | | |
```

A SelfSync Field is four 20 microsecond selfsync nibbles written as a sequence of five 16 microsecond 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 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 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:

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

- o 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."

Further Reference:

- o Apple IIGS Firmware Reference

END OF FILE TN.IIGS.025

```
#####################################################################
### FILE: TN.IIGS.026
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#26: ROM Revision Summary

Revised by: Matt Deatherage — September 1989
Written by: Rilla Reynolds — October 1987

This Technical Note summarizes revisions to the Apple IIGS ROM.
Changes since November 1988: Revised to cover ROM 3.

Apple currently supports two configurations of the Apple IIGS ROM, ROM 1 and ROM 3. In August 1989, Apple IIGS computers began shipping with a 256K ROM, referred to as version 3 or ROM 3 (ROM 2 was skipped since there was already enough confusion about the first version, ROM 0, and the second version, ROM 1). System Software continues to support ROM 1, but it no longer supports ROM 0. Authorized Apple dealers can upgrade older systems (i.e., machines with serial numbers lower than E704...) to ROM 1 upon request.

ROM 1 requires System Software 2.0 or later, while ROM 3 requires System Software 5.0 or later. Although applications may work using older system software releases, they may not function properly due to the coordination of system software and ROM revisions.

Changes from ROM 0 to ROM 1

ADB

- Absolute ADB devices are now supported correctly.
- ADB fatal system error code is now $0911 instead of $0400.
- ADBReset routine now delays about 160 microseconds before reading the buttons.
- ADBStatus TRUE is now $FFFF instead of $0001.
- All ADB error codes now include the tool number.
- SRQrmv no longer crashes when you make the call with a command pending.

AppleDisk 3.5

- AppleDisk 3.5 Macintosh block reads and writes now work as documented.
- Extended status call now returns bit 0 = 1 if AppleDisk 3.5 media has been switched since the last READ, WRITE, or FORMAT.
- New AppleDisk 3.5 status calls have been implemented to get internal variable and work buffer starting addresses.

AppleTalk

- o Link Access Protocol (LAP) inter-packet gap now handles added SCC delay.
- o Name Binding Protocol (NBP) now considers uppercase and lowercase characters identical.
- o A nonexistent protocol no longer hangs the dispatcher.

Desk Manager

- o SaveScreen and RestoreScreen now work.

Event Manager

- o Now auto-key events are not posted in the queue unless the queue is empty.
- o EMStartUp and EMShutDown code has been optimized.
- o Event Manager now returns an error instead of crashing when there is an attempt to post an invalid event.

Integer Math

New Changes:
- o Optimized the multiply routine.

RAM patches moved to ROM:
- o Changes to FixMul, FixRatio, and SDivide.
- o SDivide recovers from a divide by zero operation.
- o New calls: FracMul, FixDiv, FracDiv, FixRound, FracSqrt, FracCos, FracSin, FixATan2, HiWord, LoWord, Long2Fix, Fix2Long, Fix2Frac, Frac2Fix, Fix2X, Frac2X, X2Fix, X2Frac.

Memory Manager

- o Optimized Purge and Compact for banks 0 and 1 and moved from RAM to ROM.
- o RAM patches and enhancements moved to ROM.
- o RAMdisk now returns bytes transferred count on DIB call.
- o SetHandleSize makes a handle temporarily unpurgeable while changing handle size.

Miscellaneous Tools

RAM patches and enhancements moved to ROM:
- o AbsClamp fixes.
- o Battery RAM routines work if data bank is set to a bank other than bank data is in.
- o Firmware entry calls now return processor status in high byte instead of low byte.
- o GetAddr with ref number $000E returns SerFlag address for SCC interrupts (useful if not using serial firmware).
- o ID manager can reuse discarded IDs.
- o Keyboard interrupts now enable VBL interrupts.
- o Munger now works with 1-char strings and returns with A=0.
- o New SysBeep call.
- o PackBytes and UnpackBytes return with A=0.
- o ReadBParam and ReadBRAM error codes corrected.
- o WriteBParam and WriteBRAM do not return error codes (this is a documentation change).
- o WriteTimeHex Bad Parameter error code is now $31.

Monitor

- o 80-column screens maintained if break occurs and Pascal protocol in effect.
- o AppleSoft tabbing in 80-column mode now works correctly.
- o Control Panel's Maximum RAM Disk Size increased to 8128K instead of 4096K.
- o Firmware version number returned is $1 instead of $0.
- o Interrupts now disabled during paddle read routines.
- o Interrupts re-enabled after fatal system error (for debug DAs).
- o Mouse clamps with positive minimum and negative maximum works (e.g., $6000 min, $8000 max).
- o New monitor command, pound sign (#), installs monitor entry and memory peeker classic desk accessories (unless already installed), accessible via the Control Panel. Reinstalled automatically on reset; disabled by power off only.
- o New monitor command, Control-N, clears m, e, and x bits for native mode. (Control-R still switches to 8-bit, emulation mode.)
- o RESET entry point at $00FA62 sets state register to $0C and shadow register to $08.
- o Shadowing of the Super Hi-Res area in Bank 1 is no longer enabled automatically.
- o WAIT routine now always exits with C=1.

QuickDraw II

RAM patches and enhancements moved to ROM:
- o 640-mode pen masks now work when portRect origin not a multiple of 8.
- o Arcs, ovals, and round rects can be drawn across bank boundaries.
- o Changes to round drawing routines: PPToPort, GetFontLore, GetROMFont, and InflateTextBuffer.
- o Current bank bytes 100...106 no longer modified by scaling and mapping calls.
- o FontFlags 1 and 2 added for pen width and color control.
- o FramePoly returns with A=0.
- o GetPort returns all four bytes of GrafPort.
- o HideCursor and ShowCursor work correctly with obscured cursor.
- o MapRgn now works on rectangular regions.
- o Pixel painting routines support QuickDraw Auxiliary Tool Set stretching and shrinking.
- o PPToPort now clips correctly to the current portRect.
- o QDStartUp and QDShutDown save and restore the scan line interrupt vector.
- o RectInRgn bug fixed.
- o ScrollRect works when the ClipRgn and VisRgn are not rectangular.
- o SetSysFont works.
- o StdPixels now returns with A=0 if the pen is not visible.
- o Text underline bug fixed.
- o TextBounds works.

New QuickDraw changes:
- o Busy flag now maintained correctly by ClosePort, OffsetRgn, InsetRgn, KillPoly, FillRect, FrameOval, PaintOval, EraseOval, InvertOval, FillOval, FrameArc, PaintArc, EraseArc, InvertArc, FillArc, FrameRRect, PaintRRect, EraseRRect, InvertRRect, and FillRRect.
- o Cursor appears in correct Super Hi-Res mode as determined by the low byte's bit 7 (320/640) of the MasterSCB.

SANE

- o Elems now can be called from any part of memory.
- o HALT exception jumping through the incorrect vector fixed.
- o Integer overflow during conversion reported.
- o STATUS call moved to ROM.

Scheduler

- o Scheduler now accepts a flush function call.
- o Task-handling RAM patch (on System Disk 1.0 and later) moved to ROM.

Serial I/O

- o First character after an XON is no longer trashed when buffering is not enabled.
- o If serial mode bit 17 = 1, parity and framing error suppression are defeated.
- o Parity, baud, and data format commands work with buffering.
- o STATUS call will not report that a character is ready if the character arrives with a parity or framing error.
- o STATUS call works correctly with XON/XOFF protocol.

SmartPort

- o PR#5, following a PR#5 with I/O error (i.e., no disk in drive), now boots as expected.
- o SmartPort manipulates only Slot 6 motor on detect so the IWM can run in fast mode.

Sound

- o Fixed bug in FFStopSound call.
- o Fixed low-level RAM read/write bug.
- o Interrupts are disabled when the internal bell is active.
- o Interrupts no longer need to be disabled when accessing sound RAM.
- o New sound diagnostics with the following error codes: $0C001 = failed RAM data test, $0C002 = RAM address test, $0C003 = register data test, and $0C004 = control register test.
- o Sound Manager RAM patches and enhancements moved to ROM.

Text Tools

RAM patches moved to ROM:
- o RAM patches moved to ROM for Writing and ErrorWriting routines.
- o TextInit Illegal device error now is in 16-bit mode instead of 8.

Tool Locator

- o Optimized tool dispatcher.
- o ROM tools present on a memory expansion card are installed.

Changes from ROM 1 to ROM 3

ROM 3 is 256K (double the size of ROM 1) and contains several tools which do not exist in ROM 1. The patch file TS3 fixes known bugs in ROM 3 which were discovered after it was frozen. ROM 3 tools are basically System Software 5.0 tools, and the System Software 5.0 documentation covers these tools in detail. This Note only documents non-tool changes.

AppleDisk 3.5 and SmartPort

- Use new routines for all block reads to fast RAM to eliminate double buffering.
- The extended DIB status call returns the device subtype byte $C1.
- Fixed anomalies described in SmartPort Technical Note #6, Apple IIGS SmartPort Errata.
- Fixed a ROM 1 bug that caused Write Protected to be returned with higher priority than Device Offline for the ProDOS STATUS call.

AppleTalk

- AppleTalk moved to slots 1 and 2 from slot 7.

Control Panel CDA

- The original Options menu is now the Keyboard menu and does not contain mouse parameters.
- A new Mouse menu is present. The new keyboard microcontroller allows finer control of mouse tracking, so a selection procedure better than yes or no is present. Parameters are also available to set the keyboard mouse feature, which allows the numeric keypad to emulate a mouse.
- Added an option to resize the RAM disk on the next reset in the RAM Disk menu. This option resets to No after one reboot and resizing so the RAM disk is not accidently reformatted on every boot thereafter.
- If slot 7 is set to AppleTalk, the Control Panel displays a warning if neither slot 1 nor slot 2 is similarly set.
- The Printer Port and Modem Port menus now display only those parameters that may be changed if AppleTalk is the selection for those ports.
- The RAM disk no longer has minimum and maximum settings, but rather one RAM disk size setting.

Monitor

- Enhanced memory searching commands to automatically cross bank boundaries.
- Added Step and Trace debugging functions.
- Now provide vectors for the same functionality as the GS/OS System Service calls MEMORY_MOVER, DYN_SLOT_ARBITER and SET_SYS_SPEED in bank $E1.
- Now resize the RAM disk when the system is rebooted with the Control-Open Apple-Shift-Reset key combination.
- Handle text page 2 shadowing and power-up bits in the new CYA chip.
- Flash the border if the sound volume is set to zero and a beep is necessary.
- In ROM 1 and earlier, the Miscellaneous Tools mouse firmware called the 8-bit mouse routines in the $C400 space to do the work. In ROM 3, the 8-bit routines call the 16-bit routines to read the hardware. This change effectively means those programs which use

16-bit mouse calls (including desktop applications through the Event Manager) may use the mouse when slot 4 is set to Your Card.
- Slots 1 and 2 may now be set to Printer, Modem, AppleTalk, or Your Card. With System Software 5.0, slot 7 does not need to be set to AppleTalk to use an AppleTalk network, although one can do it for compatibility. There is no transparent printing firmware in slot 7.
- The Alternate Display Mode CDA no longer sets the system to fast speed when normal speed is selected in the Control Panel.
- Added a new command, {val}=V, to set the video screen display I/O switches when resuming a program.
- Control-T command now works as a toggle--executing it once changes to text mode, but now executing it again switches back to the previous video mode You may change this saved video mode with the =V command.
- Battery RAM value $59 now controls the presence of the Visit Monitor and Memory Peeker CDAs. If this byte has the high bit set at boot time, the CDAs are automatically installed.
- The Monitor and Memory Peeker both allow the use of Control-X to terminate a long display (i.e., a handle list or memory dump).

Serial I/O

- XON and XOFF are no longer sent with the high bit set when buffering is enabled.
- Terminal mode cursor is more consistent with the rest of the system.
- Extended Interface calls now return errors in the carry and the accumulator.

Toolbox

The following tools are now in ROM:

- Window Manager
- Menu Manager
- Control Manager
- Line Edit
- Dialog Manager
- Scrap Manager
- Font Manager
- List Manager

Further Reference

- Apple IIGS Firmware Reference
- Apple IIGS Toolbox Reference
- Apple IIGS Technical Note #52, Loading and Special Memory
- SmartPort Technical Note #6, Apple IIGS SmartPort Errata

END OF FILE TN.IIGS.026

```
#####################################################################
### FILE: TN.IIGS.027
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#27:   Graphics Image File Formats

Revised by: Matt Deatherage   November 1988
Written by: Steve Glass, Eagle Berns, Art Cabral,
               Pete McDonald & Rilla Reynolds   October 1987

This Technical Note formerly described the file formats for Apple IIGS graphics image files. File formats are now documented in Apple II File Type Notes under corresponding file types and auxiliary types:

```
File Type $C0
    Auxiliary Type $0000    "PaintWorks" Packed Format
    Auxiliary Type $0001    PackBytes Packed Format
    Auxiliary Type $0002    "Apple Preferred" Packed Format

File Type $C1
    Auxiliary Type $0000    32K unpacked picture image
    Auxiliary Type $0001    Unpacked QuickDraw II picture
```

Further Reference
o   Apple II File Type Notes

END OF FILE TN.IIGS.027

```
#####################################################################
### FILE: TN.IIGS.028
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#28: Interface Card Design Guidelines

Revised by: Matt Deatherage November 1988
Written by: Cameron Birse October 1987

This Technical Note describes suggested dimensions for interface cards for the Apple IIGS and Apple IIe upgraded sytems.

```
|---------------------7.00"---------------------|
_________________________________________________  __   __
|                                               | |     |
|    The 7" dimension is specified for slots    | |     |
|    1-3 because of the optional fan which      | |     |
|    mounts on the power supply.                | |     |
|                                               |2.75"|
|                                               | |     |
|                                               | |  3.05"
|                  SLOTS 1 - 3                  | |     |
|                                               | |     |
|____________________________                 __|__     |
                             |__________________|_|______
                             |-------2.950"-----|-|-.375"
```

```
     |------------------------10.00"----------------------------------|
     |-----2.25"----|----------------------7.75"----------------------|
     |              ___________________________________________________  __   __
     |         _ ~                                                    | |     |
     |    _ ~                                                         | |     |
 ____ _ ~                                                             | |     |
 |   |                                                                | |     |
 |   |                                                                |2.75"|
 |   |                                                                | |     |
2.20"|                                                                | 3.05"
 |   |                           SLOTS 4 - 7                          | |     |
 |   |                                                                | |     |
 __  |_____________________________________________             ___|__     |
                                                    |_________________|_|______
                                                    |-------2.950"-----|-|-.375"
```

```
      |-------------------------10.00"-----------------------------------|
      |-----2.25"----|----------------------7.75"-----------------------|
      |              ____________________________________________________ __
      |          _ ~                                                     | |
      |     _ ~                                                          | |
 ____ _ ~                                                                | |
 |    |                                                                  | |
 |    |                                                                  |2.75"
 |    |                                                                  | |
2.20"|                                                                   | |
 |    |                          SLOTS 4 - 7                             | |
 |    |                                                                  | |
 ____|_______                ________________   _______________________|_____
.35" |-.750-|                |                |  |                 .630" -|
 ____|_______|________________|                |__|_________________________|__
     |--------3.02"----------|                |  |                         |
     |--------------------6.385"--------------|--|-----------3.215"---------|
                                             .40"
```

END OF FILE TN.IIGS.028

\###
\### FILE: TN.IIGS.029
\###

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#29: Monochrome High-Resolution Mode

Revised by: Rilla Reynolds November 1988
Written by: Rilla Reynolds October 1987

This Technical Note discusses a 280 x 192 monochrome high-resolution mode available on the Apple IIGS and useful for clarifying some graphics.

You can select a 280 x 192 monochrome high-resolution mode on the Apple IIGS with the following steps:

1. Select Monochrome and 40-column from the Control Panel (which sets the 40-column soft switch and bit 5 in $C029).
2. Select Hi-Res graphics mode (which sets GR and HIRES soft switches).
3. Read from to write to $C05E (AN3).

To deselect the mode, read from or write to $C05F.

A monochrome double high-resolution mode also exists on the IIGS, and you follow the same procedure outlined above to access it.

You can use the monochrome mode to display sharper graphics-mode text and fine lines for applications which do not require color. Note that Applesoft BASIC also supports the monochrome video mode.

The soft switches you must access in software to enable the monochrome high-resolution mode are as follows:

GR	$C050
HIRES	$C057
40COL	$C00C (for monochrome double hi-res, use 80COL at $C00D)
AN3 OFF	$C05E

In addition, you must set bit 5 of the register at $C029, and you must use a read-modify-write sequence since $C029 is a multi-function register.

You can manipulate all of the soft switches listed above from the IIGS Monitor, except 40COL.

\### END OF FILE TN.IIGS.029

```
########################################################################
### FILE: TN.IIGS.030
########################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#30: Apple IIGS Hardware Reference Updates

Revised by: Rilla Reynolds November 1988
Written by: Rilla Reynolds October 1987

This Technical Note includes updates to the July 1987 edition of the Apple IIGS Hardware Reference, published by Addison-Wesley (Part Number 030-3120-A). 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.

Chapter 3: Memory

Page 36, Table 3-2, Bits in the State Register: bit 2, value 1 description should read, "LCBNK2: If this bit is 1, language-card RAM bank 2 is selected."

Chapter 6: The Apple Desktop Bus

Page 130, Table 6-9, Command byte syntax: The first row in the table should read:

```
x     x     x     x     0    0    0    0    Send Reset
```

and not

```
A(3)  A(2)  A(1)  A(0)  0    0    0    0    Device Reset
```

Page 131, Device Reset: Replace "Device Reset" with "Send Reset." The paragraph should be: "When a device receives a Send Reset command, it will clear all pending operations and data, and will initialize to the power-on state. The Send Reset command is not device-specific; it is sent to all devices simultaneously."

Pages 138-139, Collision detection: The fourth sentence in the last paragraph should be: "By using the Listen register 3 command, the host can move the device with the activator pressed."

Chapter 7: Built-in I/O Ports and Clock

Page 146, Table 7-3, Disk-port soft switches:
$C0E8 Drive disabled

```
$C0E9    Drive enabled
$C0EA    Drive 1 select
$C0EB    Drive 2 select
```

In addition to the corrections listed for Table 7-3, the reference to "spindle motor switches" in the paragraph following the table should be replaced with "drive enable switches."

The following text and table should also be added:

The drive enable switches and the drive select switches control the state of the disk port signals DR1 and DR2. The following table shows the relationship between these.

Soft Switches				Disk Port Signals	
$C0E8	$C0E9	$C0EA	$C0EB	DR1	DR2
1	o	o	o	0	0
o	1	1	o	1	0
o	1	o	1	0	1

1 = asserted state   0 = negated state   o = do not care

Page 147, The Mode register: The IWM Mode register is a write-only register, so disregard the advice to use only a read-modify-write instruction sequence when manipulating bits.

Pages 147-8, Table 7-5, Bits in the Mode register: Switch the given values and descriptions for bits 1, 2, and 4 as follows:

Bit	Value	Description
4	0	7-MHz read-clock speed selected. Set to 0 for all Apple IIGS disk accesses.
	1	8-MHz read-clock speed selected.
2	0	1-second timer selected. When the current disk drive is deselected, the drive will remain enabled for 1 second if this bit is set.
	1	1-second timer is not selected.
1	0	Synchronous handshake protocol selected; for 5.25-inch Apple disk drives.
	1	Asynchronous handshake protocol selected; for all except 5.25-inch Apple disk drives.

Chapter 8: I/O Expansion Slots

Page 167, Direct memory access: DMA bank register location is $C037.

Further Reference:
o   Apple IIGS Hardware Reference

END OF FILE TN.IIGS.030

```
#####################################################################
### FILE: TN.IIGS.031
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#31: Redirecting Output in APW C

Revised by: Guillermo Ortiz November 1988
Written by: Guillermo Ortiz November 1987

This Technical Note presents a sample program which shows how to send output to different devices under the Apple Programmer's Workshop (APW) shell.

Many programmers find the ability to redirect output an expecially useful feature. The following is a sample C program which allows this redirection through an APW shell command. Note that this is not applicable to MPW IIGS C since it is not part of the APW environment.

```
/*
redirect.c
Testing the shell REDIRECT command within APW C
Demonstrates how to send the output to different devices,
a disk file, the printer, and then back to the screen
last modified by Guillermo Ortiz 09/21/87

NOTE: This program checks no errors whatsoever. It expects to
be able to open the file with no problems and expects the
printer to be readily available.

Also remember that for this test to work the file has to be of
the type 'EXE' (executable from the shell only.)
*/

#include <types.h>
#include <misctool.h>
#include <stdio.h>
#include <shell.h>
#include <string.h>

char timestrg[20];           /* string to store the ascii time */
char myfile[80];             /* string to store the filename */
char str[80];                /* dummy string */
int dev=0x0001;              /* standard output */
int app=0x0000;              /* app=0 file is deleted, other will append */

PrintToFile()
{
 printf("Please enter the output filename: \n");
 gets(myfile);
 if (strlen(myfile)==0)
```

```
        {
       printf("Error in entering the filename, quit.\n");
       exit(0);
     }

      /* REDIRECT call requires pascal string */
      c2pstr(myfile);

     /* use the REDIRECT shell command to redirect the output to the file */
      REDIRECT(dev, app, myfile);

      /* now print a few lines of text */
      printf("This is my first line of text.\n");
      printf("And this is the second line.\n");
      printf("Finally the third and last line of text.\n");

     }

     PrintToPrinter()
     {
      /* now redirect to output to the .PRINTER. */
      REDIRECT(dev, app, "\010.PRINTER.");

      printf("We should now be going to the printer.\n");
      ReadAsciiTime(timestrg);
      printf ("The time now is %s\n",timestrg);
     }

     BackToScreen()
     {
      /* Last REDIRECT the output back to the screen. */
      REDIRECT(dev, app, "\010.CONSOLE.");

      printf("The testing of REDIRECTing the output is done.\n");
      ReadAsciiTime(timestrg);
      printf ("The time now is %s\n",timestrg);
     }

     main()
     {
      ReadAsciiTime(timestrg);
      printf ("The starting time is %s\n",timestrg);

      PrintToFile();
      PrintToPrinter();
      BackToScreen();
     }
```

Further Reference

- o Apple IIGS Programmer's Workshop Reference
- o Apple IIGS Programmer's Workshop C Reference

END OF FILE TN.IIGS.031

##
FILE: TN.IIGS.032
##

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#32: /INH Line Anomaly

Revised by: Glenn A. Baxter & Rob Moore — November 1988
Written by: Glenn A. Baxter — December 1986

This Technical Note describes a hardware anomaly which affects the use of the /INH line on the Apple IIGS.

The Apple IIGS maps logical addresses in main and auxiliary RAM spaces to physical RAM devices in such a way that using the /INH line can cause bus contention under certain conditions. This Note describes the problem and suggests a solution strategy.

In the Apple IIGS, main memory resides within four physical 64 x 4 DRAMs. Memory is logically mapped into two separate banks of 64K x 8. The logical map of main memory is slightly different than what one might expect. Owing to the demands of new video modes on the IIGS, the DRAMs need a greater amount of time to perform their function. The easiest way to allocate time in a fixed, time-based system is to use a memory interleaving mechanism, and the IIGS implements its video in this fashion.

As a result of this interleaving scheme, the logical map of main and auxiliary memory does not correspond directly to physical DRAMs, but are split in three places. The split looks like the following:

First Physical 64K		Second Physical 64K	
Main Memory	$0000 - $5FFF	Auxiliary Memory	$0000 - $5FFF
Auxiliary Memory	$6000 - $9FFF	Main Memory	$6000 - $9FFF
Main Memory	$A000 - $FFFF	Auxiliary Memory	$A000 - $FFFF

Only part of the first physical bank of RAM is inhibited when /INH is brought low; therefore, the /INH function only works between $0000 - $5FFF and $A000 - $FFFF in main memory and $6000 - $6FFF in auxiliary memory. If a card attempts to inhibit main memory in the range of $6000 - $9FFF or auxiliary memory in the ranges $0000 - $5FFF or $A000 - $FFFF, bus contention results as both the Mega II and the 74HCT245 buffer device attempt to drive the bus simultaneously (which can damage the Mega II).

Because earlier Apple II systems do not arrange their physical memory as described above, cards which use the /INH line may be compatible with the Apple][+ and IIe, but not with the IIGS. To be compatible with all Apple II systems, a card should include an address mask that will prevent /INH from going low when the address in in the sensitive ranges of main or auxiliary memory. The following logic equation represents an appropriate blocking signal for main memory inhibition:

```
    BLOCK    =    /A15    *    A14    *    A13    ;BLOCK $6000-$7FFF
             +    A15    *    /A14    *    /A13   ;BLOCK $8000-$9FFF
```

END OF FILE TN.IIGS.032

```
#####################################################################
### FILE: TN.IIGS.033
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#33: ERRORDEATH Macro

Revised by: Jim Mensch & Matt Deatherage November 1988
Written by: Allan Bell, Apple Australia & Jim Merritt December 1987

This Technical Note presents a short macro which an assembly language program can invoke to handle fatal error conditions.

Early versions of Apple-approved sample assembly language code for the Apple IIGS often invoked an APW macro named ERRORDEATH. This macro generated code that was appropriate for handling situations where program execution simply could not proceed due to "fatal" errors, such as a failure to load one or more tools that are required to display more sophisticated error dialogs or the inability to allocate sufficient direct page space for essential tool sets. The macro libraries of prototype APW systems included ERRORDEATH, but the release version does not to promote the use of more sophisticated error handling techniques in commercial software packages. The MPW IIGS release never included ERRORDEATH.

Below are two versions of ERRORDEATH; one is compatible with official standard releases of APW and the other with MPW IIGS. While Apple recommends avoiding the use of ERRORDEATH in software intended for commercial release, we feel the code is still useful for providing minimal error handling capability in prototype code and a brief, yet sophisticated, example of macro construction.

APW Assembler version:

```
        MACRO
&lab        ERRORDEATH &text
&lab        bcc end&syscnt
            pha
            pea x&syscnt|-16
            pea x&syscnt
            ldx #$1503
            jsl $E10000
x&syscnt    dc i1'end&syscnt-x&syscnt-1'
            dc c"&text"
            dc i1'13',i1'13'
            dc c'Error was $'
end&syscnt  anop
        MEND
```

MPW IIGS Assembler version:

```
            MACRO
            ErrorDeath &text
            bcc @EDeathEnd
            pha
            pea @Message>>16
            pea @Message
            ldx #$1503
            jsl $E10000
@Message    dc.B @EDeathEnd-@Message-1
            dc.B  &text
            dc.B  13
            dc.B  'Error Was $'
@EDeathEnd
            MEnd
```

The "active ingredient" in the ERRORDEATH macro is the call to SysFailMgr ($1503), which is made if carry is set at the time control passes to the beginning of the expanded macro code sequence. The APW and MPW IIGS assembler macro expansion mechanisms insert the value represented by the character string argument marker, &text, into the generated code stream and provide SysFailMgr with a pointer to that string. The pseudo-argument, &syscnt, generates unique labels in the positions occupied by the expressions x&syscnt and end&syscnt, which makes it possible to invoke ERRORDEATH more than once during any particular source assembly. In the MPW IIGS version of the macro, the MPW IIGS assembler creates a unique label for any label beginning with the at sign (@), effectively doing the equivalent of the &syscnt in the APW version.

To use ERRORDEATH, simply invoke it after any code sequence or subroutine call that sets the carry when it encounters an error (clears it, otherwise) and leaves an appropriate error code in the accumulator. Note that all ProDOS and Toolbox calls observe this convention. When control passes to the beginning of the ERRORDEATH code sequence, the CPU should be in full-native mode, which means the emulation bit should be clear and the accumulator and index registers should be 16-bits wide). Here is a small code segment which demonstrates invoking the macro:

```
                pushword #21            ; Dialog Manager
                pushword #0             ; Use any version
                _LoadOneTool

    ; If carry is now SET, following macro terminates program execution
    ; with the "sliding Apple" error screen.

    IfWeGoofed    ERRORDEATH 'Cannot load Dialog Manager!'

    ; *** If no error, normal execution continues here ***
```

END OF FILE TN.IIGS.033

```
######################################################################
### FILE: TN.IIGS.034
######################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#34: Low-Level QuickDraw II Routines

Revised by: Keith Rollin, Steven Glass, Matt Deatherage & Eric Soldan — January 1989
Written by: Steven Glass — May 1988

This Technical Note describes the low-level routines which QuickDraw II uses to do much of the work in standard calls and mechanisms for calling these routines and accessing their data.
Changed since November 1988: Expanded the section on "Dealing with the Cursor" and documented a bug in ShieldCursor.

QuickDraw II lets you customize low-level drawing operations by intercepting the "bottleneck procedures." QuickDraw II calls an appropriate "bottleneck proc" every time it receives a call to draw an object, measure text, or deal with pictures. For example, if an application calls PaintOval, QuickDraw II calls StdOval to do the real work, and if an application calls InvertRgn, QuickDraw II calls StdRgn to do the work.

Installing your own bottleneck procedures is a little bit tricky. The QuickDraw II SetStdProcs call accepts a pointer to a 56-byte ($38 hex) record and fills that record with the addresses of the standard bottleneck procedures of QuickDraw II. You may modify this record by replacing those addresses with the addresses of your own custom bottleneck procedures minus one. (QuickDraw II pushes the address on the stack and executes an RTL to it, so the address in the record must point to the byte before the routine.) After installing your own procedures, you use SetGrafProcs to tell QuickDraw II about them. The format of this call is as follows (taken from the E16.QUICKDRAW file in APW):

```
    ostdText      GEQU   $00 ; Pointer - QDProcs -
    ostdLine      GEQU   $04 ; Pointer - QDProcs -
    ostdRect      GEQU   $08 ; Pointer - QDProcs -
    ostdRRect     GEQU   $0C ; Pointer - QDProcs -
    ostdOval      GEQU   $10 ; Pointer - QDProcs -
    ostdArc       GEQU   $14 ; Pointer - QDProcs -
    ostdPoly      GEQU   $18 ; Pointer - QDProcs -
    ostdRgn       GEQU   $1C ; Pointer - QDProcs -
    ostdPixels    GEQU   $20 ; Pointer - QDProcs -
    ostdComment   GEQU   $24 ; Pointer - QDProcs -
    ostdTxMeas    GEQU   $28 ; Pointer - QDProcs -
    ostdTxBnds    GEQU   $2C ; Pointer - QDProcs -
    ostdGetPic    GEQU   $30 ; Pointer - QDProcs -
    ostdPutPic    GEQU   $34 ; Pointer - QDProcs -
```

The following code fragment shows how you might replace the StdRect procedure with your own for a given window:

```
    pha                             ; open a test window
    pha
    PushLong #MWindData             ; standard setup for NewWindow
    _NewWindow
    _SetPort

    PushLong #MyProcs               ; get a record to modify
    _SetStdProcs

    ldy #ostdRect                   ; get the low word of my rectangle routine
    lda #myRect-1                   ; (minus one) and patch it in to the record
    sta myProcs,y
    lda #^myRect                    ; do the same for the high word
    sta myProcs+2,y

    PushLong #MyProcs               ; install the procs
    _SetGrafProcs
```

The interface to bottleneck procedures is different from the interface to other QuickDraw II routines; you do not make calls via the tool dispatcher and you pass most parameters on the direct page and in registers (rather than on the stack). To write your own bottleneck procedures, you have to know where the inputs to each call are kept and how to call the standard procedures from inside your own procedures.

The standard bottleneck procedures are accessed through vectors in bank $E0.

```
    StdText       $E01E04
    StdLine       $E01E08
    StdRect       $E01E0C
    StdRRect      $E01E10
    StdOval       $E01E14
    StdArc        $E01E18
    StdPoly       $E01E1C
    StdRgn        $E01E20
    StdPixels     $E01E24
    StdComment    $E01E28
    StdTxMeas     $E01E2C
    StdTxBnds     $E01E30
    StdGetPic     $E01E34
    StdPutPic     $E01E38
```

When you call any of the standard procedures, the first direct page of QuickDraw II is active. If you pass variables on any direct page other than the first (direct page locations greater than $FF), you can use a simple trick to access them. For example, to access TheFillPat ($10E) without changing the direct page register:

```
    ldx    #$100                    ;offset to second DP
    lda    >$OE,X                   ;gets "DP" location $10E
```

Certain locations on the direct page are always valid:

```
    PortRef      $24
    MaxWidth     $20
    MasterSCB    $08
```

```
    UserID       $0A
```

DrawVerb is usually valid, but not always:

```
    DrawVerb     $38
```

Each of the bottleneck procedures uses the direct page differently.

QuickDraw II has an interesting bug relating to the standard conic bottleneck procedures. If you replace any of the standard procedures with your own, QuickDraw II does not perform some of the setups it normally would before calling the standard conic procedures (stdRRect, stdOval, stdArc). For example, if you replace StdRect with a custom rectangle routine, but leave the other conic pointers alone (as shown in the code fragment above), QuickDraw II will not do all of the normal setups when calling the standard conic routines. To deal with this bug of QuickDraw II, you must patch out the additional bottleneck procedures and set up those direct pages locations yourself, or the results will not be what you expect. The QuickDraw II direct-page variables you must initialize yourself in this instance are bulleted (o) below.

```
StdText
    DrawVerb      $38    Describes the kind of text to draw.  There
                         are three possible values:
                             DrawCharVerb    0
                             DrawTextVerb    1
                             DrawCStrVerb    2
    TextPtr       $DA    If the draw verb is DrawTextVerb or
                         DrawCStrVerb, TextPtr points to the text
                         buffer or C string to draw.
    TextLength    $D8    If the draw verb is DrawTextVerb,
                         TextLength contains the number of bytes in
                         the text buffer.
    CharToDraw    $D6    If the draw verb is DrawCharVerb,
                         CharToDraw contains the character to draw.

StdLine
    Y1            $A6    Starting Y value for the line to draw
    X1            $A8    Starting X value for the line to draw
    Y2            $AA    Ending Y value for the line to draw
    X2            $AB    Ending X value for the line to draw
    Rect2         $AE    Exactly the same thing as Y1, X1, Y2 and
                         X2 in the top, left, bottom, and right of
                         the rectangle

StdRect
    DrawVerb      $38    One of the following five drawing verbs:
                             Frame           0
                             Paint           1
                             Erase           2
                             Invert          3
                             Fill            4
    Rect1         $A6    The rectangle to draw in standard form
                         (top, left, bottom, right)
    TheFillPat    $10E   The pattern to use for the rectangle if
                         the verb is Fill
```

Note: The QuickDraw II Auxiliary SpecialRect call does not use the rectangle bottleneck procedures.

```
StdRRect
    DrawVerb      $38    One of the following five drawing verbs:
                             Frame           0
                             Paint           1
                             Erase           2
                             Invert          3
                             Fill            4
    Rect1         $A6    The boundary rectangle for the round
                         rectangle
    OvalRect      $295   A copy of the boundary rectangle for the
                         round rectangle
    OvalHeight    $208   The oval height for the rounded part of
                         the round rectangle
    OvalWidth     $20A   The oval width for the rounded part of the
                         round rectangle
    o ArcAngle    $D2    Must be 360
    o StartAngle  $D4    Must be zero
    TheFillPat    $10E   The pattern to use for the round rectangle
                         if the verb is Fill

StdOval
    DrawVerb      $38    One of the following five drawing verbs:
                             Frame           0
                             Paint           1
                             Erase           2
                             Invert          3
                             Fill            4
    Rect1         $A6    The boundary rectangle for the oval
    OvalRect      $295   A copy of the boundary rectangle for the
                         oval
    o OvalHeight  $208   Must be the height of the oval
    o OvalWidth   $20A   Must be the width of the oval
    o ArcAngle    $D2    Must be 360
    o StartAngle  $D4    Must be zero
    TheFillPat    $10E   The pattern to use for the oval if the
                         verb is Fill

StdArc
    DrawVerb      $38    One of the following five drawing verbs:
                             Frame           0
                             Paint           1
                             Erase           2
                             Invert          3
                             Fill            4
    Rect1         $A6    The boundary rectangle for the arc
    o OvalWidth   $20A   Must be the width of the boundary
                         rectangle for the arc
    ArcAngle      $D2    The number of degrees the arc will sweep
    StartAngle    $D4    The starting position of the arc
    TheFillPat    $10E   The pattern to use for the arc if the verb
                         is Fill

StdPoly
    DrawVerb      $38    One of the following five drawing verbs:
                             Frame           0
                             Paint           1
                             Erase           2
```

```
                            Invert          3
                            Fill            4
    RgnHandleA    $50    The handle to the polygon data structure
    TheFillPat    $10E   The pattern to use for the polygon if the
                         verb is Fill

StdRgn
    DrawVerb      $38    One of the following five drawing verbs:
                            Frame           0
                            Paint           1
                            Erase           2
                            Invert          3
                            Fill            4
    RgnHandleC    $70    The handle to the region to draw
    TheFillPat    $10E   The pattern to use for the region if the
                         verb is Fill

StdPixels
    SrcLocInfo    $CC    The LocInfo record for the source pixel
                         map
    DestLocInfo   $0C    The LocInfo record for the destination
                         pixel map
    SrcRect       $DC    The source rectangle for the operation in
                         local coordinates for the source pixel map
                         (as described in the source LocInfo
                         record)
    DestRect      $1C    The destination rectangle for the
                         operation in local coordinates for the
                         destination pixel map (as described in the
                         destination  LocInfo record)
    XferMode      $E4    The mode to use for data transfer
    RgnHandleA    $50    The handle to the first region to which
                         drawing is clipped (usually the ClipRgn
                         from the GrafPort)  A NIL handle is not
                         allowed.  To signify no clipping, pass a
                         handle to the WideOpen region, which is
                         defined as 10 bytes:

                         Length      $A       (word)
                         -MaxInt    -$3FFF    (word)
                         -MaxInt    -$3FFF    (word)
                         +MaxInt    +$3FFF    (word)
                         +MaxInt    +$3FFF    (word)

    RgnHandleB    $60    The handle to the second region to which
                         drawing is clipped (usually the VisRgn
                         from the GrafPort)  A NIL handle is not
                         allowed.  To signify no clipping, pass a
                         handle to the WideOpen region.
    RgnHandleC    $70    The handle to the second region to which
                         drawing is clipped (usually the mask
                         region from the CopyPixels or the
                         PaintPixels call)  A NIL handle is not
                         allowed.  To signify no clipping, pass a
                         handle to the WideOpen region.

StdComment
    TheKind       $A6    The kind of input for the comment
```

```
    TheSize        $A8    The number of bytes to put into the
                          picture
    TheHandle      $AA    The data to put into the picture

StdTxMeas
    DrawVerb       $38    Describes the kind of text to draw.  There
                          are three possible values:
                              DrawCharVerb    0
                              DrawTextVerb    1
                              DrawCStrVerb    2
    TextPtr        $DA    If the draw verb is DrawTextVerb or
                          DrawCStrVerb, TextPtr points to the text
                          buffer or C string to draw.
    TextLength     $D8    If the draw verb is DrawTextVerb,
                          TextLength contains the number of bytes in
                          the text buffer.
    CharToDraw     $D6    If the draw verb is DrawCharVerb,
                          CharToDraw contains the character to
                          measure.
    TheWidth       $DE    The resulting width should be put here.

StdTxBnds
    DrawVerb       $38    Describes the kind of text to draw.  There
                          are three possible values:
                              DrawCharVerb    0
                              DrawTextVerb    1
                              DrawCStrVerb    2
    TextPtr        $DA    If the draw verb is DrawTextVerb or
                          DrawCStrVerb, TextPtr points to the text
                          buffer or C string to draw.
    TextLength     $D8    If the draw verb is DrawTextVerb,
                          TextLength contains the number of bytes in
                          the text buffer.
    CharToDraw     $D6    If the draw verb is DrawCharVerb,
                          CharToDraw contains the character to draw.
    RectPtr        $D2    Indicates the address to put the resulting
                          rectangle.

StdGetPic
    This call takes input on the stack rather than the direct page.  This is
    the one standard bottleneck procedure which you call with the direct
    page register set to something other than the direct page of QuickDraw
    II; it is set to a part of the stack.

    Stack Diagram on Entrance to StdGetPic
        Previous Contents
        DataPtr              Pointer to destination buffer
        Count                Integer (unsigned) (bytes to read)
        RTL Address          3 bytes
        -----------------    Top of Stack

    Stack Diagram just before exit from StdGetPic
        Previous Contents
        RTL Address          3 bytes
        -----------------    Top of Stack

StdPutPic
    This call takes input on the stack rather than the direct page; however,
```

unlike StdGetPic, the direct page for QuickDraw II is active when you call this routine.

Stack Diagram on Entrance to StdPutPic

```
Previous Contents
DataPtr             Pointer to source buffer
Count               Integer (unsigned) (bytes to read)
RTL Address         3 bytes
-----------------   Top of Stack
```

Stack Diagram just before exit from StdPutPic

```
Previous Contents
RTL Address         3 bytes
-----------------   Top of Stack
```

Dealing with the Cursor

The cursor can get in your way when you want to draw directly to the screen. QuickDraw II has two low-level routines which help you avoid this problem: ShieldCursor and UnshieldCursor. ShieldCursor tells QuickDraw II to hide the cursor if it intersects the MinRect and to prevent the cursor from moving until you call UnshieldCursor.

There is a bug in ShieldCursor for System Disks 4.0 and earlier. This bug is related to the routine ObscureCursor. When the cursor is obscured, ShieldCursor does not prevent the cursor from moving; therefore, the user is able to move the cursor during a QuickDraw II operation, and this movement may disturb the screen image.

Calls to ShieldCursor must be balanced by calls to UnshieldCursor. You may not call ShieldCursor successively without calling UnshieldCursor after each call to ShieldCursor. There is no error checking, so careless use of these routines will result in an unusable system.

MinRect is the smallest possible rectangle which encloses all the pixels that may be affected by a drawing call. You keep MinRect on the direct page and usually calculate it by intersecting the rectangle of the object you are drawing with the BoundsRect, PortRect, boundary box of the VisRgn, and the boundary box of the ClipRgn. You must set up MinRect yourself.

ShieldCursor also looks at two other fields on the direct page of QuickDraw II. ImageRef is a long word located at $0E. If ImageRef does not point to $E12000, QuickDraw II assumes you are not drawing to the screen, so it does not have to shield the cursor. BoundsRect is a rectangle located at $14, and QuickDraw II uses it to translate MinRect into global coordinates. These values are generally correct, but under the following known circumstance, they are not and ShieldCursor will not function properly:

1. You have just drawn to an off-screen GrafPort with QuickDraw II.
2. You switch to a GrafPort on the screen.
3. You call ShieldCursor.

ImageRef and BoundsRect are not updated until QuickDraw II is actually committed to drawing, thus, these values are still for the off-screen GrafPort in this case, even though you switched to a GrafPort on the screen.

Therefore, when you call ShieldCursor, you have to make sure that these values are current. (If these values are current, ShieldCursor will work correctly, no matter what the circumstances.)

You can find the location of the QuickDraw II direct page with the GetWAP call. For speed reasons, you may not want to make the GetWAP call for each ShieldCursor call. You may wish to get the work area pointer value after starting QuickDraw II and store it for future reference.

Calling ShieldCursor:
1. Set direct page for QuickDraw II.
2. Set MinRect, ImageRef, and BoundsRect.
3. Call ShieldCursor.

Calling UnshieldCursor:
1. Set direct page for QuickDraw II.
2. Call UnshieldCursor.

```
ShieldCursor      $E01E98
    MinRect       $00
    ImageRef      $0E
    BoundsRect    $14

UnshieldCursor    $E01E9C
```

Further Reference

o Apple IIGS Toolbox Reference, Volume 2

END OF FILE TN.IIGS.034

```
#####################################################################
### FILE: TN.IIGS.035
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#35: Printer Driver Specifications

Revised by: Matt Deatherage & Suki Lee — September 1989
Written by: Dan Hitchens — May 1988

This Technical Note describes the routines and internal structures needed to design a printer driver for the Apple IIGS system, and you should reference it in conjunction with the Apple IIGS Toolbox Reference manuals. An overview and associated parameters for each of the printer driver routines are in the Print Manager chapter, and you should reference these for a complete picture.
Changed since March 1989: Added System Software 5.0 calls and the new driver structure.

Printing Modes

There are two printing modes: immediate and deferred.

- o Immediate mode (also known as draft mode), uses the technique of printing immediately. As you make QuickDraw II calls, you immediately generate commands which cause printing to occur. This mode is the fastest form of printing, but it can only print characters in the printer's native mode. (However, the LaserWriter translates the QuickDraw II calls into PostScript calls which can produce high-quality pixelmap images.)

- o Deferred mode (sometimes referred to as spooling), uses the technique of capturing the QuickDraw II calls for each page in a picture file and plays them back at a later time. To produce high-quality pixelmap images, you must use deferred mode because of the memory constraint of not being able to draw a complete page in memory at one time. Due to this limitation, we draw a band (a partial page) at a time. We create a GrafPort which corresponds to the band and play the picture file back, thus causing the saved commands to draw only the images which fall within the band. Once the pixel image for the band is generated, we can send it to the printer one pixel at a time.

File Structure

The user can install new printer drivers into the system by copying a printer driver file into a subdirectory called DRIVERS within the SYSTEM subdirectory. The printer driver file must be of type $BB and have an auxiliary type of $0001.

Print Driver Calls

A printer driver must support the following calls:

PrDefault	$0913	Sets print record to default
PrValidate	$0A13	Validates print record
PrStlDialog	$0B13	Performs a style dialog
PrJobDialog	$0C13	Performs a job dialog
PrPixelMap	$0D13	Prints a pixelmap
PrOpenDoc	$0E13	Opens the document
PrCloseDoc	$0F13	Closes the document
PrOpenPage	$1013	Opens a page
PrClosePage	$1113	Closes a page
PrPicFile	$1213	Prints a picture file
--RESERVED--	$1313	
PrError	$1413	Gets the error value
PrSetError	$1513	Sets the error value
GetDeviceName	$1713	Gets device's name
PrDriverVer	$2313	Gets installed driver version

Printer drivers may support the following calls if they use the new driver structure outlined below:

PrGetPrinterSpecs	$1813	Returns printer type and characteristics
PrGetPageOrientation	$3813	Returns page orientation

Print Driver Entry

- o For older drivers, entry is at the first byte (no offset). For newer (Print Manager 3.0 and later) drivers, the first word is $0000, indicating a new style driver. The next word is a count of how many calls this driver supports. All drivers must support the minimum call set. Additional calls must be supported in the sequence listed (for example, if a driver supports PrGetPageOrientation, it must also support PrGetPrinterSpecs).
- o The content of the x register is calculated beforehand for use as an index to the correct routine (see the example and note the specific ordering of the routines).
- o There are two long return addresses (six bytes) that have been pushed onto the stack. (You must take these addresses into account to access the parameters and to return correctly.)

Example

```
StartOfDriver    START

                 dc i2 '0'                ; new style driver
                 dc i2 '16'

                 jmp (PrDriverList,x)


PrDriverList     dc a4'PrDefault'
                 dc a4'PrValidate'
                 dc a4'PrStlDialog'
                 dc a4'PrJobDialog'
                 dc a4'PrPixelMap'
```

```
dc a4'PrOpenDoc'
dc a4'PrCloseDoc'
dc a4'PrOpenPage'
dc a4'PrClosePage'
dc a4'PrPicFile'
dc a4'InvalidRoutine'
dc a4'PrError'
dc a4'PrSetError'
dc a4'GetDeviceName'
dc a4'PrDriverVer'
dc a4'PrGetPrinterSpecs'
dc a4'PrGetPageOrientation'
```

Print Driver Exit

You should adjust the stack to use RTL instructions followed by any return parameters with the two long return addresses. To accomplish this, you will need to do the following:

- o Eliminate any parameters from the stack which have been passed.
- o Move the long return addresses so that they are before the space for the returned parameters (if any).

Example

Figure 1 diagrams the stack just before leaving the print driver:

```
| Previous Contents
+---------------------
| Results (if any)
+---------------------
| RTL2 (3 bytes)
+---------------------
| RTL1 (3 bytes)
+---------------------
                     <--- Stack Pointer
```

Figure 1-Stack Prior to Exiting the Print Driver

You should do an RTL with the contents of the flags and registers set appropriately. (See the Return from Call section of the "Using The Apple Tools" chapter of the Apple IIGS Toolbox Reference.)

Print Record Structure

Since application programs often need to fiddle with parts of the print record (i.e., the values in the style subrecord), we have defined ways for applications to interpret the print record, and specifically the style subrecord.

iDev, the first word of the printer information subrecord, has two defined values for third-party printer drivers. A value of $8001 indicates a dot-matrix printer while a value of $8003 indicates a laser printer.

A value of $8001 indicates that fields of the style subrecord should be interpreted as they are by the ImageWriter driver, as documented in the Apple IIGS Toolbox Reference. The first seven bits (0-6) of wDev are defined as for the ImageWriter driver. Bits 7-11 are reserved for Apple's use and must be set to zero. Bits 12-15 may be used by third-party printer drivers as necessary; these bits will be set to zero in Apple's drivers.

A value of $8003 indicates that fields of the style subrecord should be interpreted as they are by the LaserWriter driver. The first four bits (0-3) of wDev are defined as for the LaserWriter driver. Bits 4-11 are reserved for Apple's use and must be set to zero. Bits 12-15 may be used by third-party printer drivers as necessary; these bits will be set to zero in Apple's drivers.

If an application wishes to take advantages of specific features of a third-party printer driver, it has to know that it is dealing with that driver. Since all drivers will look pretty much alike, the Print Manager allows you to ask for the name of the currently selected printer driver. An application may make the Print Manager call PMGetPrinterName, which is documented in this Note. The Print Manager will return the name of the currently selected printer in a Pascal (length byte) string. The name returned is the name of the file from which the driver was loaded. If you intend to use this method to identify a driver, you must inform users not to rename the Printer Driver file on the boot disk.

The PMGetPrinterName call is as follows:

```
Note:  This is a Print Manager call, not a Printer Driver call.
       It is the only Print Manager call documented in this Note.
       Printer Drivers do not include this call.

PMGetPrinterName         ($2813)

Description:
    Returns a Pascal string with the file name of the currently selected
    printer driver.

Passed:
    Longspace            LONG        Space for result

Returned:
    NamePointer          LONG        Pointer to a Pascal string of driver
                                     filename


Print Driver Calls

PrDefault                ($0913)

Description:
    Fills the fields of the specified print record with default values for the
    printer.

Passed:
    PrintRecordHandle    LONG        Handle to the print record

Returned:
    None
```

Performs the following:
- o Validates that PrintRecordHandle is a handle and does nothing if not.
- o Determines the default values for the print record either through tables or calculations. The default values should take into account such things as paper size and orientation, print mode, printer type, etc.
- o Copies the default values to the print record specified by the PrintRecordHandle parameter.

PrValidate ($0A13)

Description:
Checks the print record to see that it is valid for the currently installed printer driver.

Passed:

PrintRecordHandle	LONG	Handle to the print record

Returned:

ChangeFlag	WORD	Boolean; TRUE if the record is adjusted

Performs the following:
- o Checks to see if the print record is from this particular driver.
- o If the print record is not from this driver, it uses the default values for this driver.
- o If the print record is from this driver, it makes any changes that might be needed (i.e., style, paper size, etc.).

PrStlDialog ($0B13)

Description:
Performs a style dialog with the user.

Passed:

PrintRecordHandle	LONG	Handle to the print record

Returned:

ConfirmFlag	WORD	Boolean; TRUE if the dialog is confirmed

Performs the following:
- o Conducts a style dialog with the user to determine the page dimensions and other information needed for page setup (the initial settings of the dialog are derived from the print record).
- o If the user confirms the dialog, the information from the dialog is saved in the specified print record, PrValidate is called, and the routine returns TRUE.
- o If the user cancels the dialog, the print record is left unchanged, and the routine returns FALSE.

Note: The following are items typically found in printer style dialogs:

- o Paper Size (US Letter, US Legal, A4 Letter, B5 Letter, International Fanfold)
- o Printing Orientation (Landscape, Portrait)
- o Vertical Sizing (Normal, Intermediate, Condensed)

- o Special Effects:
  - Font Effects (Font Substitution, Smoothing)
  - Reduction or Enlargement
  - Gaps or No Gaps between pages

Every printer style dialog should have an OK button (default) and a Cancel button.

```
PrJobDialog              ($0C13)

Description:
    Performs a job dialog with the user.

Passed:
    PrintRecordHandle    LONG        Handle to the print record

Returned:
    ConfirmFlag          WORD        Boolean; True if the dialog is
                                     confirmed
```

Performs the following:

- o Conducts a job dialog with the user to determine the print quality, range of pages to print, and other specifications. The initial settings are derived from the previous PrJobDialog call (or initial default values) except the page range which is set to ALL, and the number of copies which is set to ONE.
- o If the user confirms the dialog, PrValidate is called, the print record is updated, and the routine returns TRUE.
- o If the user cancels the dialog, the print record is left unchanged, and the routine returns FALSE.

Note: The following are items typically found in printer job dialogs:

- o Print Quality (Best, Faster, Draft, etc.)
- o Color option
- o Pages (All, Range)
- o Copies
- o Paper Source (paper cassette, manual feed)

Every printer job dialog should have an OK button (default) and a Cancel button.

```
PrPixelMap               ($0D13)

Description:
    Prints all or part of the specified pixelmap.

Passed:
    srcLocPtr            LONG        Pointer to the source LocInfo which
                                     contains the pointer to the pixelmap.
    srcRectPtr           LONG        Pointer to the rectangle which
                                     encloses the pixelmap to be printed.
    colorFlag            WORD        Boolean; FALSE if black and white,
                                     TRUE if color.

Returned:
```

None

Performs the following:
- o Calls DevIsItSafe (port driver call) to verify that the port it functioning and it is safe to proceed. If it is not functioning, set the internal error code to $1302 (Port Not On) and return with an error status.
- o Saves the current port.
- o Turns on the watch cursor to signal the user that it will take some time.
- o Clears the internal error code (default, if no errors occur).
- o Gets a new handle for a print record and set it to the defaults by calling PrDefault.
- o If colorFlag is set, sets bit 5 of wdev in prStl of the print record.
- o Do any initialization that might be needed by the driver.
- o Determine the intersection of the two rectangles (rectangle pointed to by srcRectPtr and the pixelmap's boundary rectangle from srcLocPtr) and if there is no intersection, then nothing is to be printed.
- o Print the pixel image which is within the intersection of the two rectangles.
- o Cause a page eject to occur on the printer.
- o Do any clean up that is needed.
- o Turn off the watch cursor by calling InitCursor.
- o Restore the port by calling SetPort.

PrOpenDoc ($0E13)

Description:
This routine initializes the things needed to open a document. In deferred mode, it establishes a GrafPort and makes it the current port for printing.

Passed:

PrintRecordHandle	LONG	Handle to the print record
PrinterPortPtr	LONG	Pointer to the GrafPort, if desired, zero to allocate a new GrafPort

Returned:

PrinterPortPtrRet	LONG	Pointer to the GrafPort if the PrinterPortPtr was zero

Performs the following:
- o Calls DevIsItSafe (port driver call) to verify that the port is functioning and it is safe to proceed.
- o Turns on the watch cursor to signal the user that it will take some time.
- o Validates the print record passed by calling PrValidate.
- o Clears the internal error code (default, if nothing happens).
- o Puts up a dialog indicating that printing is occurring (or preparing to print).
- o If the user needs a GrafPort, create one and internally note that one was created (PrCloseDoc will need to know that one was created here).
- o Initializes parameters (i.e., page number, document number, etc.).
- o If deferred mode, create an initial page list (an array of handles to pictures) for 20 pages (arbitrary number to start).
- o Do other initialization that might be needed to start a print job.

Possible errors:

$1302	Indicates Port Not On

PrCloseDoc ($0F13)

```
Description:
    Closes the GrafPort being used for printing.  For immediate mode, this
    routine ends the printing job.  For deferred mode, this routine ends the
    process allowing the job to be printed.

Passed:
    PrintGrafPortPtr     LONG        Pointer to the GrafPort used for printing

Returned:
    None

Performs the following:
  o  Checks that the last print driver call did not cause a Port Not On error.
     If the error occurred, do nothing and return.
  o  Call ClosePort (port driver call) to close the port.
  o  If the driver allocated a GrafPort in PrOpenDoc, dispose of it.
  o  If in immediate mode, do what is needed to shut things down.
  o  Takes down the information dialog box from PrOpenDoc.


PrOpenPage               ($1013)

Description:
    Begins a new page only if the page falls within the page range specified in
    the job subrecord.

Passed:
    PrintGrafPortPtr     LONG        Pointer to the GrafPort used for printing
    PageFramePtr         LONG        Pointer to the scaling parameter,
                                     zero for none.
Returned:
    None

Performs the following:
  o  Looks at the driver's internal error value, and if an error has occurred,
     it returns without doing anything.
  o  Increments the page number.
  o  Calls SetPort to make the specified port the current port.
  o  Initializes the port and zeroes the boundary rectangle so no actual drawing
     will occur.
  o  If immediate mode, then do the following:
       If this page is to be printed, install immediate mode procedures by
       doing the following:
         o  Create a procedure table (get the standard procedures. from
            SetStdProcs).
         o  Put pointers to your procedures into the table and call the
            QuickDraw II routine SetGrafProcs.  This will cause QuickDraw
            II calls to print instead of writing to the GrafPort.
  o  If deferred mode, then do the following:
       o  If the current page is out of the page range, then return without
          doing anything further.
       o  If the user passes his own PageFramePtr , then get it.
       o  Open a picture by calling OpenPicture and adding its handle to
          the page list array described in PrOpenDoc.
       o  Set the ClipRgn and VisRgn to the sizing framing rectangle
          specified by PageFramePtr , or if none was specified, to the
          default of rPage.
```

PrClosePage ($1113)

Description:
This signals the end of a page.

Passed:

PrintGrafPortPtr	LONG	Pointer to the GrafPort used for printing

Returned:
None

Performs the following:

- o Looks at the driver's internal error value and if a Port Not On error has occurred, it returns without doing anything.
- o If immediate mode, do the following:
- o If the current page is within the range of pages to be printed, then cause a form feed (unless no gap was specified).
- o If deferred mode, do the following:
- o If there was no picture generated, then do nothing and just return.
- o Call SetPort to make the specified port the current port.
- o Do a ClosePicture to close the picture.

PrPicFile ($1213)

Description:
Prints a picture file generated in deferred mode.

Passed:

PrintRecordHandle	LONG	Handle to the print record
PrintGrafPortPtr	LONG	Pointer to the GrafPort used for printing
StatusRecPtr	LONG	Pointer to the printer status record

Returned:
None

Performs the following:

- o Looks at the driver's internal error value and if a Port Not On error has occurred, it returns without doing anything.
- o If immediate mode, return without doing anything.
- o If deferred mode, then do the following:
- o If the error code is not zero (errors) then dispose of everything.
- o Put up an information dialog indicating that printing is occurring.
- o If PrintGrafPortPtr is NIL, create one and make a note of it.
- o Call OpenPort to make the GrafPort the current port.
- o If StatusRecPtr is NIL, use an internal one.
- o Initialize the status record and the number of copies counter.
- o If the idle proc in the print record is NIL, point to an internal one.
- o Do The Following For Each Copy:
  - o Calculate the number of bands it will take to print one page and initialize the page counter.
  - o Do The Following For Each Page:
    - o Call the idle procedure routine and initialize the band counter.
    - o Get the handle to the picture associated with the current page.
    - o Set the dirty flag in the status record to FALSE.

```
            o  If manual paper feed, put up a dialog and wait for
               a response.
            o  Do The Following For Each Band:
                  o  Call the idle procedure.
                  o  Calculate the band rectangle and update
                     icurband with the current band number.
                  o  Call the idle proc again.
                  o  Set the imaging flag in the status record to TRUE.
                  o  Call InitPort to reinitialize the port.
                  o  Adjust fields in the port to cause drawing into
                     the band buffer.
                  o  Adjust fields in the location information field
                     of the status record and calculate the sizing rectangle.
                  o  Calculate the boundary rectangle for the band and set
                     the port rectangle to it.
                  o  Set the ClipRgn and the VisRgn to the sizing rectangle.
                  o  Initialize the band by filling it with white space.
                  o  Call DrawPicture to draw the picture into the band's
                     rectangle.
                  o  Do whatever is needed to print the pixel image in the
                     band's rectangle.
                  o  Clear the imaging flag.
                  o  Calculate the next band's position.
                  o  Increment the band's counter and loop back if not done.
            o  If GAP was specified, cause a form feed.
            o  Increment the page count to the next page and loop back
               if not done.
       o  Increment the number of copies counter and loop back if not done.
  o  Free any buffers that you own and close the port.
  o  Dispose of the information dialog that you put up.
  o  Dispose of each picture in the picture list by calling KillPicture.
  o  Dispose of the picture list itself.
  o  Reset the cursor.


PrError                  ($1413)

Description:
    Gets the error code from the last Print Manager call.

Passed:
    None

Returned:
    LastError            WORD        Result code from last Print Manager call

Performs the following:
  o  Gets the driver's internal error value (which was determined by the last
     driver call) and sets the return parameter LastError to it.

Possible Errors:
noError                  $0000
PrAbort                  $0080       Indicates print job was aborted
                         $1301       Indicates missing drivers
                         $1302       Indicates Port Not On
                         $1303       Indicates No Print Record
                         $1306       Indicates PAP Connection Not Made
                         $1307       Indicates Read/Write PAP Error
```

	$1308	Indicates Printer Connection Failed

PrSetError ($1513)

Description:
Sets the error value.

Passed:

ErrorNumber	WORD	Error number to be set

Returned:
None

Performs the following:
- o Sets the driver's internal error value to the value of the passed ErrorNumber parameter.

GetDeviceName ($1713)

Description:
Used as a communications tool between the printer driver and port driver.

Passed:
None

Returned:
None

Performs the following:
- o Calls the port driver routine PrDevPrChanged with the printer name as input. This is necessary for drivers that will work over AppleTalk. The name passed as the parameter to PrDevPrChanged should be what AppleTalk will use in an NBPLookup situation; for AppleTalk, such a name should follow NBP conventions.

PrDriverVer ($2313)

Description:
Returns the version number of the currently installed printer driver.

Passed:

Wordspace	WORD	Space for results

Returned:

versionInfo	WORD	Printer driver's version number

Performs the following:
- o Gets the internal version number of the printer driver and returns it on the stack at versionInfo.

Note: The internal version number is stored
major byte, minor byte (i.e., $0103 represents version 1.3)

PrGetPrinterSpecs ($1813)

```
Description:
    Returns the type of printer and the printer's characteristics.

Passed:
    Wordspace             WORD         Space for results
    Wordspace             WORD         Space for results

Returned:
    PrinterType           WORD         0 = undefined
                                       1 = ImageWriter or ImageWriter II
                                       2 = ImageWriter LQ
                                       3 = LaserWriter family (except IISC)
                                       4 = Epson
                                       $8001 = generic dot matrix printer
                                       $8003 = generic laser printer
    PrCharacteristics     WORD         Bits 15 - 2 = reserved, must be zero
                                       Bits 1-0:     00 = cannot determine
                                                     01 = black and white
                                                          only
                                                     10 = color capable

Performs the following:
  o  Returns characteristics intrinsic for the printer being supported.


PrGetPgOrientation        ($3813)

Description:
    Returns the page orientation from the current print record.

Passed:
    Wordspace             WORD         Space for results

Returned:
    PgOrientation         WORD         Current page orientation:
                                       0 = portrait
                                       1 = landscape

Performs the following:
  o  Returns the page orientation from the current page setup information in the
     print record.
```

Immediate Mode Procedures

To print in the immediate mode, you need to install procedures which will cause printing when you make QuickDraw II calls (as noted in PrOpenPage). This section describes the structure and parameters for these routines.

To install the immediate mode procedures, first create a procedure table for sixteen entries (16*4 bytes) and fill it with the standard procedures by calling SetStdProcs. Once you have the standard procedures, install the addresses of your procedures into it and call SetGrafProcs. Installing your procedure addresses will cause the appropriate QuickDraw II calls to call your procedures, which, in turn, will perform the actual printing.

The routines that need to be written are known as QuickDraw II "bottleneck procedures." Access to the routines are from bank $E0 (accessed by doing a JSL to the appropriate address in bank $E0). When you call any of the bottleneck procedures, the first direct page of QuickDraw II is active and the following direct page locations are valid:

```
PortRef      $24
MaxWidth     $20
MasterSCB    $08
UserId       $0A
```

Two bottleneck procedures, StdText and StdPixels, are of most concern when writing immediate mode procedures. (Refer to Apple IIGS Technical Note #34 for more information on bottleneck procedures.)

The routine StdText (standard text) is the standard text drawing routine. To install this routine into your procedure table (as described above), make it the first entry (offset of $00). Once it's installed, you can access it by doing a long call to absolute address $E01E04. Its direct page parameters are as follows:

```
DrawVerb      $38    Describes the kind of text to draw.  There
                     are three possible values:
                         DrawCharVerb    0
                         DrawTextVerb    1
                         DrawCStrVerb    2
TextPtr       $DA    If the draw verb is DrawTextVerb or
                     DrawCStrVerb, TextPtr points to the text
                     buffer or C string to draw.
TextLength    $D8    If the draw verb is DrawTextVerb,
                     TextLength contains the number of bytes in
                     the text buffer.
CharToDraw    $D6    If the draw verb is DrawCharVerb,
                     CharToDraw contains the character to draw.
```

The routine StdPixels is the standard pixelmap drawing routine. To install this routine into your procedure table (as described above), put it at offset $20. Once it's installed, you can access it by doing a long call to absolute address $E01E24. Its direct page parameters are as follows:

```
SrcLocInfo    $CC    The LocInfo record for the source pixel
                     map
DestLocInfo   $0C    The LocInfo record for the destination
                     pixel map
SrcRect       $DC    The source rectangle for the operation in
                     local coordinates for the source pixel map
                     (as described in the source LocInfo
                     record)
DestRect      $1C    The destination rectangle for the
                     operation in local coordinates for the
                     destination pixel map (as described in the
                     destination  LocInfo record)
XferMode      $E4    The mode to use for data transfer
RgnHandleA    $50    The handle to the first region to which
                     drawing is clipped (usually the ClipRgn
                     from the GrafPort)  A NIL handle is not
                     allowed.  To signify no clipping, pass a
                     handle to the WideOpen region, which is
                     defined as 10 bytes:
```

```
                              Length      $A       (word)
                              -MaxInt    -$3FFF    (word)
                              -MaxInt    -$3FFF    (word)
                              +MaxInt    +$3FFF    (word)
                              +MaxInt    +$3FFF    (word)

    RgnHandleB    $60     The handle to the second region to which
                          drawing is clipped (usually the VisRgn
                          from the GrafPort)  A NIL handle is not
                          allowed.  To signify no clipping, pass a
                          handle to the WideOpen region.
    RgnHandleC    $70     The handle to the second region to which
                          drawing is clipped (usually the mask
                          region from the CopyPixels or the
                          PaintPixels call)  A NIL handle is not
                          allowed.  To signify no clipping, pass a
                          handle to the WideOpen region.
```

Example:

```
;*****************************************************************
;** Example of Immediate Mode Printer Procedures.               **
;*****************************************************************

Immedprocs    Start

SrcRect       equ $DC
SrcLocInfo    equ $CC
DrawVerb      equ $38
TextPtr       equ $da
TextLength    equ $d8
CharToDraw    equ $d6

;-----------------------------------------------------------------
;
; _StdPixels Procedure (Prints Pixelmaps)
;
;-----------------------------------------------------------------
Pixel         Entry

              phb                     ;save data bank reg on stack
              phk                     ;get program bank reg.
              plb                     ;use as data bank reg.

              lda iPrErr              ;get errors
              beq Continue            ;branch if none
              brl ExitPixel           ;branch if errors

Continue      anop
;This gets the source rectangle and stores it at PixelRect
              ldx #6
MoveSrc       lda SrcRect,x
              sta PixelRect,x
              dex
              dex
              bpl MoveSrc
```

```
;This gets the source LocInfo and stores it at PixelLoc
              ldx #16-2
MoveLI        lda SrcLocInfo,x
              sta PixelLoc,x
              dex
              dex
              bpl MoveLI

              pushptr PixelLoc     ;push pointer to LocInfo
              pushptr PixelRect    ;push pointer to rectangle

;+++++++++++++++++++++++
; Insert code here to print a pixelmap
;    INPUT:    PixelLoc     LONG, Pointer to pixel LocInfo
;              PixelRect    LONG, Pointer to pixels BoundsRect
;           SP->
;+++++++++++++++++++++++

Exitpixel     lda #0               ;return with no errors
              clc
              plb                  ;restore data bank
              rtl                  ;returnith long

PixelLoc      ds 16                ;pixel LocInfo
PixelRect     ds 8                 ;pixel rectangle


;-------------------------------------------------------------------
;
; _StdText Procedure (Prints Standard Text)
;
;-------------------------------------------------------------------
StdText       Entry

              phb                  ;save data bank reg on stack
              phk                  ;get program bank reg.
              plb                  ;use as data bank reg.

              pushptr PenPos
              _GetPen              ;current pen pos. -> PenPos

;+++++++++++++++++++++++
; Insert Code Here to move the printers head to the corresponding
; PenPos position (if needed).
;+++++++++++++++++++++++

              pushword #0          ;space for textwidth
                                   ;(for call to _TextWidth)

              lda DrawVerb         ;get DrawVerb
              beq DoCar            ;if DrawVerb=0 then DoCar

              cmp #1
              beq Dotext2          ;if DrawVerb=1 then Dotext2
;
;We get here if it's a "C" string (DrawVerb=2)
;
```

```
DoCstring      anop
               sep #$20
               longa off
;Search down through string looking for terminator to calc. length
               ldy #0
KeepLooking    lda [TextPtr],y
               beq TheEnd
               iny
               bra KeepLooking
TheEnd         rep #$20
               longa on
               lda TextPtr+2
               pha                     ;push the pointer to string
               lda Textptr
               pha
               phy                     ;push the length of sting
               bra Common
;
;We get here if it's just one character (DrawVerb=0)
;
DoCar          anop
               pushword #0
               tdc
               clc
               adc #CharToDraw         ;calculate addr. of char.
               pha                     ;push addr. of character
               pushword #1             ;push length of one char.
               bra Common
;
;We get here if it's a string of text (DrawVerb=1)
;
DoText2        anop
               lda TextPtr+2
               pha                     ;push pointer to the string
               lda Textptr
               pha
               lda TextLength
               pha                     ;push the strings length
Common         lda 5,s                 ;Dup the last 3 words of
               pha                     ;the stack (for _TextWidth)
               lda 5,s
               pha
               lda 5,s
               pha
;++++++++++++++++++++++
; Insert code here to print the text
;
;    INPUT:    TextPointer    LONG, Pointer to text to print
;              TextLength     WORD, No. of bytes to print
;          SP->
;++++++++++++++++++++++
               _TextWidth              ;get the texts width (DH)
               pushword #0             ;set (DV)=0
               _Move                   ;move current pen location

ExitText       lda #0                  ;return with no errors
               clc
               plb                     ;restore data bank
```

```
              rtl                  ;returnith long

PenPos        ds 4                 ;pen position
              end
```

Further Reference

- Apple IIGS Toolbox Reference, Volumes 1 & 2

END OF FILE TN.IIGS.035

##
FILE: TN.IIGS.036
##

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#36: Port Driver Specifications

Revised by: Matt Deatherage & Suki Lee September 1989
Written by: Dan Hitchens May 1988

This Technical Note describes how to write your own drivers for Apple IIGS ports.
Changed since January 1989: Added description of new port driver structure.

Introduction

A port driver handles certain hardware-specific duties for the Print Manager, such as initializing firmware and handling low-level hardware handshaking protocols, if any are implemented. The port driver structure, like the printer driver structure, insulates the Print Manager from low-level details of printers and interface cards (or ports) so that the same calls work across various hardware configurations, provided drivers are installed on the boot disk.

Note that a port driver could also easily be called a card driver; the term port is used because the first ones written were for the internal ports of the Apple IIGS. A port driver could interface any printer (for which there is a printer driver) with any kind of port or peripheral card that can handle it. A familiar example would be a parallel printer interface card--a port driver for a parallel card would enable the Print Manager to print graphics to any parallel printer connected to it (provided, again, there was a printer driver for the particular printer installed).

In general, you need a port driver for each port or interface card through which you intend to print, and a printer driver for each printer to which you intend to print. On System Disk 4.0, Apple provides port driver files for the printer port (PRINTER), the modem port (MODEM), a port connected to the AppleTalk network (APPLETALK), and a parallel printer interface card (PARALLEL.CARD). Apple also provides printer drivers for the ImageWriter and ImageWriter II (IMAGEWRITER), the ImageWriter LQ (IMAGEWRITER.LQ), the LaserWriter family.(LASERWRITER), and an Epson (EPSON). With this configuration, you can print to any of the printer types above through any of the ports, cards, or over AppleTalk. Other printer drivers and port drivers would extend the user's selection of available configurations.

What's in a Port Driver

File Structure

Users can install new port drivers into the system by copying a port driver file into a subdirectory called DRIVERS within the SYSTEM subdirectory or by running the Installer if the driver is supplied with a script to install it. The port driver file must be of type $BB. There are two kinds of port drivers: local drivers, intended to drive a printer connected locally, and network drivers, which handle printers connected over an AppleTalk network. Local drivers have an auxiliary type of $0002, and AppleTalk drivers (there should be only one) have an auxiliary type of $0003.

Port Driver Calls

A port driver must support the following calls:

```
    PrDevPrChanged          $1913
    PrDevStartup            $1A13
    PrDevShutDown           $1B13
    PrDevOpen               $1C13
    PrDevRead               $1D13
    PrDevWrite              $1E13
    PrDevClose              $1F13
    PrDevStatus             $2013
    PrDevAsyncRead          $2113    (alias PrDevInitBack)
    PrDevWriteBackground    $2213    (alias PrDevFillBack)
    PrPortVer               $2413
    PrDevIsItSafe           $3013
```

Note that a network port driver has much more work to do than a regular (local) port or card driver. A local driver only has to worry about one printer, whereas a network port driver may find that there is not even a printer available on a running network. The information on network drivers is provided mostly for informational purposes; you should never find it necessary to write your own AppleTalk port driver.

Entering and Exiting a Port Driver

Entering and exiting is the same as described for the printer driver calls in Apple IIGS Technical Note #35, Printer Driver Specifications. The new driver structure described there applies as well. As of this writing, there are no optional calls a port driver may support. The documented list must be supported in its entirety.

PrDevPrChanged $1913

Description:
The Print Manager makes this call every time the user accepts this port driver in the Choose Printer dialog.

Input: LONG printer name pointer

Direct Connect:
- o Makes sure that this port has been set up correctly in the Control Panel (parity, baud rate, etc.), and puts up an alert for the user if it has not been. Remember that if you change settings, even at the user's request, you should change the Battery RAM parameters as well, so the setting changes will be reflected when the user enters the Control Panel.

Network:
- o Copies the printer name to local storage for use in the NBPLookup function of the AppleTalk PAPopen and PAPstatus calls, usually by placing it in the AppleTalk parameter block. This function is similar to that performed by PrStartUp, except that PrDevPrChanged is called whenever the printer is changed by the user with the Choose Printer dialog.

PrDevStartUp $1A13

Description:
This call is not required to do anything. However, if your driver needs to initialize itself by allocating memory or other setup tasks, this is the place to do it. Network drivers should copy the printer name to a local storage area for later use.

Input:	LONG	printer name pointer
	LONG	zone name pointer

Direct Connect:
- o Required to do nothing. This is a good place to do your own set-up tasks, if you have any.

Network:
- o Copies the printer name and the zone name to local storage for use in the NBPLookup function of the AppleTalk PAPopen and PAPstatus calls, usually by placing it in the AppleTalk parameter block.

PrDevShutDown $1B13

Description:
This call, like PrDevStartUp, is not required to do anything. However, if your driver performs other tasks when it starts, from the normal (allocating memory) to the obscure (installing heartbeat tasks), it should undo them here. If you allocate anything when you start, you should deallocate it when you shutdown. Note that this call may be made without a balancing PrDevStartUp, so be prepared for this instance. For example, do not try to blindly deallocate a handle that your PrDevStartUp routine allocates and stores in local storage; if you have not called PrDevStartUp, there is no telling what will be in your local storage area.

Input: none

PrDevOpen $1C13

Description:
This call basically prepares the firmware for printing. It must initialize the firmware for both input and output. Input is required so the connected printer may be polled for its status.

A network driver has considerably more work to do, including the possibility of asynchronous communications. Details are provided below.

Input:	LONG	completion routine pointer
	LONG	reserved long

Direct Connect:
- o Initializes the firmware for input and output, preparing for reading from or writing to the printer.
- o If the completion pointer is NIL, then RTL. If it is not NIL, then perform a JSL to the completion routine.

Network:
- o Initializes the End-Of-Write parameter in the AppleTalk PAPWrite parameter block to zero. Never call AppleTalk INIT to initialize the firmware.
- o If the completion pointer is NIL, then prepares for synchronous communications. If it is not NIL, prepares for asynchronous printing.
- o Calls AppleTalk PAPopen to make connection, returning an error if one is returned to you.
- o Stores the AppleTalk Session number in the PAPRead, PAPWrite and PAPClose parameter blocks.
- o Executes an RTL if there is no completion routine (pointer is NIL), otherwise perform a JSL to the completion routine.

PrDevRead $1D13

Description:
This call reads input from the printer.

Input:	WORD	space for result
	LONG	buffer pointer
	WORD	number of bytes to transfer
Output:	WORD	number of bytes transferred

Direct Connect:
- o Reads a specified number of bytes from the printer into the buffer.

Network:
- o Calls AppleTalk PAPRead to read synchronously. Since there is no completion pointer, reading from a network device must always be done synchronously. To read asynchronously, use PrDevAsyncRead.

PrDevWrite $1E13

Description:
Writes the data in the buffer to the printer and calls the completion routine.

Input:	LONG	write completion pointer
	LONG	buffer pointer
	WORD	buffer length

Direct Connect:
- o Writes the contents of the buffer to the printer.
- o If the completion pointer is NIL, then RTL. If it is not, then perform a JSL to the completion routine.

Network:
- o If the completion pointer is NIL, then writing will occur synchronously. Otherwise, writing will occur asynchronously.
- o Calls AppleTalk PAPWrite to transfer the contents of the buffer.

- o If the completion pointer is NIL, then RTL to the caller. Otherwise, perform a JSL to the completion routine first, with the error code in the accumulator.

PrDevClose $1F13

Description:
This call is not required to do anything. However, if you allocate any system resources with PrDevOpen, you should deallocate them at this time. As with start and shutdown, note that PrDevClose could be called without a balancing PrDevOpen (the reverse is not true), and you must be prepared for this if you try to deallocate resources which were never allocated.

Input: none

Direct Connect:
- o No required function.

Network:
- o Sets End-Of-Write parameter in AppleTalk PAPWrite parameter block to one.
- o Calls PAPWrite with no data.
- o Calls PAPClose.

PrDevStatus $2013

Description:
This call performs differently for direct connect and network drivers. For direct connect drivers, it currently has no required function, although it may return the status of the port in the future. For network drivers, it calls an AppleTalk status routine, which returns a status string in the buffer (normally a string like "Status: The print server is spooling your document").

Input:	LONG	status buffer pointer

Direct Connect:
- o Does nothing.

Network:
- o Calls AppleTalk PAPStatus.

PrDevAsyncRead $2113

Description:
Since PrDevRead cannot read asynchronously, this call is provided for that task. Note that this does nothing for direct connect drivers, and if the completion pointer is NIL, it behaves for network drivers exactly as PrDevRead does.

Input:	WORD	space for result
	LONG	completion pointer
	WORD	buffer length
	LONG	buffer pointer
Output:	WORD	number of bytes transferred

Direct Connect:
- o Does nothing.

Network:
- o If the completion pointer is NIL, then performs exactly as PrDevRead.
- o Calls AppleTalk PAPRead; the actual length read is passed back in the PAPRead parameter block.
- o Perform a JSL to the completion routine, which returns the length read in the X register and an EOF flag in the Y register. As usual, the accumulator contains the error code and the carry is set if an error occurs.
- o In the case of a synchronous call, it performs a JSL to the completion routine, which pushes the length read onto the stack.

PrDevWriteBackground $2213

Description:
This routine is not implemented at this time.

Input:	LONG	completion procedure pointer
	WORD	buffer length
	LONG	buffer pointer

PrPortVer $2413

Description:
Returns the version number of the currently installed port driver.

Input:	WORD	space for result
Output:	WORD	Port driver's version number

Direct Connect and Network:
- o Gets the internal version number of the port driver and returns it on the stack.

Note: The internal version number is stored as a major byte and a minor byte (i.e., $0103 represents version 1.3)

PrDevIsItSafe $3013

Description:
This call checks to see if the port or card which your driver controls is enabled. It should check at least the corresponding bit of $E0C02D, and checking the Battery RAM settings wouldn't hurt any either.

Input:	WORD	space for result
Output:	WORD	Boolean indicating if port is enabled

Direct Connect and Network:
- o Checks the system to see if the hardware and/or firmware for the card or port this driver controls is enabled, and returns TRUE if it is safe to

proceed and FALSE if not. Note that for a port driver that controls an interface card, this call should return FALSE if the card is disabled and the port is enabled, while for a port driver which controls an Apple IIGS internal port, the returned value should be TRUE if the port is enabled and FALSE if not.

Further Reference

- Apple IIGS Toolbox Reference, Volumes 1 & 2
- Apple IIGS Technical Note #35, Printer Driver Specifications

END OF FILE TN.IIGS.036

```
#####################################################################
### FILE: TN.IIGS.037
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#37: Free-Form Synthesizer Tips

Revised by: Jim Mensch November 1988
Written by: Jim Mensch May 1988

This Technical Note is intended to help a person who is unfamiliar with the Apple IIGS Sound Tool Set use the Free-Form Synthesizer effectively.

The primary function of the Free-Form Synthesizer is to allow an application program to start one or more complex digitized or computed waveforms playing on the Apple IIGS without further intervention from the application. The waveform is a series of bytes, each representing the amplitude of your outgoing sound at a particular moment in time (defined by the sampling frequency you set). After a call to FFStartSound, the Sound Tool Set takes care of all chores involved in loading the DOC RAM, setting up registers, and actually playing your sound. Once playing, your sound will continue until either the Sound Tool Set encounters a NIL pointer in the waveform list, or until you call FFStopSound.

FFStartSound Parameters

FFStartSound has only two parameters: the first a Word containing channel, generator, and mode information, and the second a Pointer to a parameter block.

```
      |15 |14 |13 |12 |11 |10| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
        |___________|   |__________|   |___________|   |___________|
              |              |               |               |
              |              |               |               |
DOC channel number ($0-$1)   |   Reserved must be set to 0   |
top 3 bits should be set to 0|                               |
                             |                  Free-Form Synthesizer = $01
                 Generator number ($0-$E)           Note Synthesizer = $02
                                                        Reserved = $03-$07
                                              Application defined = $08-$0F
```

Figure 1 - Channel-Generator-Mode Word

The Channel-Generator-Mode Word is broken down into 4 nibbles. The low-order nibble specifies the particular synthesizer you are using. (Because this Note is only about the Free-Form Synthesizer, we will be using only a 1 in this nibble.) The adjacent nibble must be set to 0 for now. The next nibble specifies which generator to use. The IIGS has 15 generators from which to choose, and as the application designer, it is up to you to decide which one to use. It might be appropriate, however, to call FFGeneratorStatus first to ensure that the generator currently is available. (It could be in use already by a desk accessory or previously started sound.) The high-order nibble specifies which channel to use. The IIGS supports two separate sound channels for output. If you are using a stereo adapter, you could start up many sounds and route them to either channel 0 or channel 1 to get a full stereo effect. (The channel is ignored if you are not using a special piece of multi-channel hardware.)

The parameter block contains parameters describing the sound and how it should be played. Here is a sample Pascal definition of that parameter block:

```
    FFParmBlock = record
                   waveStart:Ptr;
                   waveSize:Integer;
                   freqOffset:Integer;
                   DOCBuffer:Integer;     { High order byte significant }
                   bufferSize:Integer;    { Low order byte significant }
                   nextWave:^FFParmBlock;
                   volSetting:Integer;
                  end;
```

The first parameter is a 4-byte address telling the Free-Form Synthesizer where in memory it can locate your sample data. The next parameter is a word specifying the number of 256-byte pages of sound you wish to play. The waveform data should be a series of bytes, each representing one sample. Wave tables must be exact multiples of 256 bytes.

Note: A zero value in the waveform can cause a sound to stop, so be sure to check your data to ensure that this does not happen.

The frequency offset parameter specifies the sampling frequency that the Free-Form Synthesizer should use during playback. This number can be computed by the following formula:

```
freqOffset = ((32*Sample rate in Hertz)/1645)
```

The frequency offset parameter is the most often misunderstood parameter, so I will explain a little about sampling rates. The sampling rate is how many samples (bytes) per second to play. If you have a digitized wave that represents 2 seconds of sound, and it takes up 44K of memory, then it was sampled at 22 kHz (which, by the way, is good for full sound reproduction). The sampling rate must be at least twice that of the maximum fundamental frequency you want to sample. However, for good sound reproduction, you may want to sample at least eight times the fundamental frequency in order to capture the higher harmonics of musical instruments and the human voice.

The DOC starting address and buffer size tell the Free-Form Synthesizer which portion of the 64K sound RAM to use as a buffer during playback. The wave is taken from your waveform in chunks and placed in sound RAM for playback. Each time the buffer nears empty, it will need to be reloaded with more sound. The size of the buffer specified determines how often the Free-Form Synthesizer must interrupt the 65816 to reload the buffer. The buffer size must be a power of two because of the way the sound General Logic Unit (GLU) specifies addresses. (The value for this parameter must also be a power of two.) A good length to use would be at least 1/10 second of sound. For example, if you were using a sampling rate of 16 kHz (16,000 samples per second), you would want a buffer at least 2,048 bytes long, or about 8 pages. It does not hurt to round this number up. You manage the DOC RAM, so you should decide what memory to use. It is usually a good idea to have multiple buffers if you have a chain of waves. (I like leaving page zero free, as the Note Synthesizer uses the data in the first 256 bytes, and accidentally placing a zero in that page could cause it to fail.)

The next wave pointer is a 4-byte pointer to the next parameter block. With this parameter you can string together many waveforms for more continuous sound, or you can make your sounds infinitely recursive by pointing back to the original wave form.

The volume setting is a word which represents the relative playback volume. It can range from 0 to 255.

Other Tips

When you shut down the Sound Tool Set, it will stop all pending sounds, so be sure to leave ample time between starting and ending a sound. If you have a series of wave forms strung together, you can change their parameters on the fly. Changes take effect as soon as the waveform is started. (You could use this to find the correct sampling frequency of a wave, by having the next wave pointer point back to the start of your parameter block. This would cause the sound to play indefinitely. You then could change the freqOffset value, and the sound would change each time it is restarted.)

Here is a sample code segment (in APW Assembler format) that creates a 1-kHz wave in memory sampled at 16 kHz and plays it:

```
FFSound       DATA

theSound      ds    $2000           ; FFSound wave...
MyFFRecord    dc    A4'theSound'    ; address of wave
              dc    i'$20'          ; size of wave in pages..
Rate          dc    i'311'          ; 16-kHz sample rate
              dc    i'1'            ; DOC starting address
              dc    i'$0800'        ; DOC buffer size
              dc    a4'0'           ; no next wave
Vol1          dc    i'$007F'        ; kinda medium..

; 1-kHz triangle wave sampled at 16 kHz one full segment
oneAngle      dc    i1'$40,$50,$60,$70,$80,$90,$A0,$B0'
              dc    i1'$C0,$B0,$A0,$90,$80,$70,$60,$50'
              End

TestFF        Start
              Using    FFSound
MakeWave      ANop
              ldx    #$0000
MW0010        txa                   ; get index
              and    #$000F         ; use just low nibble as index
              tay                   ; into triangle wave table
              lda    oneAngle,y     ;
              sta    theSound,X     ; and store it into sound buf
              inx
              inx
              cpx    #$2000         ; we Done?
              blt    MW0010         ; nope better finish
```

```
              PushWord     #$0001
              PushLong     #MyFFRecord
              _FFStartSound
              rts
              end
```

Further Reference

- Apple IIGS Toolbox Reference, Volume 2

END OF FILE TN.IIGS.037

##
FILE: TN.IIGS.038
##

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#38: List Controls in Dialog Boxes

Revised by: Eric Soldan July 1989
Written by: Keith Rollin May 1988

This Technical Note describes how to include a list control into a dialog box. Sample APW C source code is included.
Changes since November 1988: Added discussion of System Software 5.0 considerations.

The need to put a list control into a dialog box is obvious. The Print Manager does it. The Font Manager does it. You may want to use one in your own application to manage a list of data base fields or spreadsheet functions. However, performing the task is not as obvious as the need.

Given the new features of TaskMaster in System Software 5.0, it is now much easier to emulate a modal dialog in a normal window. If you need to add a list control to a modal dialog, you should seriously consider emulating a modal dialog with a normal window instead of using the Dialog Manager. If you use the Dialog Manager, the following procedure and sample C fragment illustrate the technique necessary for adding a list control.

Individual Steps

Basically, there are three check-off items for putting a list control into a dialog box:

1. You must install the list explicitly into the dialog box yourself. This should be done after you have created the dialog box with a call to NewModalDialog or GetNewModalDialog. Do not install it as a UserItem or UserCtlItem. Installing it as a UserItem would cause the Dialog Manager to place an invisible custom control over the list, preventing later use of FindControl to manage it. Installing the list as a UserCtlItem does not allow the list control to be properly initialized.

```
InitValues()
{
    /* Get a Full Screen, invisible dialog window with only
       a Quit button in it*/
    myDialog = GetNewModalDialog(&PrintDialog);

    /* Add this List Control ourselves */
    myListHndl = CreateList(myDialog,&myList);
```

```
        /* Get the handle for the Scrollbar Control */
        listScrollHandle = (**myListHndl).ctlListBar;

        /* Save and Zero out the RefCons */
        listRefCons = GetCtlRefCon(myListHndl);
        scrollRefCons = GetCtlRefCon(listScrollHandle);
        ZeroRefCons(); /* This is explained below in item #3 */

        /* Now show the dialog box */
        ShowWindow(myDialog);
    }
```

2. Because the list control is not a dialog item, a custom FilterProc must be installed for ModalDialog to test for mouse-down events. Pass the address of this routine (with the high bit set so that default handling of items will be in effect) when you call ModalDialog.

```
    pascal Word myFilterProc(theDialog, theEvent, theItem)
        GrafPortPtr      theDialog;
        EventRecord      *theEvent;
        long             *theItem;

    {
        CtlRecHndl  tHandle;

        if ((*theEvent).what == mouseDownEvt) {
           FindControl(&tHandle,(*theEvent).where,theDialog);
           if ((tHandle == myListHndl) || (tHandle == listScrollHandle)) {

              /* Set the RefCons back to the way the list manager
                 likes them */
              RestoreRefCons();
              TrackControl((*theEvent).where,(LongProcPtr) -1, tHandle);
              ZeroRefCons();

              /* Tell the Dialog Manager that we handled this event */
                 return(true);
           }
        }
        /* We didn't do anything, so return false to get Dialog Manager
           to handle this event */
      return(false);
    }
```

3. The Dialog Manager uses the RefCon field of its items (all of which are installed as controls). Unfortunately, the List Manager also uses the RefCon field for its own purposes. This shared use means that a judicious juggling of those values is required. This juggling is the reason for the two routines RestoreRefCons and ZeroRefCons used above.

```
    /* Zero out the RefCons for the Dialog Manager */
    ZeroRefCons()
    {
        SetCtlRefCon(0,myListHndl);
        SetCtlRefCon(0,listScrollHandle);
```

```
      }

      /* Restore the RefCons for the List Manager */
      RestoreRefCons()
      {
          SetCtlRefCon(listRefCons,myListHndl);
          SetCtlRefCon(scrollRefCons,listScrollHandle);
      }
```

Note: Because the Dialog Manager currently uses the RefCon to keep track of which dialog item is identified with which particular control, zeroing the RefCon fields can cause a little confusion. Specifically, those who would like to do GetFirstDItem from within a Standard File call may get a zeroed RefCon as a result. This is true for Standard File 3.0 and later (System Software 5.0), as this is the first implementation of Standard File to use the List Manager.

Putting It All Together

Here are most of the pieces put together. InitTools and ShutDownStuff routines have been omitted, but they are straightforward.

```
char                  **y,*z;
GrafPortPtr           myDialog;
ListCtlRecHndl        myListHndl;
CtlRecHndl            listScrollHandle;
long                  listRefCons, scrollRefCons;


#define Quit          ok

char    quitStr[]     = "\pQuit";

ItemTemplate quitButton =  {
               Quit,
               140,450,154,590,
               buttonItem,
               quitStr,
               0,
               0,
               NULL};


DialogTemplate PrintDialog = {
               30,20,190,620,
               false,
               0,
               &quitButton,
               NULL};

char string1[] = "String1";
char string2[] = "String2";
char string3[] = "String3";
char string4[] = "String4";
char string5[] = "String5";
char string6[] = "String6";
```

```
char string7[] = "String7";
char string8[] = "String8";

MemRec  myMembers[8] = {
               string1, 00,
               string2, 00,
               string3, 00,
               string4, 00,
               string5, 00,
               string6, 00,
               string7, 00,
               string8, 00};

ListRec  myList = {
               40,175,102,400, /* Enclosing Rectangle */
               8,              /* Number of List Members */
               6,              /* Max Viewable members */
               3,              /* Bit Flag */
               1,              /* First member in view */
               NULL,           /* List control's handle */
               NULL,           /* Address of Custom drawing routine */
               10,             /* Height of list members */
               5,              /* Size of Member Records */
               (MemRecPtr)myMembers,/* Pointer to first element in MemRec[] */
               NULL,           /* Becomes Control's refCon */
               NULL            /* Color table for list's scroll bar */
               };

/* ************************** */

main()
{
    word what;

    InitTools();               /* initialize tools */
    InitValues();              /* Get dialog box. Install List control */
    do {
        what = ModalDialog((WordProcPtr)((long)myFilterProc | 0x80000000));
    } while (what != Quit);
    ShutDownStuff();
}

pascal Word myFilterProc(theDialog, theEvent, theItem)
       GrafPortPtr    theDialog;
       EventRecord    *theEvent;
       long           *theItem;

{
       CtlRecHndl     tHandle;

       if ((*theEvent).what == mouseDownEvt) {
           FindControl(&tHandle,(*theEvent).where,theDialog);
           if ((tHandle == myListHndl) || (tHandle == listScrollHandle)) {

               /* Set the RefCons back to the way the list manager
                  likes them */
               RestoreRefCons();
               TrackControl((*theEvent).where,(LongProcPtr) -1, tHandle);
```

```
                ZeroRefCons();

                /* Tell the Dialog Manager that we handled this event */
                return(true);
            }
        }
        /* We didn't do anything, so return false to get Dialog Manager
           to handle this event */
  return(false);
}

/* Zero out the Refcons for the Dialog Manager */
ZeroRefCons()
{
    SetCtlRefCon(0,myListHndl);
    SetCtlRefCon(0,listScrollHandle);
}

/* Restore the Refcons for the List Manager */
RestoreRefCons()
{
    SetCtlRefCon(listRefCons,myListHndl);
    SetCtlRefCon(scrollRefCons,listScrollHandle);
}


InitValues()
{
    /* Get a Full Screen, invisible dialog window with
       only a Quit button in it */
    myDialog = GetNewModalDialog(&PrintDialog);

    /* Add this List Control ourselves */
    myListHndl = CreateList(myDialog,&myList);

    /* Get the handle for the Scrollbar Control */
    listScrollHandle = (**myListHndl).ctlListBar;

    /* Save and Zero out the RefCons */
    listRefCons = GetCtlRefCon(myListHndl);
    scrollRefCons = GetCtlRefCon(listScrollHandle);
    ZeroRefCons();

    /* Now show the dialog box */
    ShowWindow(myDialog);
}
```

END OF FILE TN.IIGS.038

###
FILE: TN.IIGS.039
###

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#39: Mega II Video Counters

Revised by: Dave Lyons July 1989
Written by: J. Rickard May 1988

This Technical Note describes the Mega II video output registers, which your applications can use to get information about where the beam is located on the Apple IIGS display.
Changes since November 1988: Corrected description of when VBL begins and simplified example code to read the scan line number.

The Mega II controls video timing for the Apple IIGS with a 16-bit counter split into a 7-bit horizontal and a 9-bit vertical part (Figure 1). The counter outputs are made available to programs running on the machine through two addresses in the I/O space, $C02E for the vertical count and $C02F for the horizontal count. These outputs can be used by a program for finer control over display update timing.

```
 _______________________________________________________________
|           Vertical Counter           |     Horizontal Counter  |
|______________________________________|_________________________|
| V5| V4| V3| V2| V1| V0| VC| VB| VA|HPE| H5| H4| H3| H2| H1| H0|
|__________________________________|___|_________________________|
|             $E0C02E              |           $E0C02F           |
|________________________________________________________________|
```

Figure 1 - Mega II Video Counter

You can see that one bit of the nine-bit vertical counter is in location $E0C02F with the seven bits of the horizontal counter. Keep this location in mind when reading the counters.

The seven-bit horizontal counter starts at $00 and counts from $40 to $7F (the sequence is $00, $40, $41,...,$7E, $7F, $00, $40,...). The active video time consists of 40 one microsecond clock cycles starting with $58 and ending with $7F. Since this count changes at 980 nanosecond intervals, it will probably be of little use to most programs.

The nine-bit vertical counter ranges from $FA through $1FF (250 through 511) in NTSC mode (vertical line count of 262) and from $C8 through $1FF (200 through 511) in PAL video timing mode (vertical line count of 312). Vertical counter value $100 corresponds to scan line zero in NTSC mode. The vertical count changes at 63.7 microsecond intervals, giving a program time to respond to a specific count before it changes. The vertical counter byte, at $E0C02E, only changes half as often (at 127 microsecond intervals) since the lowest bit of the nine-bit counter is actually stored in the next byte (at $E0C02F).

The nine-bit counter consists of bits VA, VB, VC, V0, V1, V2, V3, V4 and V5. Bits V0 through V5 can be read as a six-bit value. If this value is between 0 and 23, it is the line on the text screen currently being updated. Other values indicate the vertical blanking cycle is occurring. Bits VA through VC can be read as a three-bit value (0-7) indicating which scan line of a text character (characters are composed of eight lines) is currently being drawn.

The vertical counter can also be used to determine which scan line (0-191 for most video modes, including high-resolution and double high-resolution, and 0-199 for super high-resolution) is being updated at any given moment.

Example

Suppose you want to repaint a portion of the super high-resolution screen that will require more time than the vertical blanking period allows. You will have a tear in your animation when the screen's refresh cycle catches up with your drawing.

One solution to this problem would be locating the approximate place the tear occurs and starting your drawing when the system is scanning that line of graphics. Let's say you are painting an area that is about (for example) 100 pixels wide and 200 pixels tall in 320 mode, and that the tear will occur somewhere around scan line 80. To avoid the tear, you would wait until the system is scanning line 80, then you would start redrawing at the top of the screen. This way, you should be finished drawing when the system is back to scanning line 80 again and you will have flicker-free screen updating.

The tricky part is trying to determine just when the system is scanning any given scan line. One way to determine this is to examine the Mega II video counter registers at $E0C02E (vertical) and $E0C02F (horizontal), described above. By using some simple arithmetic you can come up with the exact scan line being updated. The following piece of code computes the current scan line number (assuming eight-bit native mode):

```
    lda    >$E0C02F
    asl    A                    ;VA is now in the Carry flag
    lda    >$E0C02E
    rol    A                    ;roll Carry into bit 0
```

The result (in A) is the low byte of the vertical counter. This value is 0 for the first scan line, 1 for the second scan line, etc. Values $FA to $FF are used twice, since you ignore the high byte of the vertical counter. (The six scan lines immediately above scan line 0 are numbered $0FA to $0FF, and the six above those are $1FA to $1FF.) The example code leaves the highest bit of the vertical counter in the Carry flag, if you really want it.

Note that the VBL interrupts always trigger at scan line 192, even in Super Hi-Res display mode, and that the $C019 soft switch indicates vertical blanking is in effect starting at scan line 192. Be careful polling for a specific scan line number--if interrupts are enabled, it is conceivable that the system will be busy processing an interrupt every time that scan line is being scanned, so your program will hang forever waiting for it.

Setting a scan line interrupt is another way to determine when a particular super high-resolution scan line is being drawn. However, you must be careful in turning scan line interrupts on and off so that you do not interfere with the cursor in QuickDraw II (which uses scan line interrupts).

Further Reference

- Apple IIGS Toolbox Reference, Volume 2
- Apple IIGS Technical Note #40, VBL Signal

END OF FILE TN.IIGS.039

```
#####################################################################
### FILE: TN.IIGS.040
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#40: VBL Signal

Revised by: Dave Lyons July 1989
Written by: Rob Moore & Rilla Reynolds May 1988

This Technical Note discusses reading the VBL signal to accomplish smooth animation.
Changes since November 1988: Noted that vertical blanking does not begin when you might expect on the Apple IIGS and removed references to the Apple IIc.

Applications can accomplish smooth animation on the Apple IIGS and Apple IIe by changing the data on the screen during the time the system is tracing the unusable area of the display. This time is called "vertical blanking" or "VBL" in this Note. You can determine the state of the VBL signal by reading location $C019.

On the Apple IIGS, the $C019 sense of the VBL signal differs from the IIe. On the IIGS, the screen is blanked when the most significant bit of $C019 is high (greater than 127 or $7F), while on the IIe, the screen is blanked when the bit is low (less than 128 or $80).

A VBL interrupt also is available on Apple II systems via the Apple IIGS Miscellaneous Tool Set or mouse firmware, the Apple IIe mouse card, and the Apple IIc mouse firmware.

On the Apple IIGS, vertical blanking begins at scan line 192 regardless of the display mode. When the Super Hi-Res display is visible, vertical blanking begins eight scan lines before the bottom of the display area. If the VBL interrupt is enabled, it triggers at scan line 192.

Further Reference

- o Apple IIGS Technical Note #39, Mega II Video Counters

END OF FILE TN.IIGS.040

```
#######################################################################
### FILE: TN.IIGS.041
#######################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#41: Font Family Numbers

Revised by: Keith Rollin & Matt Deatherage — November 1988
Written by: Rilla Reynolds & Jeff Erickson — May 1988

This Technical Note lists fonts and font family numbers as well as considerations when printing to a LaserWriter printer and a word of caution about using font family numbers.

The following table lists fonts and their corresponding font family numbers. All family numbers are listed in decimal format except the first three.

ID	Family Name	ID	Family Name
$FFFD	Chicago	12	Los Angeles
$FFFE	Shaston	13	Zapf Dingbats*
$FFFF	(no font)	14	Bookman*
0	System Font	15	Helvetica Narrow*
1	System Font	16	Palatino*
2	New York	18	Zapf Chancery*
3	Geneva	20	Times*
4	Monaco	21	Helvetica*
5	Venice	22	Courier*
6	London	23	Symbol*
7	Athens	24	Taliesin
8	San Francisco	33	Avant Garde*
9	Toronto	34	New Century Schoolbook*
11	Cairo		

Fonts denoted with an asterisk (*) are resident in the ROM on the LaserWriter Plus, IINT and IINTX printers. The name of Times on these printers is actually Times-Roman. The decimal font family ID for Shaston (a modified Helvetica) is 65534 (-2), not 65524 as documented in the Font Manager chapter of the Apple IIGS Toolbox Reference.

When printing to a LaserWriter printer with the font substitution option turned on, the system substitutes Times, Helvetica, and Courier for the screen fonts New York, Geneva, and Monaco respectively.

Prior to System Disk 3.2, all non-LaserWriter fonts (except New York, Geneva, and Shaston) were converted to Courier when printing. With System Disk 3.2 and later, the LaserWriter driver will print bitmap versions of the screen fonts if they are non-LaserWriter fonts unless it is driving an original LaserWriter printer. In this case, fonts which are in ROM on later LaserWriter printers are converted to Courier unless you download a PostScript version of the font prior to printing. This difference is a limitation of the current LaserWriter driver and it occurs even if the font substitution option is turned off.

Caution

Font family numbers can be arbitrary numbers which the system assigns to fonts. We recommend that you always ask for a font by name (with the Font Manager call GetFamNum), then use the returned family number as input to those calls which require it. (On the Macintosh, the Font/DA Mover checks to see if a font family number is already in use by the system when it installs fonts. If it finds that a number is already in use, it changes the current font number to an unused number. If you move a font from the Macintosh to the IIGS, the font family number is likely to be arbitrary, as is the font family number of any user-created fonts.

Further Reference
o   Apple IIGS Toolbox Reference, Volumes 1 & 2

END OF FILE TN.IIGS.041

###
FILE: TN.IIGS.042
###

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#42: Custom Windows

Written by: Dan Oliver & Keith Rollin November 1988

This Technical Note describes custom windows which are now supported with Window Manager version 2.2. This Note supersedes all prior documentation on custom windows.

With Window Manager version 2.2 or later, which is available on Apple IIGS System Disk 3.2 and later, you may now define your own type of window or window shape, such as a round or hexagonal window. You also may define a window which performs tasks that would normally be handled by an application.

To define your own type of window, a custom window, you must write a routine that performs some window functions. This routine is a window definition procedure (defProc), and in this case it is a custom window defProc. When the Window Manager needs to do something window specific, it calls your defProc.

The window defProc is a good part of the Window Manager, and writing one is not an easy task. A window defProc must perform complicated tasks that are very dependent on the state of the machine, and it must be very careful not to disturb the state of the machine. One of the problems in writing a defProc is knowing when it can do something and when it cannot. It is almost impossible to document all of the combinations of calls that you can or cannot make from one part or another of the defProc, and even if all cases were found, the resulting document would read like something from an obscure government bureau and probably be even harder to understand.

Now that you know writing a defProc is tough, here's how to make things as easy as possible. Try to understand how the system interacts with the defProc and work with the system. For example, a defProc is called to hit test window parts when the user presses the mouse button. The Window Manager will pass that part back to the defProc to perform drawing while the Window Manager is tracking the pressed button. The defProc could keep control when asked to hit test and perform the tracking itself, but since this is not how the system is designed to work, your defProc will be hard to write, may not ever work correctly, and may break in future versions of the Window Manager. Try to stay on the path outlined in this Technical Note. Also understand that the interface to definition procedures is as general as possible to allow them to perform tasks which are as yet unknown. To allow for this future growth, the outlined path is not always a clear path.

Another way to make things easier is to write conservative code. Do not assume things like the data bank being set to something nice when the defProc is called or the caller restoring the direct page pointer upon return if you have changed it. Use caution. A defProc can be very difficult to debug because it is not very linear and can be called when you least expect.

Interaction Between the Window Manager and TaskMaster

The Window Manager and TaskMaster actually do much less than many people think since window definition procedures perform most of the tasks. The definition procedures handle such things as title bars, information bars, and scroll bars, while the Window Manager and TaskMaster support these things by passing requests to the defProc in standard ways. The Window Manager knows that windows have some shape, overlap, may contain parts, may be invisible, and are created and deleted, but it does not know much else. TaskMaster knows to call GetNextEvent and performs some tasks, but much of what many people consider TaskMaster is contained in the standard document window defProc. In addition to the list mentioned above, the defProc handles calling TrackGoAway and scrolling the content. The remainder of this Note describes what is expected of a defProc and when.

Telling the Window Manager About Your Window

You tell the Window Manager about your custom window when NewWindow creates it. Instead of passing the parameter list defined in NewWindow, you pass a pointer to a custom window parameter list. A custom window parameter list is defined as follows:

```
    paramID       WORD       ID of parameter list, zero for custom.
    newDefProc    LONG       Address of your custom defProc.
    newData       BYTE[n]    Additional data defined by your defProc.
```

NewWindow checks the paramID field and calls your defProc with the pointer to the parameter list. See the wNew operation under Calling the Custom DefProc for more information.

Once NewWindow creates the window, the Window Manager will always know that it is defined by your defProc.

Calling the Custom defProc

A window defProc is called with the following items on the stack:

```
    16 |result                        | LONG - result returned to Window Manager,
       |______________________________|        defined by each operation
    14 |windGlobals                   | LONG - pointer to Window Globals (defined
below)
       |______________________________|
    12 |OperationCode|                  WORD - operation number to be performed
       |_____________|_______________
     8 |theWindow                     | LONG - pointer to window's record
       |______________________________|
     4 |param                         | LONG - pointer to additional parameter
       |______________________________|        defined by each operation
     1 | RTL address         |          BYTE[3] - long return address
       |_____________________|________
       |                              | <-- Stack Pointer
       |                              |
```

Figure 1 - Stack Prior to Calling a Window defProc

The defProc must return with the carry flag clear if there was no error or with the carry flag set and the y register set with an error code if there was an error.

Window globals (windGlobals) is a pointer to a table of variables which the Window Manager maintains for use by the defProc. The table is defined as follows:

lineW	WORD	Width of vertical lines (size depends on video mode).
titleHeight	WORD	Height of a standard title bar.
titleYPos	WORD	Y offset for the title (in system font) to center in a standard title bar.
closeHeight	WORD	Height of the close box icon.
closeWidth	WORD	Width of the close box icon.
defWindClr	LONG	Pointer to the default window color table.
windIconFont	LONG	Handle of the current window icon font.
screenMode	WORD	TRUE if 640 mode, FALSE if 320 mode.
pattern	BYTE[32]	Temporary pattern buffer.
callerDpage	WORD	Direct page pointer of the last caller to TaskMaster.
callerDataB	WORD	Data bank of the last caller to TaskMaster (bank in both bytes).

Operation numbers are as follows (each operation is described later in its own section):

wDraw	0	Draw the window's frame.
wHit	1	Tell in what region the mouse button was pressed.
wCalcRgns	2	Calculate wStrucRgn and wContRgn.
wNew	3	Complete the creation of a window.
wDispose	4	Complete the disposal of a window.
wGetDrag	5	Return address that will draw the outline of the window while dragging.
wGrowFrame	6	Draw the outline of a window being resized.
wRecSize	7	Return size of the additional space needed in the window record.
wPosition	8	Return RECT that is the window's portRect.
wBehind	9	Return where the window should be placed in the window list.
wCallDefProc	10	Generic call to a defProc, defined by the defProc.

wDraw, Operation 0

The wDraw operation draws the window's frame and is only called for visible windows. This operation draws in local coordinates in the current GrafPort, which is the Window Manager's GrafPort. When the drawing is finished, the only states of the GrafPort that may have changed are the pen pattern, the fill pattern, and the pen size, as all other states must be the same as when the defProc was called. This means that if you change the font to print some text, you must save and restore the original font. For the pen, PenNormal will restore the pen to an acceptable state.

Param is defined as follows:

```
    Bit 31        1 to highlight the indicated part, 0 to unhighlight.
    Bits 0-30     The part to draw (either highlighted or unhighlighted):
                  0    Draw the window's entire frame, including any frame
                       controls and the items listed below.  Note that you
                       should check the window's fHilited flag to determine
                       how to draw the frame.
                  1    Draw the go-away region.
                  2    Draw the zoom region.
                  3    Draw the information bar.
```

Result returned must be zero and the carry flag must be clear.

The Window Manager will draw the content.

Need to Redraw Your Window?

If your custom window defProc gets called to change some item in its window record (see wCallDefProc below), you may want to redraw your window. For instance, if your application makes a SetWTitle call, you would want to draw the name of the new title on the screen.

The routine wCallDefProc can call the wDraw routine to do this drawing. However, it should bracket the calls to wDraw with two Window Manager calls that save and restore some internal variables:

```
    StartFrameDrawing    $5A0E
    PUSH:LONG            Pointer to the window record (not the GrafPort)
```

This call does the setup for drawing a window frame and is only called by a window definition procedure before drawing the frame. You should call EndFrameDrawing when finished drawing.

```
    EndFrameDrawing      $5B0E
    No input or output
```

This call restores the Window Manager variables after a call to StartFrameDrawing and is only called by a window definition procedure after drawing a window frame.

wHit, Operation 1

The wHit operation is called to hit test the window's frame. Given a set of screen coordinates, this operation should return what part, if any, of the window is at that coordinate. This operation is only called for visible windows. The current port will be that of the Window Manager and the window frame will be in local coordinates.

Param is defined as:

```
    Bits 0-15     Vertical (Y) coordinate in local coordinates.
    Bits 16-31    Horizontal (X) coordinate in local coordinates.
```

Result returned must be one of the following values and the carry flag must be clear:

```
    wNoHit          0    Not on the window at all.
    wInDrag        20    Coordinates are in the window's drag region (title bar).
    wInGrow        21    Coordinates are in the window's grow region (size box).
    wInGoAway      22    Coordinates are in the window's go-away region (close
box).
    wInZoom        23    Coordinates are in the window's zoom region (zoom box).
    wInInfo        24    Coordinates are in the window's information bar.
    wInFrame       27    Coordinates are in the window, but not in any of the
                         other areas.
                   xx    Any code the application can handle (bit 15 is
                         reserved for theWindow Manager)
```

wCalcRgns, Operation 2

The wCalcRgns operation, which is called only for visible windows, is used to calculate the window's entire region (frame plus content called StrucRgn) and just its content region (called ContRgn). Both regions must be set to global coordinates, and both will already be allocated with their handles stored in the window record's wStrucRgn and wContRgn fields.

Use the portRect and the boundsRect of the window's GrafPort to calculate these two regions. The port will have been set from the information passed to NewWindow along with any size changes. A method for obtaining the global RECT of the content is given below. Refer to the QuickDraw II chapter in the Apple IIGS Toolbox Reference for a full description of ports. When calculating the regions, do not change the clip region (ClipRgn) or the visible region (VisRgn) of the GrafPort.

Param is not defined and should not be used.

Result returned must be zero and the carry flag must be clear.

```
    IN:    window = pointer to window record.
    OUT:    rect = global RECT of window's content.

        ldy    #wPort+portRect+y1
        lda    [<window],y
        ldy    #wPort+portInfo+boundsRect+y1
        sec
        sbc    [<window],y
        sta    <rect+y1
    ;
        ldy    #wPort+portRect+x1
        lda    [<window],y
        ldy    #wPort+portInfo+boundsRect+x1
        sec
        sbc    [<window],y
        sta    <rect+x1
    ;
        ldy    #wPort+portRect+y2
        lda    [<window],y
        ldy    #wPort+portInfo+boundsRect+y1
        sec
        sbc    [<window],y
        sta    <rect+y2
    ;
```

```
        ldy    #wPort+portRect+x2
        lda    [<window],y
        ldy    #wPort+portInfo+boundsRect+x1
        sec
        sbc    [<window],y
        sta    <rect+x2
```

Although there are other ways to obtain the global RECT of the content, this example gives the correct method. You should never rely on the top and left side of the portRect being zero.

wNew, Operation 3

The wNew operation is called to perform any additional initialization that may be required for a custom window. The following items are already done for the window:

- If a window record is supposed to be allocated, it is. All fields, other than those fields listed below, are set to zero
- A port opens in the window record's wPort field.
- The window is added to the Window Manager's window list, and the wNext field is set.
- The wDefProc, wStrucRgn, wContRgn and wUpdate regions are set with the handles of the allocated regions. It is the responsibility of the defProc to define the shape of the wStrucRgn and wContRgn regions.
- The fAllocated and fHilited bits in the wFrame field of the window record are set (see the window record definition for a definition of these bits) and should not be disturbed; all other bits in wFrame are set to zero. The defProc should set the fCtlTie, fVis and fQContent bits, and it can set and use other bits in the wFrame field as it wishes.
- It is the responsibility of the defProc to set the wRefCon, wContDraw, and wFrameCtls fields, the bits already mentioned in the wFrame field, and any other fields which it defines in the wCustom part of the window record.

Param is a pointer to the parameter list pointer which was passed to NewWindow.

Result returned must be zero and the carry flag must be clear.

wDispose, Operation 4

The wDispose operation is called to perform any additional disposal that may be required of a custom window. This operation is called before the Window Manager performs any disposal actions on the window.

Param is not defined and should not be used.

Result should be FALSE to continue disposal or TRUE to abort the disposal. In either case, the carry flag should be clear. Returning TRUE would be very unusual and should be carefully thought out. After returning FALSE, the Window Manager will erase the window, remove the window from the Window Manager's window list, free any controls in the window's wControls and wFrameCtl lists, free the handles in the wStrucRgn, wContRgn and wUpdateRgn fields, close the window's GrafPort, and free its record if it is allocated (see the wFrame field).

wGetDrag, Operation 5

The wGetDrag operation is called to get the address of a routine that will draw an outline of the window.

Param is not defined and should not be used.

Result returned must be the address of a frame outline routine or zero for a default frame; the default frame is the bounds RECT of the strucRgn. The frame outline routine is called from DragRect with dragRectPtr set to the bounds RECT of the strucRgn. Your routine is called with the following parameters:

```
    PUSH:WORD - delta X
    PUSH:WORD - delta Y
    PUSH:BYTE[3] - return address
```

Your routine should draw or erase the outline of the object in its new position using the passed deltas. You have several different methods of determining whether to erase or draw and how to compute the position of the object, the easiest method being to draw the outline using XOR mode. The first time your routine is called, you draw. The next time your routine is called, you erase. Your routine should draw in the current port. The current pen pattern will be the pattern pointed to by dragPatternPtr from DragRect and the pen mode is XOR.

You also need to know where to draw the outline. One way is to offset the starting RECT (dragRectPtr) by the given deltas. You should make a copy of the bounds RECT of the strucRgn when wGetDrag is called. Modify that rectangle with the deltas to obtain the rectangle to frame.

wGrowFrame, Operation 6

The wGrowFrame operation is called to draw an outline of the window when the window is being resized.

This operation should use the current port, pen pattern, and pen mode. The frame should be drawn with only the following QuickDraw II calls: Line, LineTo, FrameRect, FrameRgn, FramePoly, FrameOval, FrameRRect, and FrameArc (the Invert equivalents to Frame could also be used). You want to use the current GrafPort setting with only certain QuickDraw II calls since this routine will be called an even number of times; the first time it is called to draw the frame and the next time to erase that which it drew the first time. If it needs to use QuickDraw II calls other than those listed above, this operation handler could keep track of odd and even calls to know whether to draw or erase the frame.

Param is a pointer to the following parameter list:

newSize	RECT	Rectangle that defines the new size.
drawFlag	WORD	TRUE to draw the frame, FALSE to erase.
startRect	RECT	Bounds of wStrucRgn when dragging started.
deltaY	WORD	Vertical movement since starting to drag (signed).
deltaX	WORD	Horizontal movement since starting to drag (signed).

Result should be:

```
 ____________...____________________________
| 31| 30| 29|...| 6 | 5 | 4 | 3 | 2 | 1 | 0 |
 ____________...____________________________
                                     |   |
          TRUE if newSize RECT has been   +-- TRUE if frame drawn,
          recomputed, FALSE if newSize        FALSE to draw default frame.
          RECT OK.
```

The Window Manager assumes that the frame of the grow outline is the same as the bounds of the window's wStrucRgn. This RECT is stored in the startRect of the parameter list and does not change through out the dragging. The next assumption is that the window grows from the lower right corner. As the cursor moves, the lower right corner of the RECT in newSize changes. However, if these assumptions are not correct for a custom window they can be overridden by changing the RECT in newSize (by using startRect or the window's record and the deltas) and returning TRUE for bit 1 in Result. The carry flag should return clear.

wRecSize, Operation 7

The wRecSize operation is called to ask how large a window record should be allocated.

Note: The window pointer passed in theWindow is not valid for this call.

Param is the parameter list pointer that is passed to NewWindow.

Result is the number of additional bytes required in the window record. The standard window record header will always be allocated.

Example:

If your custom window needs a one word field in the window record for your own use you would return 2 in Result. The Window Manager takes Result and adds to it the size of the standard record header of 212 bytes and allocates a window record that is 214 bytes long in this case. Your one word field is at the end of the standard window record header with an offset of 212 bytes.

If there is some error, return the carry flag set with an error code in the y register, which will cause NewWindow to abort and return the error code to the application which called it. If there is no error, return the carry flag clear.

Window Record Already Allocated?

If the window record is already allocated then Result should be the pointer to the window record with bit 31 of the pointer set to TRUE. Generally, window records are allocated (refer to Window Record Definition at the end of this Note for more information about window records).

wPosition, Operation 8

Param is the parameter list pointer that is passed to NewWindow.

Result is a pointer to the RECT that will be the window's portRect, and you should return the carry flag clear.

wBehind, Operation 9

Param is the parameter list pointer that is passed to NewWindow.

Result is where the window should be placed in the window list. A long $FFFFFFFF means insert the window as the top window while a long $00000000 means to insert it as the bottom window. Any other value is a pointer to the window behind which this window should be placed. You should return the carry flag clear.

wCallDefProc, Operation 10

WCallDefProc is a generic call to the defProc that is defined by the defProc. With this call a window defProc can define many special functions.

The input to the defProc is:

```
    param = pointer to the following parameter table:

    dRequest    WORD       Requested operation number.
    paramID     WORD       Parameter block type:
                           $0000-$7FFF reserved by system ($0000 defined below).
                           $8000-$FFFF reserved for custom defProcs.
    newParam    BYTE[n]    New parameter field used by some operations.
```

The paramID field defines dRequest, which in turn defines newParam and the result of the wCallDefProc call. You can think of dRequest as the operation number passed to the defProc. Here is an example of how the paramID defines dRequest: if paramID is zero, dRequest 3 is defined as wSetPage (defined below); but if paramID is $8345 (or any number other than zero), dRequest 3 could be defined as something entirely different.

The following dRequest values are defined for wCallDefProc operations with a paramID of zero. Your defProc should check for handling only these codes. In the future, codes 34 and greater may be defined, and your defProc should know not to handle them.

wSetOrgMask	0	wGetInfoDraw	17
wSetMaxGrow	1	wGetOrigin	18
wSetScroll	2	wGetDataSize	19
wSetPage	3	wGetZoomRect	20
wSetInfoRefCon	4	wGetTitle	21
wSetInfoDraw	5	wGetColorTable	22
wSetOrigin	6	wGetFrameFlag	23
wSetDataSize	7	wGetInfoRect	24
wSetZoomRect	8	wGetDrawInfo	25
wSetTitle	9	wGetStartInfoDraw	26
wSetColorTable	10	wGetEndInfoDraw	27
wSetFrameFlag	11	wZoomWindow	28
wGetOrgMask	12	wStartDrawing	29
wGetMaxGrow	13	wStartMove	30
wGetScroll	14	wStartGrow	31
wGetPage	15	wNewSize	32
wGetInfoRefCon	16	wTask	33

```
wSetOrgMask                 0
    newParam    =    WORD - window's origin mask.
    result      =    None.

    Called when SetOriginMask is called.

wSetMaxGrow                 1
    newParam    =    WORD - maximum window height.
                     WORD - maximum window width.
    result      =    None.

    Called when SetMaxGrow is called.

wSetScroll                  2
    newParam    =    WORD - number of pixels to scroll when arrow is
                            selected.
    result      =    None.

    Called when SetScroll is called.

wSetPage                    3
    newParam    =    WORD - pixels to scroll when page region is selected.
    result      =    None.

    Called when SetPage is called.

wSetInfoRefCon              4
    newParam    =    LONG - value passed to info bar draw routine
                            (app's use only).
    result      =    None.

    Called when SetInfoRefCon is called.

wSetInfoDraw                5
    newParam    =    LONG - address of info bar draw routine.
    result      =    None.

    Called when SetInfoDraw is called.

wSetOrigin                  6
    newParam    =    WORD - flag, TRUE to scroll content.
                     WORD - window's Y origin.
                     WORD - window's X origin.
    result      =    None.

    Called when SetContentOrigin is called.

wSetDataSize                7
    newParam    =    WORD - height of window's data area.
                     WORD - width of window's data area.
    result      =    None.

    Called when SetDataSize is called.

wSetZoomRect                8
    newParam    =    LONG - pointer to new zoom RECT.
    result      =    None.
```

```
    Called when SetZoomRect is called.

wSetTitle                  9
    newParam    =    LONG - pointer to new title.
    result      =    None.

    Called when SetWTitle is called.

wSetColorTable            10
    newParam    =    LONG - pointer to new color table.
    result      =    None.

    Called when SetFrameColor is called.

wSetFrameFlag             11
    newParam    =    LONG - pointer to new zoom RECT.
    result      =    None.

    Called when SetWFrame is called.

wGetOrgMask               12
    newParam    =    None.
    result      =    WORD - window's origin mask.

wGetMaxGrow               13
    newParam    =    None.
    result      =    Low word is window's maximum height when grown.
                     High word is window's maximum width when grown.

    Called when GetMaxGrow is called.

wGetScroll                14
    newParam    =    None.
    result      =    Low word is number of pixels to scroll when arrow is
                     selected.

    Called when GetScroll is called.

wGetPage                  15
    newParam    =    None.
    result      =    Low word is pixels to scroll when page region is selected.

    Called when GetPage is called.

wGetInfoRefCon            16
    newParam    =    None.
    result      =    Value passed to info bar draw routine.

    Called when GetInfoRefCon is called.

wGetInfoDraw              17
    newParam    =    None.
    result      =    Address of info bar draw routine.

    Called when GetInfoDraw is called.

wGetOrigin                18
```

```
    newParam    =    None.
    result      =    Low word is content's Y origin.
                     High word is content's X origin.

    Called when GetContentOrigin is called.

wGetDataSize              19
    newParam    =    None.
    result      =    Low word is window's data height.
                     High word is window's data width.

    Called when GetDataSize is called.

wGetZoomRect              20
    newParam    =    None
    result      =    Pointer to window's current zoom RECT.

    Called when GetZoomRect is called.

wGetTitle                 21
    newParam    =    None
    result      =    Pointer to window's title.

    Called when SetWTitle is called.

wGetColorTable            22
    newParam    =    None.
    result      =    Pointer to window's color table.

    Called when SetFrameColor is called.

wGetFrameFlag             23
    newParam    =    None.
    result      =    Low word is window's wFrame field.

    Called when SetWFrame is called.

wGetInfoRect              24
    newParam    =    LONG - pointer to place to store info bar's enclosing RECT.
    result      =    None.

    Called when GetRectInfo is called.

wGetDrawInfo              25
    newParam    =    None.
    result      =    None.

    Called when DrawInfoBar is called.

wGetStartInfoDraw         26
    newParam    =    LONG - pointer to place to store info bar's enclosing
                     RECT.
    result      =    None.

    Called when StartInfoDrawing is called.

wGetEndInfoDraw           27
    newParam    =    None.
```

```
    result      =    None.

    Called when EndInfoDrawing is called.

wZoomWindow                 28
    newParam    =    None.
    result      =    None.

    Called when ZoomWindow is called.

wStartDrawing               29
    newParam    =    None.
    result      =    None.

    Called when StartDrawing is called.

wStartMove                  30
    newParam    =    WORD - new y position (global).
                     WORD - x position (global).
    result      =    Low word is new y position (global).
                     High word is x position (global).

    Called before MoveWindow moves a window.

wStartGrow                  31
    newParam    =    None.
    result      =    None.

    Called before GrowWindow tracks the growing of a window.

wNewSize                    32
    newParam    =    LONG - pointer to:
                     WORD - proposed new height.
                     WORD - proposed new width.
                     These two values can be changed.
    result      =    Low word TRUE if only uncovered content should be drawn.
                     FALSE if entire content should be redrawn.

    Called by SizeWindow before it resizes a window.  The new height and
    width can be changed by modifying the words pointed to by the pointer in
    newParam.

wTask                       33
    newParam    =    LONG - pointer to task record.
                     WORD - result from FindWindow.
    result      =    Low word is code returned by TaskMaster (zero if handled).
                     High word is task performed.  Returned in TaskData if code
                     is 0.

    Called from TaskMaster when it cannot handle a task.  If the user
    presses the mouse button over a window, TaskMaster will call FindWindow
    to find out what part of the window.  TaskMaster will then handle the
    task if FindWindow returns wInMenuBar or bit 15 of the window pointer is
    set (system window).  Otherwise, the result of FindWindow is passed to
    wTask to be handled or not.

    If the defProc can handle the task it should do so and return zero in
    the low word of the result (which will be the result to the application
```

returned from TaskMaster) and a code of the task performed in the high word of the result (which is returned to the application in its task record TaskData field). Fields in the task record may also be modified to return parameters to the application as this is the same record passed to TaskMaster.

If the defProc cannot handle the task, it should return the result from FindWindow (the second field in newParam) in the low word of the result. The high word of the result is not used.

For example, the standard document window defProc handles the following results from FindWindow if the taskMask record allows.

wInContent	Brings the window to the top.
wInDrag	Calls DragWindow.
wInGrow	Brings the window to the top. If it is already on the top, it calls GrowWindow and SizeWindow.
wInGoAway	Calls TrackGoAway.
wInZoom	Calls TrackZoom and ZoomWindow.
wInInfo	Brings the window to the top.
wInFrame	Brings the window to the top. If it is already on the top, checks if it is on one of the window's scroll bars, tracks it, and scrolls the window's content as needed.

A custom window defProc can return any code (bit 15 is used for system windows) it wants when it is called to do a hit test. This code would be that returned by FindWindow, and the application would have to know about the code if it called FindWindow instead of TaskMaster. If TaskMaster is used, the code that FindWindow returns is passed back to your defProc with a wCallDefProc and wTask. The defProc could perform any task it wanted: change colors, eject a disk, run a spelling checker, or anything else.

Window Record Definition

0	wNext	LONG - Pointer to next window record, zero is end of list.
4	wPort ///	BYTE[170] - Window's GrafPort.
174	wDefProc	LONG - Address of window's definition procedure.
178	wRefCon	LONG - Reserved for application's use.
182	wContDraw	LONG - Address of routine that will draw window's content.
186	wReserved	LONG - Reserved by the Window Manager, do not use.
190	wStrucRgn	LONG - Handle of window's structure region.
194	wContRgn	LONG - Handle of window's content region.
198	wUpdateRgn	LONG - Handle of window's update region.
202	wCtls	LONG - Handle of first control in window's content.

```
   206 |wFrameCtls             | LONG - Handle of first control in
       |_______________________|        window's frame.
   210 |wFrame      |            WORD - Flags that define window.
       |____________|
   212 |wCustom    ...           BYTE[n] - Additional data space defined by
       |____________...                    window's definition procedure.
```

The changes use some vacant space under the window port and add the wReserved field to the record for future expansion.

In addition to defining the window record, the wFrame field needs to be further defined. In the diagram below the shaded bits are reserved for use by each window defProc (the values shown are those used by the standard document window defProc). Bits not shaded are reserved by the Window Manager and are applicable to all windows.

Further Reference

- o Apple IIGS Toolbox Reference, Volume 1
- o System Disk 4.0 Release Notes

END OF FILE TN.IIGS.042

##
FILE: TN.IIGS.043
##

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#43: Undocumented Feature of CalcMenuSize

Written by: Dan Oliver November 1988

This Technical Note documents that CalcMenuSize can accept a parameter of $FFFF to recalculate menus with uninitialized heights and widths.

The newWidth and newHeight parameters of CalcMenuSize can be the actual new width and height or zero to automatically compute the width and the height. In addition to these two possibilities, you can also pass $FFFF in newWidth, newHeight, or even both to tell CalcMenuSize to automatically compute the width and height only if the current setting is zero. Here are some examples of how these three, previously undocumented features might be used:

Pass a New Value

Pass a new value when you need the width and height to be a specific size.

Pass $0000

Pass $0000 when you want to compute a new value regardless of the current value. You could compute a new value after you add or delete a new item or change its title since the current values for width and height are set to the current size which may be incorrect due to the change. Passing $0000 adjusts the menu size according to the new menu items.

Pass $FFFF

Pass $FFFF when you want to compute a new value only when the current value is zero, as when the menu is first created. FixMenuBar passes $FFFF to compute sizes for only those menus with widths and heights of zero.

Further Reference
o Apple IIGS Toolbox Reference, Volume 1

END OF FILE TN.IIGS.043

```
#####################################################################
### FILE: TN.IIGS.044
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#44: GetPenState and SetPenState Record Error

Written by: Keith Rollin November 1988

This Technical Note corrects an error in the record used for GetPenState and SetPenState.

The Apple IIGS Toolbox Reference, Volume 2 incorrectly describes the record used to save and restore information about the drawing pen in Figure 16-38 on page 16-238. The include files for the Apple Programmer's Workshop (APW) and the Macintosh Programmer's Workshop IIGS (MPW IIGS) assemblers and C compilers also reflect this error. The correct record is as follows:

```
     Offset       Field                     Description
                  --------------------
        $0       |__               __|
        $1       |__    psPenLoc   __|    LONG - Point specifying pen
        $2       |__               __|           location
        $3       |____________________|
        $4       |__               __|
        $5       |__   psPenSize   __|    LONG - Point specifying pen size
        $6       |__               __|
        $7       |____________________|
        $8       |__   psPenMode   __|    WORD - Pen mode
        $9       |____________________|
        $A       |                    |
                 |                    |
                 |      psPenPat      |    32 bytes - Pen pattern
                 |                    |
       $21       |____________________|
       $22       |                    |
                 |                    |
                 |     psPenMask      |     8 bytes - Pen mask
                 |                    |
       $31       |____________________|
       $32
```

END OF FILE TN.IIGS.044

```
#####################################################################
### FILE: TN.IIGS.045
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#45: Parameters for GetFrameColor

Revised by: Matt Deatherage — September 1989
Written by: Dan Oliver — November 1988

This Technical Note formerly attempted to correct the description of the parameters passed to and returned from the routine GetFrameColor in the Window Manager chapter of the Apple IIGS Toolbox Reference. This call works as documented since System Software 3.2; therefore, former versions of this Note were incorrect.
Changes since November 1988: Corrected our error. Sorry for any inconvenience.

This Note formerly stated the following: "The Apple IIGS Toolbox Reference, Volume 2 incorrectly describes the parameters passed to and returned from GetFrameColor on page 25-57."

However, this is incorrect. Beginning with System Software 3.2, GetFrameColor works as documented in the Apple IIGS Toolbox Reference, Volume 2. Prior to System Software 3.2, the call did not work at all. We apologize for any inconvenience this confusion may have caused.

Further Reference

- Apple IIGS Toolbox Reference, Volume 2

END OF FILE TN.IIGS.045

```
#####################################################################
### FILE: TN.IIGS.046
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#46: DrawPicture Data Format

Written by: Jeff Erickson & Keith Rollin November 1988

This Technical Note describes the internal format of the QuickDraw II picture data structure.

This Technical Note presents the internal format of the QuickDraw II picture data structure for informational purposes only. You should not use this information to write your own bottleneck procedures; the only routines which should create and read PICT format files are those provided in QuickDraw II. If we added new objects to the picture definition, your program would not operate on new pictures. This Note documents this information for debugging purposes only.

Picture Data Structure Definition

Pictures are stored in memory in the following format:

They begin with a WORD which indicates the mode of the port which was used to record when the picture was created. This information is useful when the picture is played back, possibly in a different graphics mode.

Following the WORD is a RECT which indicates the frame of the picture and is used for scaling when you redraw the picture. Following the RECT is the version number of this PICT format, then a series of word-sized opcodes which describe the sequences of QuickDraw II commands that were used to create the picture.

Name	Description	Size (bytes)
pictSCB	picture's scan line control byte	2 (high byte = 0)
picFrame	picture's boundary rectangle	8
version	picture version	2 (Currently $8211)
opcode	operation code	2
<data>	operation data	variable, depending on opcode
:		
opcode	operation code	2
<data>	operation data	variable, depending on opcode

Opcodes

As mentioned above, pictures are described by a series of opcodes which are used to record the QuickDraw II commands that created the picture. These opcodes are two bytes long and are usually followed by a number of parameters.

All currently defined opcodes and their parameters are listed below. Any opcodes not listed here are reserved.

Opcode	Name	Description	Parm Bytes	Parameter Description
$0000	NOP	no operation	0	none
$0001	ClipRgn	clip to a region	[region size]	region
$0002	BkPat	background pattern	32	background pattern (8x8 pixels)
$0003	TxFont	text font	4	Font Manager font ID (long)
$0004	TxFace	text face	2	text face (word)
$0005	TxMode	text mode	2	text mode (word)
$0006	SpExtra	space extra	4	space extra (fixed)
$0007	PnSize	pen size	4	pen size (point)
$0008	PnMode	pen mode	2	pen mode (word)
$0009	PnPat	pen pattern	32	pen pattern (8x8 pixels)
$000A	FillPat	fill pattern	32	fill pattern (8x8 pixels)
$000B	OvSize	oval size	4	oval size (point)
$000C	Origin	origin	4	origin (point)
$000D	TxSize	text size	2	text size (word)
$000E	FGColor	foreground color	2	color (word)
$000F	BGColor	background color	2	color (word)
$XX11	Version	version	0	none: high byte=version (currently $82)
$0012	ChExtra	character extra	4	char. extra (fixed)
$0013	PnMask	pen mask	8	mask (8 bytes)
$0014	ArcRot	arc rot	2	Reserved (related to things drawn with patterns). (word)
$0015	FontFlags	font flags	2	font flags (word)
$0020	Line	line	8	pnLoc (point), newPt (point)
$0021	LineFrom	line from pen loc.	4	newPt (point)
$0022	ShortLine	short line	6	pnLoc (point), dv, dh (signed bytes)
$0023	ShortLFrom	ditto from pen loc	2	dv, dh (signed bytes)
$0028	LongText	long text	5+text	txLoc (point), count (byte), text
$0029	DHText	hor. offset text	2+text	dh (unsigned byte), count (byte), text
$002A	DVText	vert. offset text	2+text	dv (unsigned byte), count (byte), text
$002B	DHDVText	offset text	3+text	dv, dh (unsigned bytes), count (byte), text
$002C	RealLongText	very long text	6+text	txLoc (point), count (word), text

Opcodes between $0030 and $008C are a combination of a graphic verb and a graphic object, as listed below (where "V" stands for the graphic verb, and "X" is a stands for the graphic object). For example, $0069 means PaintSameArc, and is followed by two one-word parameters.

Graphic Verbs:

$00X0	Frame...	frame something	[Specific to object type

			see below.]
$00X1	Paint...	paint something	
$00X2	Erase...	erase something	
$00X3	Invert...	invert something	
$00X4	Fill...	fill something	
$00XV+8	...Same...	draw same thing somehow	[See below; {braced} parms do not appear.]

Graphic Objects:

$003V	...Rect	draw a rectangle somehow	8 (0 if - SameRect) {rect (2 points)}
$004V	...RRect	draw a round rect somehow	8 (0) {rect (2 points)}
$005V	...Oval	draw an oval somehow	8 (0) {rect (2 points)}
$006V	...Arc	draw an arc somehow	12 (4) {rect (2 points)}, start, arc angle (words)
$007V	...Poly	draw a polygon somehow	[polygon size] (0){polygon}
$008V	...Rgn	draw a region somehow	[region size] (0) {region}
$0090	BitsRect	copybits, rect clipped	variable* (see below, but without maskRgn)
$0091	BitsRgn	copybits, rgn clipped	variable* (see below)
$00A1	LongComment	long comment	4+data kind (word), size (word), data

*Bits... data:

origSCB	original scan line control byte	2	SCB (word -- high byte = 0)
BWvsColor	black and white vs. color	2	reserved (word)
width	width of pixel image in bytes	2	width (word)
boundsRect	bounds rectangle	8	rect (2 points)
srcRect	source rectangle	8	rect (2 points)
destRect	destination rectangle	8	rect (2 points)
mode	transfer mode	2	pen mode (word)
maskRgn	mask region (BitsRgn ONLY!)	[region size]	region
pixData	pixel image	[pixdata size]	width* (bounds.bottom-bounds.top)

Differences Between IIGS Pictures and Macintosh Pictures

1. QuickDraw II pictures are modeled after PICT2 on the Macintosh, which use two bytes for its opcodes and data (the exception to this is the $11 (version) opcode, which is followed by a one-byte parameter). Macintosh PICT 1.0 formats, which use one-byte opcodes, would have to undergo extensive modifications to be displayed on the IIGS.
2. There is no EndOfPicture opcode on the IIGS as there is on the Macintosh. Also, the first word of the picture is a pictSCB, not the length of the picture. The picture size is determined solely by the size of the handle on the IIGS. There is also no picture header on the IIGS as on the Macintosh.
3. The number sex of the Macintosh is opposite that of the Apple IIGS. The Macintosh stores the high bytes of words and long words first, whereas the IIGS stores the low byte first.
4. The following Macintosh picture opcodes are not available on the IIGS: txRatio, PackBitsRect, PackBitsRgn, shortComment,

EndOfPicture.

5. QuickDraw II defines the following opcodes that the Macintosh does not: ChExtra ($12), PnMask ($13), ArcRot ($14), FontFlags ($15), and RealLongText ($2C).

Notes on the Interpretation of IIGS Pictures

- The state of the pen, the clip region, various patterns and colors, and the origin of the current port is saved before a picture is drawn, and restored afterwards. The current port is set up in a default state equivalent to that of a newly created port just before drawing begins. Picture opcodes act just like their QuickDraw II tool counterparts, with a few exceptions.
- Two pen locations are tracked as the picture is drawn, one for lines and one for text. Thus, LineFrom always draws from the end of the last line, regardless of any intermediate text opcodes.
- Text calls do not change the position of the "text pen," as do normal QuickDraw II text calls. Thus, if a picture contains two lines of text, the second one directly below the first, the second will be stored using a DVtext opcode.
- DrawPicture performs considerable setup before it draws pictures. Among other things, it calls InstallFont, which is a Font Manager call. If you are going to support pictures in your application, you should load and start the Font Manager.

Further Reference

- Apple IIGS Toolbox Reference, Volume 2

END OF FILE TN.IIGS.046

\###
\### FILE: TN.IIGS.047
\###

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#47: What SetDataSize Does

Written by: Keith Rollin November 1988

This Technical Note clears up any ambiguity in the description of the SetDataSize call.

The Apple IIGS supports windows that contain scroll bars in their frames. These scroll bars are handled by TaskMaster and differ from Macintosh scroll bars in that the size of the "thumb" or "elevator" is used to indicate the size of the visible area of the document in relation to the total size of the document (the "data size"). Initially, the visible size and the data size are defined by the parameter list passed to NewWindow; however, either of these can be changed by SizeWindow and SetDataSize, respectively.

SetDataSize is used to not only change the range of scrolling allowed, but also to redraw the size of the thumb to reflect the fact that the data size has changed with respect to the visible area. However, page 25-97 of the Apple IIGS Toolbox Reference contains the following description of SetDataSize:

> "Sets the height and width of the data area of a specified window. Setting these values will not change the scroll bars or generate update events."

When the manual states that SetDataSize "will not change the scroll bars," it is referring to the location, or value, of the thumb. Assume a situation where you have a word processor that scrolls the page using TaskMaster scroll bars. If you delete a range of text, you would also shorten the entire size of the document. Calling SetDataSize to reflect that would indeed change the size of the thumb, but it would not change its location. If you were already scrolled to the bottom of the document when you called SetDataSize, the thumb would become larger (to reflect the fact the the total data size became smaller with respect to the visible data size) and overwrite the down arrow of the scroll bar. To prevent this situation from occurring, you should also change the origin of the window with SetContentOrigin before calling SetDataSize.

Further Reference
o Apple IIGS Toolbox Reference, Volume 2

\### END OF FILE TN.IIGS.047

```
#####################################################################
### FILE: TN.IIGS.048
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#48: All About AlertWindow

Revised by: Dave Lyons July 1989
Written by: Dan Oliver & Keith Rollin November 1988

This Technical Note documents a new call in the Window Manager which eases the creation of Alert windows.
Changes since November 1988: Documented new features and behavior of AlertWindow in System Software 5.0.

AlertWindow is available in Window Manager version 2.2 and later (Apple IIGS System Disk 3.2 and later). This call takes three parameters which are used to create a dialog box with text, buttons, and an optional icon.

AlertWindow works by taking a pointer to an "alert string." This alert string defines the size and location of the alert window, specifies what icon (if any) it uses, defines the text it displays, and indicates the number of buttons and their names which it displays.

The alert string is a very powerful and complicated structure to be able to specify all of this information, and it is even more powerful with the added capability of "substitution strings." Substitution strings work in a manner similar to ParamText substitutions; certain sections of the text are designated as variables to be replaced by other text when you display the dialog.

The Call

AlertWindow ($590E)

```
    input:     WORD    Space for result.
               WORD    alertFlags.  Bit 0 is 0 if substitution strings are C-
                       type strings (NULL terminated), 1 if they are Pascal-
                       type strings (leading length byte).  Bits 1 and 2 are
                       00 if alertStrRef is a pointer, 01 if it is a handle,
                       and 10 if it is a resource ID (the resource type is
                       $8015).
               LONG    Pointer to substitution string array; anything if no
                       substitutions are to be made.
               LONG    alertString (see alertFlags)
    output:    WORD    Button number that was selected where 0 is the first
                       button title in the alert string, 1 is the next button
                       title, and 2 is the third.
```

The Format of the Alert String

Size Character

The first character of the alert string is the size character, which indicates the size of the alert window. The character can be ASCII 1-9 to indicate any one of nine standard alert window sizes. The goal of standard sizes is to have alert windows that can contain the same amount of message text in 320 mode or 640 mode, without changing the size. However, making this happen may require some careful message and button text composition.

Size Character	Approximate Maximum # of Characters
1	30
2	60
3	110
4	175
5	110
6	150
7	200
8	250
9	300

The following table shows the dimensions of the standard alert windows. This table gives an idea of the size of each window. Application code should not rely on the exact widths, heights, or position of standard windows. If an application needs an exact window size, it can specify zero as the size character and use the next eight bytes as four word-sized integers which specify the rectangle of the window.

Character	Height 320	Width 320	Height 640	Width 640
1	46	152	46	200
2	62	176	54	228
3	62	252	62	300
4	90	252	72	352
5	54	252	46	400
6	62	300	54	452
7	80	300	62	500
8	108	300	72	552
9	134	300	80	600
0	(Character followed by 4 integers that represent size and position.)			

Icon Number

The next character is the icon number. The icon number can be 0-9 where:

```
    0 =        No icon.
    1 =        custom icon, followed by:
               LONG    Pointer to image data.
               WORD    Number of bytes image data is wide.
               WORD    Number of scan lines image data is high.
    2 =        Stop icon.
    3 =        Note icon.
    4 =        Caution icon.
```

5 = Disk icon.
6 = Disk swap icon.
7 - 9 = Reserved--do not use.

Separator Character

The next character is a separator character. The separator can be any character and cannot appear in the message text or button strings. The separator is used to separate the message from the first button string and each button string from each other. For purposes of standardization the slash (/) might be a good choice, unless you will be substituting pathnames (see the section "Don't Use Separator Characters in Substitution Strings").

Message Text

Following the separator character is the message text. Any characters which LETextBox2 allows are valid in the message text. See the section "Special Characters" for additional message text functions. The total size of the message text, after string substitution, is limited to 1,000 characters.

Button Strings

The first character after the separator character following the message text is the beginning of the first button's title. The title can be followed with either another separator character and button title or a string termination character (i.e., zero (0)) to end the alert string. A total of three button titles may be included at the end of the alert string. These buttons are evenly spaced and centered at the bottom of the alert window. The width of each button is the same size and is set according to the widest button title. The total size of the button text, after string substitution, is limited to 80 characters.

Termination of Alert String

A zero ($00) comes after the last button title to end the alert string.

The Substitution Strings

The message text and button strings can contain special characters that are replaced by substitution strings when you display the alert window.

Special Characters

The following special characters can be embedded in the message text and button strings of an alert. If you want the special character itself to appear in the body of the text, enter it twice in the string.

^ If a caret (^) is the first character in a button string, the button is the default button. The default button is the button selected if the user presses the Return key or the Enter key. This button also has a bold outline on the screen. Only one button can be the default button. After the caret, the button title must follow like any other title (including other special characters).

Note: If the caret is to appear in the message text, it must be entered twice, side-by-side. A single caret in message text has no effect and is deleted from the message.

#	Substitute with a standard string. The pound sign (#) must be followed by an ASCII character. Characters 0-6 can be used (7-9 are reserved and should not be used) where:

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

*	Substitute with the given string. The asterisk (*) followed by ASCII character in the range 0-9 denotes that a substitution string should be inserted at that point. The asterisk and the following character are replaced by the corresponding string in the substitution array. A pointer to the substitution array is one of the parameters passed to AlertWindow and is defined as an array of LONG pointers where:

LONG[0]	Pointer to the string that substitutes for *0.
LONG[1]	Pointer to the string that substitutes for *1.
LONG[2]	Pointer to the string that substitutes for *2.
LONG[3]	Pointer to the string that substitutes for *3.
LONG[4]	Pointer to the string that substitutes for *4.
LONG[5]	Pointer to the string that substitutes for *5.
LONG[6]	Pointer to the string that substitutes for *6.
LONG[7]	Pointer to the string that substitutes for *7.
LONG[8]	Pointer to the string that substitutes for *8.
LONG[9]	Pointer to the string that substitutes for *9.

Substitution strings can be a C-type (NULL-terminated), Pascal-type (a string prefixed with a length byte), or Return-terminated. NULL- and Return-terminated strings are selected by passing 0 to AlertWindow as the string flag. Pascal strings are selected by passing 1.

Elements do not need to be defined if they are not referenced in the alert.

Don't Use Separator Characters in Substitution Strings

Do not include a separator character in any substitution strings. Beginning with System Software 5.0, substitutions are performed before the alert string is scanned for separators. For example, if the separator character is a slash and a pathname containing several slashes is substituted for the string, the resulting alert will contain several more buttons than expected.

Examples

Following are some examples of alert strings that can be passed to AlertWindow in APW 65816 assembler syntax.

A simple alert string:

```
          dc    c'13/Text of Message/Button 1',il'0'---
                   \     \              \              \
   Size 50 high     Icon  Message        Button Title   Zero terminates alert.
    by 200 wide.
```

```
|--------------------------------------|
| #################################### |
| #                                  # |
| #   ______                         # |
| #  | Note |    Text of Message     # |
| #  | Icon |                        # |
| #  |______|                        # |
| #                                  # |
| #               _____________      # |
| #              /             \     # |
| #             |    Button 1   |    # |
| #              \_____________/     # |
| #                                  # |
| #################################### |
|--------------------------------------|
```

A more complex alert string:

```
    dc   c'51/This is the *0 of *3 alert *2*1 and standard'
    dc   c'text called "#4" /'
    dc   c'^#0,Really/*4/Yo!',il'0'
```

```
|-----------------------------------------------------------------------|
| ##################################################################### |
| #   ______                                                          # |
| #  | Stop |    This is the message text of an alert window and      # |
| #  | Icon |    standard text called "Try Again."                    # |
| #  |______|                                                         # |
| #                                                                   # |
| #          =============       _____________      _____________     # |
| #        //             \\    /             \    /             \    # |
| #       ||  OK, Really   ||  |    Door #2    |  |     Yo!      |   # |
| #        \\             //    \_____________/    \_____________/    # |
| #          =============                                            # |
| ##################################################################### |
|-----------------------------------------------------------------------|
```

Where substitution array =

```
                 dc   i4'sub0,sub1,sub2,sub3,sub4'
         sub0    dc   c'message text',il'0'
         sub1    dc   c'dow',il'0'
         sub2    dc   c'win',il'13'
         sub3    dc   c'an',il'0'
         sub4    dc   c'Door #2',il'0'
```

END OF FILE TN.IIGS.048

##
FILE: TN.IIGS.049
##

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#49: Rebooting (Really)

Revised by: Matt Deatherage January 1989
Written by: Matt Deatherage & Jim Merritt November 1988

This Technical Note discusses rebooting the Apple IIGS from software.
Changed since November 1988: Corrected two assembly-language instructions in the FROMNATV routine in the example code.

In days gone by, many Apple II applications had a Quit menu option. Unfortunately, a large number of these simply rebooted the machine. Today, this is far from desirable. Even with the advantages of GS/OS-reduced booting time (around 34 seconds with an Apple 3.5 Drive), waiting for the operating system to reload, as well as wiping out any ongoing tasks by desk accessories (such as an alarm clock) makes the standard ProDOS 8 or GS/OS QUIT call much more attractive.

However, there are still instances where an application may wish to require the user to reboot. A common example might be a game. The game might use GS/OS in a completely standard way, but if you QUIT from the program GS/OS booted into, you will be returned to the same program. Since most applications will boot into the Finder, this is not a widespread problem. However, the Finder must also provide the reboot option, and alternate program selector applications may wish to provide this functionality as well.

The Easy Way

GS/OS provides a mechanism for rebooting with the OSShutdown call. This call, documented in GS/OS Reference, Volume 1, will either reboot the system (after first shutting down all loaded and generated drivers and closing all open sessions) or will shut down everything and present a dialog box which states, "You may now power down your Apple IIGS safely." A Restart button is provided which allows the user to reboot without pressing Control-Open Apple-Reset .

Note: When using System Disk 4.0, if the Window Manager is active when you issue the OSShutdown call, there must be at least one open window; it need not be visible, but it must be open. This will be fixed in the next revision of GS/OS.

The OSShutdown call also provides a way to resize the internal RAM disk (named /RAM5 by default). Most programs have absolutely no need to use this mechanism, and should avoid it whenever possible. A notable exception would be a third-party RAM disk utility which uses a battery backup, which may need to make changes which require resizing the RAM disk. Of course, such a utility should ask the user to ensure that erasing the RAM disk content is acceptable. Resizing the RAM disk is only possible when using the OSShutdown call; any other method you may be using to accomplish this function from software will break in the future.

If you are using GS/OS, you should always use OSShutdown. You must not reboot the system in any other fashion. The OSShutdown mechanism provides a convenient and supported way to restart or shut down the system. Doing it another way can easily cause a loss of data.

The Hard Way

Programs not using GS/OS have a little more work to do. The supported non-GS/OS method of rebooting is similar to the method used on 8-bit machines: change the value of POWERUP ($00/03F4) and do a long jump to RESET ($FA62). However, there are a few catches:

1. The jump must be made in emulation mode.
2. Interrupts must be disabled.
3. The data bank register must be set to zero.
4. The direct page must be zero.
5. ROM firmware must be visible in the memory map.
6. Internal interrupt sources (such as the ones for AppleTalk) must be shut down.

Simply disabling interrupts without shutting down AppleTalk interrupt sources inside the system will cause the system to hang when the jump to RESET is made. Turning off these internal interrupt sources is accomplished by changing softswitch values at $C039 (SCCAREG), $C041 (INTEN), and $C047 (CLRVBLINT).

The following code example demonstrates the correct method:

```
POWRUP       equ    $0003F4    ;the power-up byte in bank zero
STATEREG     equ    $C068      ;ROM/RAM state register
CLRVBLINT    equ    $C047      ;clear VBL interrupt flags register
INTEN        equ    $C041      ;interrupt enable register
SCCAREG      equ    $C039      ;SCC register
RESET        equ    $00FA62    ;ROM reset entry point
;
FROMNATV     anop              ;enter here from native mode
             sei               ;disable interrupts
             pea    0
             pea    0          ;push four zero bytes on the stack
             plb               ;pull data bank register
             plb               ;(twice to balance the stack)
             pld               ;pull 16-bit data bank register
             sec
             xce               ;go into emulation mode
             longa    off
             longi    off
FROMEMUL     anop              ;enter here from emulation mode
             sei               ;disable interrupts for people entering here
             dec    POWRUP     ;invalidate the power up byte
             lda    #$0C       ;ROM parameters
             sta    STATEREG   ;swap in the ROM and everything else out
             stz    CLRVBLINT  ;clear VBL interrupts
```

```
             stz    INTEN       ;turn off internal interrupt sources
             lda    #$09
             sta    SCCAREG     ;shut down SCC interrupt sources
             lda    #$C0
             sta    SCCAREG
             jml    RESET       ;and off we go into the wild blue yonder
```

These methods of restarting the system are presented for those applications that absolutely must do so. Rebooting is not a suggested way of ending an application and the techniques described in this Note should be used with extreme caution.

Further Reference

- Apple IIGS Firmware Reference
- GS/OS Reference, Volume 1

END OF FILE TN.IIGS.049

##
FILE: TN.IIGS.050
##

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#50: Extended Serial Interface Error Handling

Written by: Dan Strnad January 1989

This Technical Note discusses error reporting by the Extended Serial Interface.

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. The following eight-bit APW code demonstrates this error checking using the SetDTR command. The SetDTR routine zeros the result bytes if no error has occurred.

```
           LONGA    OFF             ;PREPARE ASSEMBLER FOR EMULATION MODE
           LONGI    OFF
           65C02    ON
           KEEP     SETDTR2
           START
SLOT       EQU      $01
           SEC                      ;SET EMULATION MODE
           XCE
           JMP      BEGIN
CMDLST     DC       H'03'           ;PARAMETER COUNT
           DC       H'0B'           ;SETDTR COMMAND CODE
RESLT      DC       I'0'            ;RESULT CODE (OUTPUT)
DTRSTAT    DC       I'0'            ;BIT 7 IS STATE OF DTR (INPUT)
BEGIN      LDA      #SLOT           ;COMPUTE $CN VALUE TO BE USED
           ORA      #$C0
           STA      OFFSET+2        ;MODIFY INSTRUCTIONS LOADING OFFSETS
           STA      XOFFSET+2
           STA      ICALL+2         ;MODIFY INSTRUCTIONS CALLING FIRMWARE
           STA      XCALL+2
IOFFSET    LDA      $C00D           ;THIS INSTRUCTION MODIFIED AT RUNTIME
           STA      ICALL+1         ;MODIFY JSR TO INIT
XOFFSET    LDA      $C012           ;THIS INSTRUCTION MODIFIED AT RUNTIME
           STA      XCALL+1         ;MODIFY JSR TO EXTENDED SERIAL INTERFACE
ICALL      JSR      $C000           ;THIS INSTRUCTION MODIFIED AT RUNTIME
           LDA      #<CMDLST        ;LOW BYTE OF COMMAND LIST
           LDX      #>CMDLST        ;HIGH BYTE OF COMMAND LIST
           LDY      #0              ;24-BIT ADDRESS NOT USED BY 8-BIT PROGRAM
XCALL      JSR      $C000           ;THIS INSTRUCTION MODIFIED AT RUNTIME
           LDA      RESLT           ;DID AN ERROR OCCUR?
           BNE      ERROR           ;YES- HANDLE THE ERROR
           ...
ERROR
```

```
          ...
          END
```

END OF FILE TN.IIGS.050

###
FILE: TN.IIGS.051
###

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#51: Reserving Memory for the Toolbox

Written by: Eric Soldan January 1989

This Technical Note discusses handling nearly-out-of-memory situations when working with the IIGS tools.

Introduction

Running out of memory is a concern for most every application. Working with the Toolbox makes monitoring this situation a little more difficult since your application is not the only one allocating memory.

Just waiting for an out-of-memory error is not necessarily adequate memory management. If you execute a NewHandle call successfully, there could be any amount of memory left, from one byte to nearly all of it. If there is not much memory left, there might not be enough for the Toolbox. A better scheme of memory management would be to determine when the tools will need more memory than is available. You can treat this situation as "out-of-memory" as well.

The Memory Manager calls to determine how much memory is available are MaxBlock, FreeMem, and RealFreeMem. However, none of these calls can give you the complete picture. FreeMem does not count purgeable handles. RealFreeMem does count purgeable handles, but memory may be very fragmented. MaxBlock can only tell you if you have enough RAM in a single block to complete a new handle request, but it does not provide a good indication of when you do not have enough, since memory may be fragmented.

Another way of determining if you have enough memory is with the NewHandle call. If you know that you are going to do a sequence of operations that will not exceed N bytes of RAM, you can try a NewHandle call for that number of bytes. If it works, dispose the temporary handle and go for it. Of course, this may leave no memory available for the Toolbox, but you could fix this by trying a NewHandle of size N+ToolboxNeeds. The problem with this method is that NewHandle is not the fastest Memory Manager call, and executing it repeatedly can seriously degrade the performance of your application.

A Suggested Method

Another method of checking for a nearly-out-of-memory condition is to have your own purgeable handle just for this task. You can use the purgeable handle as a check for a nearly-out-of-memory situation. If the handle has not been purged, then you have plenty of memory for the Toolbox, and in the worst case, the Toolbox will purge your handle if it needs the RAM.

The less often your purgeable handle gets purged, the better performance you will probably get in nearly-out-of-memory situations. Therefore, the purge level of this handle should probably be 1. (It might be better to have your handle purged before several other purgeable handles which are of greater use and could belong to you or others in the system.) The check to see if a handle has been purged is very fast. If it has been purged, you will have to see if it can be reallocated which is not a fast process, so the fewer times the handle is purged, the faster the check will be and the better your performance. Unless you are in a nearly-out-of-memory situation, the handle will not be purged at all, and you will have virtually no overhead for this process.

This technique could be implemented as follows:

```
appStart              ;
;
;
;                     Somewhere at start, create a purgeable handle of size N,
;                     called "loMemHndl", purge level 1.
;
;
                      rts

******************
;
;                     Here's an example of checking for nearly-out-of-memory:
;
                      jsr    preCheckLoMem
                      bcc    goForIt
                      bcs    HandleError            ;Handle errors appropriately.
goForIt               (_ToolboxCall[s])             ;Make as many as needed.
;
;                     Here you can make your toolbox calls.  Since you prechecked
;                     for nearly-out-of-memory conditions, you should have no
memory
;                     errors at this point.
;
;                     You could also check after calls, as shown here:
;
                      (_ToolboxCall)
                      jsr    checkLoMem             ;Call this to see if low.
                      bcc    noError
                      bcs    HandleError            ;Take care of errors.

noError               jsr    lifeIsGood
                      .
                      .
                      .
                      rts

******************

******************

;
```

```
;                      Here are some sample routines to check for the nearly-out-
of-
;                      memory condition.
;

checkLoMem             bcs    retErr
preCheckLoMem          lda    [loMemHndl]
                       ldy    #2
                       ora    [loMemHndl],y
                       beq    gotPurged
                       lda    #0
                       clc
                       rts
gotPurged              (Try reallocating it into loMemHndl, purge level 1.)
                       (If you can't, you will get a $0201 error.  You may wish to
                       return the $201 error, or you may wish to change it into
                       your own error code.)
;
retErr                 rts                        ;This is a single exit point
                                                  ;whether errors were present
                                                  ;or not.
```

You can determine the size of this purgeable handle, but, like determining what size stack is adequate for an application, there is no single "right" answer. There are different considerations for size of the purgeable handle for each application, and these may change during the development process. Use your best judgement.

Conclusion

This Note is not meant to suggest that nearly-out-of-memory situations require detection in this way, and there are many applications which allocate enough RAM when they start (i.e., many paint programs) that if they get the requested memory, they will not encounter out-of-memory situations during that session. There may also be other ways for a particular application to detect and handle nearly-out-of-memory situations, but this Note addresses this situation in a general way and offers only one solution for your consideration.

Further Reference:

o   Apple IIGS Toolbox Reference, Volumes 1 & 2

END OF FILE TN.IIGS.051

```
#####################################################################
### FILE: TN.IIGS.052
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#52: Loading and Special Memory

Revised by: Eric Soldan — July 1989
Written by: Eric Soldan — January 1989

This Technical Note discusses strategies for preventing applications from loading into special memory.
Changes since January 1989: Modified code sample so the Loader no longer disposes of some segments before a Restart call is complete.

The System Loader loads your application starting at the lowest memory location possible. If you allow your program to load into special memory, the Loader first tries bank $01. If your program cannot load into special memory, it starts at bank $02. Either way, the Loader progresses to higher banks, and eventually, it may even try loading into bank $E1, which contains the super hi-res screen.

The problem with allowing your application to load into special memory is that the super hi-res screen is part of special memory. If you have a desktop application, part of your application may load into the super hi-res screen, and when you try to start QuickDraw II, it fails because the screen memory is already allocated.

When QuickDraw II fails because your program loaded into the SHR screen, it seems reasonable to assume that the Loader put your program there because it needed the RAM which special memory provides. This logic seems to make sense, but it is not completely reliable. The Loader tries to put your program into special memory before it tries purging dormant applications. This means that the more programs that run from the Finder that set the GS/OS or ProDOS 16 "restartable from memory" bit, the more likely it is that the next application launched that can load into special memory will load into the super hi-res screen.

For this reason, it is important not to let your application load into special memory, or at least not load into the super hi-res screen. If your application is not allowed to load into special memory, then the Loader will purge other dormant applications to make space for yours. One way to accomplish this is when linking your application. You can set the "no special memory" bit in the OMF KIND field of applications using OMF 2.0 or later, but this also prohibits your application from using bank $01.

Another way to avoid loading into the super hi-res screen is to have your initial segment allocate the super hi-res screen. You can accomplish this by starting QuickDraw II in your initial segment, then the rest of your program cannot load into the already-allocated super hi-res screen. This strategy could fail if the initial segment loaded into the super hi-res screen, but this is very unlikely and can be prevented by flagging the initial segment to only load into non-special memory. You can do this by setting the "no special memory" bit in the KIND field only for the initial segment.

Here's an example of such an initial segment in MPW IIGS format:

```
*************************************************************************
*
* You may wish to do this stuff in the initial segment of your
* application.  The initial segment should be set so that it does not
* load into special memory, or else it is possible that it would load
* into the super hi-res screen.  If this occurred, then QuickDraw II would
* not be able to be started.
*
* Once QuickDraw II is started, the super hi-res screen is taken,
* therefore the rest of the application can not load into it.  Therefore,
* special memory is generally an acceptable place for the rest of the
* application to load, since the special memory needed for the screen
* is already taken.
*
* If the performance of your application would be adversely affected
* by memory fragmentation, then you should also consider purging
* other dormant applications and dormant tools, and then compacting
* memory.  This will prevent fragmentation as much as possible
* while your application is loading.  It also has the cost of longer
* startup time since some tools may have to be reloaded.  This is the
* only way to be sure that tools that you don't want are removed
* from memory before the rest of your application tries to load
* around them.
*
* The Finder is a dormant application when your application is
* launched.  This will cause the Finder to be thrown out of memory,
* and it will have to be reloaded when your application is quit.
*
*************************************************************************

            case on

            include 'e16.memory'
            include 'm16.memory'
            include 'm16.quickdraw'


screenMode         equ    $80
AppMaxWidth        equ    160            ;Double this is your application
                                         ;will print in BetterText mode.
******************


initialScreen    PROC

myID               equ    1              ;long
zpagehndl          equ    myID+4         ;long

stkAfterLocals     equ    zpagehndl+4
```

```
directReg           equ    stkAfterLocals
retAddr             equ    directReg+2
passedParms         equ    retAddr+3

                    phd                   ;Set up stack frame.
                    tsc
                    sec
                    sbc    #stkAfterLocals-1
                    tcs
                    tcd
                    pha
                    _MMStartUp
                    pla
                    sta    myID           ;Get the userID

                    pha
                    _HLockAll             ;Lock down the rest of ourselves, in
                                          ;case we are being restarted.  The
                                          ;loader does not prelock down stuff,
                                          ;so we would be disposing of the rest
                                          ;of ourselves.

                    pea    $1000
                    _PurgeAll             ;Purge other dormant applications.
                                          ;This is optional.
                    pea    $4000
                    _PurgeAll             ;Purge dormant tools.
                                          ;This is optional.

                    _CompactMem           ;Clean up memory.  This is advised.

                    pha                   ;Make direct space for QuickDraw.
                    pha
                    pea    $300>>16       ;Hi-byte of $300 address.
                    pea    $300
                    pei    myID
                    pea    attrLocked+attrFixed+attrPage+attrBank
                    lda    #0
                    pha
                    pha
                    _NewHandle
                    plx
                    stx    zpagehndl
                    plx
                    stx    zpagehndl+2
                    bcc    @a
                    ERRORDEATH 'Out of bank 0 memory'

@a                  lda    zpagehndl
                    sta    >qdstarthndl  ;Used for disposing handle at shutdown.
                    txa
                    sta    >qdstarthndl+2
                    lda    [zpagehndl]   ;Start up QuickDraw.  This protects
                    pha                  ;screen ram from the rest of the
                    pea    screenMode    ;application from loading into it.
                    pea    AppMaxWidth
                    pei    myID
                    _QDStartUp
```

```
                   bcc    @b
                   ERRORDEATH 'Can''t start up QuickDraw'
@b                                    ;Do title screen here.
                   tsc
                   clc
                   adc    #stkAfterLocals-1
                   tcs
                   pld
                   rtl

qdstarthndl        dc.l   0

                 ENDP
                 END
```

Further Reference:

- GS/OS Reference, Volume 1
- MPW IIGS Tools Reference
- APW Assembler Reference

END OF FILE TN.IIGS.052

```
#####################################################################
### FILE: TN.IIGS.053
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#53: Desk Accessories and Tools

Written by: Matt Deatherage & Jim Mensch March 1989

This Technical Note describes new guidelines for developers to help applications and desk accessories live together in the same system at the same time.

Introduction

Desk accessories vary widely in complexity. Classic Desk Accessories (CDAs) range from simple status-reporting programs to complete system-level debugging utilities, and similarly, New Desk Accessories (NDAs) range from static windows with pictures to nearly full-fledged applications.

This Note presents some new guidelines aimed at helping developers of both applications and desk accessories to get their products to work together now and in the future.

Tool Sets

The greatest current conflict between applications and desk accessories, especially NDAs, is the use of system tool sets. The Apple IIGS Toolbox Reference, Volume 1, defines which tools are available for use by NDAs. The Desk Manager requires starting the following tool sets before calling FixAppleMenu (which installs the names of the NDAs in the Apple menu):

- Tool Locator (#1)
- Memory Manager (#2)
- Miscellaneous Tools (#3)
- QuickDraw II (#4)
- Event Manager (#5)
- Window Manager (#14)
- Menu Manager (#15)
- Control Manager (#16)
- LineEdit (#20)
- Dialog Manager (#21)
- Scrap Manager (#22)

Since the Desk Manager requires starting these tools before calling FixAppleMenu, NDAs may assume that these tools are all present and running, so they do not need to check for their presence.

In addition to these requirements by the Desk Manager, Apple II Developer Technical Support strongly recommends that all applications start the following tools:

    QuickDraw Auxiliary (#16)
    Font Manager (#27)

These two additional tools are so widely used by desk accessories that they should be present. An NDA may not assume their presence, but any application that calls FixAppleMenu should also start these two tools.

The Golden NDA Guideline

Developers who wish to maintain maximum compatibility between their NDAs and applications, both now and in the future, should consider every environment change they make with the following Golden NDA Guideline firmly in mind:

> "I, an NDA, pledge not to alter the environment of the application under which I run, and I will behave in such a way that the application runs the same whether I am present or not."

Of course, this guideline does not include such necessary tasks as the normal (and reasonable) allocation of memory. An application must be prepared to handle a memory allocation call by a desk accessory, operating system, or even a tool at unexpected times. The guideline does, however, mean that your desk accessory cannot change the operating environment, including such things as the presence of tools and operating system parameters. The following sections detail some of the most important ways of following the Golden NDA Guideline.

Desk Accessory Guidelines

Extra Tools

o   If an NDA wishes to use a tool which is not available in the standard list (e.g., Standard File), it should check to see if the tool is already running. If it is not running, the NDA must use LoadOneTool to load it, then it must start the tool before using it. When finished with the tool, the NDA must shut it down and unload it with UnloadOneTool.

o   If an NDA wishes to use a tool which is not available in the standard list and the NDA wants to use it in a non-modal fashion, then it has even more work to do. In addition to the conditions set for a tool which is not available in the standard list, if your NDA uses a tool in a non-modal fashion, you must shut it down and unload it when your window is deactivated.

     The Golden NDA Guideline shows why this is true. If your NDA started a tool which the application was going to use but had not yet started (i.e., the Font Manager), and your NDA does not shut it down when the application takes control of the system, the application will get error $1B01 (Font Manager Already Started) when it makes the FMStartUp call, and this error can cause the application to fail.

     Therefore, when your window is deactivated, if you shut down all the tools you have started, the application will be free to start those tools which it requires. When your window is reactivated, you must check the status of each tool in question then reload and restart those which are not present before reusing them.

In this case, the Golden NDA Guideline means that an application must not be forced to check the status of a tool which it has not started. Applications are not required to do so, and most of them do not.

- o Sound tools provide the one exception to the rule of freely using a tool which is already started. Refer to the section on System Parameters for more details on using sound tools.

- o An NDA must not shut down tools which it has not started. If you start a tool then shut it down when finished with it, you must be sure not to accidently shut it down the next time your window is deactivated.

- o A CDA is nearly always modal, but it has the capability to install HeartBeat Tasks and other ways of being called after returning to an application (which could be running under ProDOS 8). If a CDA installs a method of performing tasks after the user has returned to the application, it must be careful not to use any tools which are not started, since the list of available tools for NDAs does not hold true for CDAs.

System Parameters

- o A desk accessory (CDA or NDA) must not change a system resource or parameter which cannot be restored to its original condition. A trivial, but illustrative, example of this is the number of times a pull-down menu item blinks when you select it. This number (three by default) may be changed with the SetMItemBlink call, but there is not corresponding GetMItemBlink call, so you cannot retrieve the current value. Therefore, a desk accessory must not change this parameter, and the same rule applies to any other system parameter for which you cannot determine a current value.

- o This idea extends to calling tool startup functions. Even if a tool's startup function does not return an error if the tool is already active, it may reset certain parameters upon which the application depends. An example of this is TLStartUp for the Tool Locator. A seemingly innocuous call, TLStartUp actually disconnects any user tool sets present, which, in this case, would most likely have been installed by the current application.

  A desk accessory should not call any tool's startup function if the tool is already active. The one exception to this rule is the Memory Manager's MMStartUp call, which a desk accessory may make to obtain its User ID. MMStartUp may be considered equivalent to a GetMyID call.

- o A desk accessory cannot use any of the sound tools if they are already started. This is contrary to all other tool sets, but it is required in this case since there is no memory management of the sound RAM. If the Sound Tools (#8) are started, the application has exclusive control of the 64K DOC RAM used to play sounds. Anything your desk accessory might put there is likely to overwrite information the application needs.

  Saving and restoring DOC RAM around desk accessory usage is not sufficient. Many of the sound functions are interrupt-driven, altering the contents of DOC RAM only at sound interrupts, so your desk accessory might attempt to replace parts of DOC RAM which are being played. Since there is no memory management of DOC RAM, desk accessories must avoid the sound functions of the IIGS if the application is already using them.

o A desk accessory must not install user tool sets, because there is no arbitration of user tool set numbers.

Application Guidelines

To coexist peacefully with desk accessories, particularly NDAs, applications generally need to follow the guidelines listed in the Desk Manager chapter of the Apple IIGS Toolbox Reference, Volume 1. However, those applications which wish to ensure maximum compatibility now and in the future will also want to adhere to the following:

o Don't just start the Scrap Manager--use it. Many desk accessories are capable of cutting and pasting information between themselves and your application, but they cannot do so if you do not use the Scrap Manager. If you handle the Edit menu functions privately, without placing the information on your internal clipboard in the public scrap, a desk accessory will not be able to access it. This inability to share information frustrates both the users and the developers who write desk accessories.

o Start tools at the beginning of your application and leave them started. Every time you call SystemTask or TaskMaster, a desk accessory might take control of the system, and if your application has shut down a tool that a desk accessory found running and is using, it might not be able to complete an operation. For example, a desk accessory might be using the Print Manager, having found it started by your application. If your application takes control of the system and shuts down the Print Manager while the desk accessory is printing a document, the desk accessory will not be able to finish when it regains control.

  For maximum compatibility, do not shut down any tools which were ever active when you called SystemTask or TaskMaster. You can start more tools, but do not shut down those which are already active. If you intend to start a tool and not keep it started, use it then shut it down immediately, being sure not to call SystemTask or TaskMaster during that time.

o An application with some memory to spare can save NDAs time by providing them the additional tools which they are most likely to use. If a desk accessory wishes to use the List Manager and your application starts it, the desk accessory will naturally run faster since it will succeed on the ListStatus call every time and can avoid loading and starting the tool on every activation.

  The most common tools which desk accessories require besides those available in the standard Desk Manager set are QuickDraw Auxiliary (#16), the Print Manager (#19), Standard File (#24), the Font Manager (#27), and the List Manager (#28). QuickDraw Auxiliary and the Font Manager are especially important--not only do they work well together, but they are also widely used. In addition, FMStartUp can take a long time, and waiting for it every time you activate an NDA window gets really frustrating. Many desk accessories also use the Print Manager, the List Manager, and Standard File, and if they are always available, desk accessories will work more smoothly with your application.

Further Reference:

- Apple IIGS Toolbox Reference, Volume 1
- Programmer's Introduction to the Apple IIGS

END OF FILE TN.IIGS.053

###
FILE: TN.IIGS.054
###

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#54: MIDI Drivers

Written by: Jim Mensch May 1989

This Technical Note describes how to write a driver for use with the Apple IIGS MIDI tools.

Apple ships two drivers with the MIDI tool set, APPLE.MIDI and CARD6850.MIDI, respectively. These drivers are adequate for almost all MIDI hardware currently on the market for the Apple IIGS; however, if your hardware is not compatible with either of these drivers, you will have to write your own. This Note includes all the information you need to create a MIDI driver.

Purpose of the Driver and Description of Hardware Requirements

The Apple MIDI tools communicate to the MIDI world via a simple driver. The driver's function is managing the transmission and reception of single bytes of MIDI data between the tools and the particular MIDI hardware involved. The MIDI tools operate on the assumption that the hardware has a method of interrupting the system when a character has been received and when a character can be transmitted. Since there is quite a bit of overhead in processing MIDI data, and since MIDI data can comes across a standard MIDI bus at a rate of over 3000 bytes per second, it is suggested that you provide a means for your device to buffer a few characters to reduce system overhead caused by interrupts if you are designing hardware to be used with the MIDI tools.

Format of the Driver File

The driver file is a standard OMF load file, which can be created with any of the popular Apple IIGS assemblers. The file must start with a dispatch table that contains the addresses of the standard driver routines. All driver routines must be in the same segment as the dispatch table. The dispatch table should have 13 four-byte entries, each of which contains the address of the appropriate routine minus one. Table 1 contains addresses of routines in the MIDI driver to perform specific functions.

Call	Function
Init	Called to initialize the port and prime the driver
ShutDown	Called to close the port and clean up

	after driver
Reset	Called at reset time by the MIDI tools
IntHandler	Called when your interrupt occurs
PollRecv	Poll input the port for data
RecvIntOn	Turns on receiver interrupts
RecvIntOff	Turns off receiver interrupts
PollXmit	Polls the transmitter to see if another character can be sent
XmitIntOn	Enables transmitter interrupts
XmitIntOff	Disables transmitter interrupts
NotImp	Currently unused
NotImp	Currently unused
NotImp	Currently unused

Table 1-MIDI Driver Function Routines

Routine Calling Conventions

All driver routines are called with full 16-bit mode enabled and should exit the same way. On entry to each routine, the accumulator contains the direct page pointer that the driver should use if it wants to use the MIDI tools' direct page. It is the driver's responsibility to set the direct page register and restore it on exit. All other parameters are passed on the stack and should be removed from the stack before the routine exits. The MIDI tools set aside 128 bytes of space on the passed direct page for use by the driver. They are bytes $80–$FF.

If you want to report an error inside of any routine (except IntHandler), set the carry flag on exit and load the accumulator with the error code. Use predefined error codes whenever possible. If you need to report a device specific error, use errors in the range $C0–$FF. The MIDI tools will set the high byte of the error code properly for you, so you do not need to do it yourself. Table 2 lists all of the potential predefined error codes.

Error Code	Error Definition
miToolsErr ($2004)	The required tools were not started
miNoBufErr ($2007)	No buffer is currently allocated
miDevNotAvail ($2080)	Requested device is not available
miDevSlotBusy ($2081)	Requested slot is already in use
miDevBusy ($2082)	Requested device is already in use
miDevOverrun ($2083)	Device overrun by incoming MIDI data
miDevNoConnect ($2084)	No connection to MIDI
miDevReadErr ($2085)	Framing error in received MIDI data
miDevVersion ($2086)	ROM version is incompatible with driver
miDevIntHndlr ($2087)	Conflicting interrupt handler installed

Table 2-Predefined Error Codes

The Driver Routines

Init

This routine is called by the MIDI tools when it wants to initialize your port and tell the driver to prepare itself for the rest of the calls. Figure 1 shows how the stack looks on entry to this call.

```
        | previous contents |
        |-------------------|
        |    long space     |  LONG - Space for result
        |                   |
        |-------------------|
        |    NewIntAddr     |  LONG - Pointer to MIDI tools int vector
        |                   |
        |-------------------|
        |      SlotNum      |  WORD - Number of slot/port to use
        |-------------------|
        |      SlotFlag     |  WORD - 0=Internal port 1=card in slot
        |-------------------|
        |       UserID      |  WORD - User ID of current application
        |-------------------|
        |   RTL   |   RTL   |
        |-------------------|
        |   RTL   |
        |---------|  <------ Stack Pointer
```

Figure 1–The Stack on Entry to Init

The Init routine should first test to see if the port specified by SlotFlag and SlotNum is available for use. SlotNum is the number of the slot or the port that the user has requested for use, and SlotFlag indicates whether it is a built-in port or a card in a slot. After determining that the requested device is available, you should initialize the device, allocate any memory that your driver may require (beyond what is available in the direct page), and set the proper system interrupt vector to the address passed in NewIntAddr. Before setting the vector, be sure to save the old value, as the MIDI tools expect the result from this routine to be the old address stored in the vector. On exit, the stack should contain the return address and the old vector address.

ShutDown

This routine is called when the MIDI tools want your driver to release the MIDI device and prepare to be unloaded. Figure 2 shows how the stack looks on entry to this call.

```
        | previous contents |
        |-------------------|
        |   OldIntVector    |  LONG - Old interrupt vector pointer
        |                   |
        |-------------------|
        |   RTL   |   RTL   |
        |-------------------|
        |   RTL   |
        |---------|  <------ Stack Pointer
```

Figure 2–The Stack on Entry to ShutDown

Your routine should change the interrupt vector that you used to OldIntVector. It should then deallocate all the memory that it allocated, disable all interrupts on the device, and if needed, tell the system that you are no longer using the port in question.

Reset

This routine is called when the system has been reset by the user. Figure 3 shows how the stack looks on entry to this call.

```
        | previous contents |
        |-------------------|
        |   OldIntVector    |  LONG - Old interrupt vector pointer
        |                   |
        |-------------------|
        |   RTL   |   RTL   |
        |-------------------|
        |   RTL   |
        |---------|  <------ Stack Pointer
```

Figure 3–The Stack on Entry to Reset

All you should do at this point is attempt to deallocate any memory you were using and disable interrupts on the device you were using.

Note: Do not set the interrupt vector to OldIntVector, instead remove the value from the stack and dispose of it.

IntHandler

The IntHandler routine is called by the MIDI tools when an interrupt occurs for the vector that you are using. The MIDI driver performs some setup then calls your routine. This routine does not have any parameters on the stack.

Once called, your IntHandler routine should test the port to see if an interrupt has occurred on your device. If your device did not cause the interrupt, you should set the carry and exit as quickly as possible, reducing the system interrupt overhead.

If your device caused the interrupt, you should test the receiver to see if any bytes of data are waiting to be read. If there is data waiting, you should load that data into the accumulator and perform a JSL to the following code:

```
    InBufGlue    PEA $0400
                 PHD
                 RTL
```

This code calls the MIDI tools and tell them to accept the character in the accumulator into its input buffer. After accepting the data, control is passed back to the instruction following your JSL. If you received a byte of data and an error occurred during reception, you should load the number of the error code into the y register and perform a JSL to the following code:

```
    InErrGlue    PEA $0500
                 PHD
                 RTL
```

Again, you will regain control right after the JSL. Once in your interrupt routine, you may perform the calls above for as much data as you like. For example, if your device has a three-byte buffer, you could call InBufGlue once for each waiting character, thus reducing your interrupt overhead and possibly preventing unneeded interrupts.

If the transmitter on your device is ready to send data, you should perform a JSL to the following code:

```
    OutBufGlue    PEA $8400
                  PHD
                  RTL
```

This routine will return with the carry set if no data is waiting to be transmitted or the carry clear if data is available. If data is waiting, the next character to send will be in the accumulator, and you should simply send it at that time. If no more data is available, you should disable transmitter interrupts and exit. The MIDI tools will re-enable transmitter interrupts the next time it has data to send.

PollRecv

The PollRecv (Poll Receive) routine is called by the MIDI tools every now and then to see if any data might be waiting to be read. There are no parameters on the stack for this call. Your driver should test to see if any data is available and transmit it all to the MIDI tools via the InBufGlue described in the IntHandler description.

PollXmit

The PollXmit (Poll Transmit) routine is called by the MIDI tools when any data is added to the MIDI output buffer. There are no parameters on the stack for this routine. Your driver should enable transmitter interrupts, test to see if it can send any data immediately, and if it can, call OutBufGlue as described int the IntHandler description to get data to send.

XmitIntOn and RecvIntOn

These routines are called when the MIDI tools want to explicitly enable transmitter or receiver interrupts. They have no parameters on the stack and should, when called, enable transmitter interrupts for XmitIntOn and receiver interrupts for RecvIntOn.

XmitIntOff and RecvIntOff

These routine are called when the MIDI tools want to explicitly disable transmitter or receiver interrupts. They have no parameters on the stack and should, when called, disable transmitter interrupts for XmitIntOff and receiver interrupts for RecvIntOff.

NotImp

These routines are not yet implemented, but your driver should be ready to handle a call to them. When called, they should clear the accumulator, clear the carry and perform an RTL back to the MIDI tools.

A MIDI Driver Skeleton

You can use the following sample code as a basis for a MIDI driver. It is not a complete driver in itself, and you will need to add code where comments with asterisks (***) appear for it to be functional. This example is in MPW IIGS assembler format.

```
*******************************************************************************
*
* MIDI.DRVR.Aii
*
* (C)  Copyright Apple Computer, Inc. 1988
* All rights reserved.
*
* by Don Marsh & Jim Mensch
* 10/26/88
*
* This is a shell that can be used to create custom MIDI drivers for use with
* the Apple MIDI tool set. This shell is not functional, but can be used as a
* starting point for creating your own custom MIDI drivers.
*
* Files:    System Macros and equates
*
*
*
* Modification History:
*
* Version 1.0   Mensch
*
*       10/26/88
*
*       Create first    draft
*
*******************************************************************************
*
    Include 'E16.MIDI'
    Include 'M16.MiscTool'
    Include 'E16.MiscTool'
    Include 'M16.util'

;
; Direct page usage    Note:
; MIDI drivers may use the upper half ($80-$FF) of the MIDI direct page. When
; a MIDI driver routine is called the Accumulator will contain the direct page
; pointer for the MIDI tool set. If your driver requires more storage than
; 128 bytes, it will have to allocate them itself using the memory manager.

theuserID     equ $80                ; location to store the passed user ID
PortInUse     equ theuserID+2        ; storage for the port number in use
deref         equ PortInUse+2
Temp          equ Deref+4
              EJECT

*******************************************************************************
```

```
*
*
DispatchTable    RECORD
*
* Description:    Every MIDI Driver must start with a driver dispatch table
*       that contains the entry point minus 1 of each of the
*       required entry points.
*
*
* Inputs:     None
*
* Outputs:    None
*
* External Refs:
        Import DRVRInit
        Import DRVRShutDown
        Import DRVRReset
        Import DRVRIntHandler
        Import DRVRPollRecv
        Import DRVRRecvIntOn
        Import DRVRRecvIntOff
        Import DRVRPollXmit
        Import DRVRXmitIntOn
        Import DRVRXmitIntOff
        Import DRVRNotImplemented
*
* Entry Points:    None
*
******************************************************************************
*

              DC.L DRVRInit
              DC.L DRVRShutDown
              DC.L DRVRReset
              DC.L DRVRIntHandler
              DC.L DRVRPollRecv
              DC.L DRVRRecvIntOn
              DC.L DRVRRecvIntOff
              DC.L DRVRPollXmit
              DC.L DRVRXmitIntOn
              DC.L DRVRXmitIntOff
              DC.L DRVRNotImplemented
              DC.L DRVRNotImplemented
              DC.L DRVRNotImplemented


; a few of the routines will need a temporary storage location that can be
; used even after the direct page is set back to what it was, This is a good
; place to put it!

ErrorCode     ds.W 1                     ; temporary holder of an error code
              EndR

              EJECT

******************************************************************************
*
*
```

```
DRVRInit    PROC
*
* Description:    This is called by the MIDI Tools when it needs to Init
*                 your MIDI Driver. This is usually in response to a MIDIxxx
*                 call made by the application.
*                 When this routine is called, you should allocate any buffer
*                 space that you will need beyond the direct page, you should
*                 enable the interrupts on your MIDI Device, and then set the
*                 appropriate system interrupt vector and return the old vector
*                 value. If the init works fine, clear the carry and return.
*                 If an error occurs return the appropriate error code
*                 in the Accumulator, and set the carry.
*
*
* Inputs:     UserID:Word         ID of application, for mem allocation
*             SlotFlag:Word       0 for internal port/ 1 for slot
*             SlotNum:Word        number of slot/port to use
*             NewIntVector:Long   address to give system as its new
*                                 interrupt vector. This routine is in the
*                                 MIDI tool set, and it performs needed
*                                 setup before it calls your interrupt
*                                 routine
*
* Outputs:    OldIntVector:Long   Address interrupt vector used to have
*
* External Refs:    None
*
* Entry Points:     None
*
******************************************************************************
*
; Offsets for parameters on the stack

ProcStatus     equ 1
OldDPage       equ ProcStatus+1
ReturnAddress  equ OldDPage+2
UserID         equ ReturnAddress+3
SlotFlag       equ UserID+2
SlotNum        equ SlotFlag+2
NewIntVector   equ SlotNum+2
OldIntVector   equ NewIntVector+4
ParmBytes      equ 10
ParmEnd        equ ReturnAddress+ParmBytes

; first disable interrupts since we are going to be setting up interrupt
; vectors and enabling interrupt generating hardware. We wouldn't want an
; interrupt to go off before we were ready to handle it! Then set us up to
; use the MIDI direct page.

               php                   ; save the old proc status
               phd                   ; save the old direct page
               tcd                   ; Set Direct page to the one passed
               SEI                   ; and disable interrupts

; now get the user ID and save it, and allocate any buffers that we may need
; Since most drivers will never need more than 128 bytes of storage we will
; not allocate any storage space
```

```
               lda UserID,s         ; first save the user ID for later
               sta theUserID        ; in our section of the MIDI DPage

; *** Insert any memory allocation needed here ***

; Next, you should check the slot flag and number to see if they are
; compatible with this driver. If they are, you should continue and
; initialize the proper port. If they are not proper, you should exit with
; an error.  For this example, I will be testing the SlotFlag, to see if
; it is set to external.

               lda SlotFlag,s       ; first test the slot flag to be sure
               bne FlagOK           ; its non-zero.

               ldy #miDevNotAvail   ; if its zero, signal not available
               bra InitError        ; and exit via error routine

FlagOK         lda SlotNum,s        ; Now save the slot number in
               sta PortInUse        ; our data area

; *** At this point you should test the firmware in the desired slot to be
; sure that the card you want is properly installed, if it is not then you
; should pass back the appropriate error ***

; Now that you know that you have the proper slot information and you have
; tested to be sure that you have the hardware needed for the driver it is
; time for you to initialize the interface and to enable its interrupts.

; *** Install code to initialize your hardware/interrupts here ***

; Now that the Port has been properly initialized, you must set up the proper
; system interrupt vector. Since we required an external card above it would
; make sense that you need to use the "Other unspecified interrupt handler"
; vector (Number $0017). But first, remember to get the original vector
; pointer because we must return it to the MIDI tools.

               PushLong #0          ; space for result
               PushWord #otherIntHnd  ; vector to retrieve
               _GetVector           ; and get the vector in question
               PullLong Temp        ; place in storage for a sec

               lda Temp             ; now place it on the stack
               sta OldIntVector     ; as the result of this function
               lda Temp+2
               sta OldIntVector+2

               lda NewIntVector     ; now move the MIDI Interrupt routine
               sta Temp             ; pointer into temporary storage
               lda NewIntVector+2
               sta Temp+2

               PushWord #otherIntHnd ; now set the vector to point to
               PushLong Temp        ; the MIDI drivers interrupt routine
               _SetVector

; The driver is now all set up, pull off the passed parms and we are done!
Done           ldy #0               ; set the error code to 0. No error
;
```

```
; This is the alternate label for the Done routine that should be called when
; an error has occurred.
InitError
               lda ReturnAddress,s  ; Move the return address below the
               sta ParmEnd,s        ; parameters
               lda ReturnAddress+1,s
               sta ParmEnd+1,s

               pld                  ; get the direct page back
               plp                  ; get the processor status back

               tsc                  ; now adjust the stack pointer
               sec                  ; so that the parameters are gone
               sbc #ParmBytes
               tcs                  ; now the return address is on Top

               tya                  ; put any error into <A>
               cmp #1               ; set the carry if non-zero
               RTL                  ; and return

               EndP

               EJECT
******************************************************************************
*
*
DRVRShutDown    PROC
*
* Description:    This routine will be called whenever the MIDI Tools want
*                 to cause your driver to let go of the port it was using.
*
*
* Inputs:    OldIntVector:Long    Address to place back into the system
*                                 interrupt vector you were using
*
* Outputs:    Carry clear if successful
*             Carry set if not, error in <A>
*
* External Refs:
    Import DrvrRecvIntOff
    Import DRVRXMitIntOff
*
* Entry Points:
*
******************************************************************************
*
        With DispatchTable

ProcStatus     equ 1
OldDPage       equ ProcStatus+1
ReturnAddress  equ OldDPage+2
OldIntVector   equ ReturnAddress+3
ParmBytes      equ 4
ParmEnd        equ ReturnAddress+ParmBytes

; first disable interrupts since we are going to be setting up interrupt
; vectors  We wouldn't want an interrupt to go off before we were ready
; to handle it!  Then set us up to use the MIDI direct page.
```

```
               php                    ; save the old proc status
               phd                    ; save the old direct page
               tcd                    ; Set Direct page to the one passed
               SEI                    ; and disable interrupts

               lda #0                 ; zero out the temp error code
               sta >ErrorCode
; Now First, re-install the old interrupt vector

               lda OldIntVector       ; get the old vector off the stack
               sta Temp               ; and save it in globals for a sec
               lda OldIntVector+2
               sta Temp+2

               PushWord #otherIntHnd ; now set the vector to point to
               PushLong Temp          ; its original routine.
               _SetVector

; Next, turn off the interface hardware, and tell it to stop generating
; interrupts. We can share some code here and call our DRVRRecvIntOff and
; DRVRXmitIntOff routines. Always remember load the direct page into the
; accumulator.

               tdc                    ; get direct page into <A>
               jsl DRVRXmitIntOff     ; and turn off transmitter interrupts

               tdc
               jsl DRVRRecvIntOff     ; and now receiver interrupts.

; *** Usually turning off interrupts will be all that you would need to do at
; this point, however, if your interface card requires extra shutdown code
; this is where you would place it ***

; *** If you allocated any memory in the DRVRInit call, this is the place to
; get rid of it.

; If an error were to occur in this routine, you should simply store the error
; number in our temporary error code variable like this
;
;               lda #ErrorNumber
;               Sta >ErrorCode


Done
; Now that we are done shutting down the driver, pull off the passed data
; and end.
               pld                    ; first retrieve the old dpage
               plp                    ; and processor status

               Longa Off              ; next move the return address
               SEP #$20               ; we need a short acc for this trick

               pla                    ; pull the 3 byte return address
               ply                    ; into <A> and <Y>

               plx                    ; now remove the remaining bytes
               plx                    ; of passed parameters
```

```
               phy                    ; and restore the return address
               pha

               Longa On
               REP #$30               ; and turn back on full 16-bit mode

               lda >ErrorCode         ; retrieve the error code
               cmp #1                 ; and set the carry if non-zero
               RTL
               EndP

               EJECT

******************************************************************************
*
*
DRVRReset    PROC
*
* Description:    This routine will be called whenever MIDIReset is called.
*                 and that should only happen when an actual reset occurred.
*                 It should in most cases perform the exact same functions
*                 as MIDI Shutdown.
*
*
* Inputs:    OldIntVector:Long    Original contents of interrupt vector
*
* Outputs:    None
*
* External Refs:
*
* Entry Points:
*
******************************************************************************
*

               jmp DRVRShutDown

               EndP

               EJECT
******************************************************************************
*
*
DRVRIntHandler    PROC
*
* Description:    This routine is the very core of the MIDI driver. It takes
*                 care of passing data back and forth between the MIDI tools
*                 and your hardware. It will be called for both input and
*                 output.
*
*
* Inputs:     None
*
* Outputs:    Carry set if interrupt not serviced
*
* External Refs:
        Import DRVRXmitIntOff
```

```
*
* Entry Points:
        Export InBufGlue
        Export InErrGlue
        Export OutBufGlue
*
******************************************************************************
*

               phd                    ; first, save the current dpage
               tcd                    ; and use the MIDI DPage

; The first thing the interrupt routine should do is to test to see if the
; interrupt was actually generated by our port. If it was then we should
; handle it, but if not, we should simply exit this routine with the carry
; set as fast as we can, so that the next interrupt handler will get it
; in a timely manner.

; *** Insert code here to test to see if the original interrupt was yours ***

               beq ServicePort        ; if it was our, handle it

; If the interrupt was not ours, set the carry and leave
               pld                    ; restore the direct page
               sec
               rtl

ServicePort                           ; the interrupt was ours, continue

; This routine should test the interrupt again, too see if the port is ready
; to transmit or receive, If it is ready to transmit or receive, it should
; then call the ServiceRecv, or ServiceXMit routines

; *** Insert code here to test for receive

               bne ServiceRecv        ; if chars waiting try receive it

; If no more characters are waiting, see if we are ready to transmit any
; characters.

               bne ServiceXMit        ; if can send a character do it

; If both the above tests fail, then exit the interrupt handler for now
               pld                    ; restore the direct page
               clc                    ; clear the carry to indicate serviced
               RTL                    ; and return

; The following routine ServiceRecv will be called when a character is waiting
; It should retrieve that character, pass it to the MIDI drivers, and then
; branch back to the beginning of ServicePort, to see if any more chars are
; waiting.
ServiceRecv

; *** Place code here that retrieves a byte of data from the port ***

; Call MIDI tools this way if no error has occurred on receive (<A>
; contains the data read)
RecvOK
```

```
               jsl InBufGlue        ; call the MIDI tools
               bra ServicePort      ; and check for more data in or out

; Call MIDI this way if a reception error has occurred (<A> contains the
; data read)
RecvErr
               ldy #miDevReadErr    ; load Y with the error
               jsl InErrGlue        ; call the midi tools
               bra ServicePort

; The routine ServiceXmit will be called when the port is ready to send data.
; it will actually call the MIDI tools and get a character to send.
ServiceXmit

               jsl OutBufGlue       ; call the MIDI tools for the next char
               bcs NoMoreData       ; if the carry set then no data to send

; *** at this point the byte to transmit is in <A>, place your code to output
; it thru the port here ***


; Now that the data has been sent, you can either loop thru ServicePort again,
; or you could simply end and wait for the next interrupt to send another
; character. This sample will simply exit at this point
               bra Done             ; after sending the character end.

; NoMoreData is called when the MIDI Tools said that they did not have any
; more data to transmit, so we should turn off transmitter interrupts at
; this point in case our device likes to keep interrupting if its empty.
NoMoreData
               phd                  ; push the direct page reg on the stack
               jsl DRVRXmitIntOff   ; enable xmit interrupts
Done
               pld                  ; restore the DPage
               clc                  ; signal the interrupt as handled
               rtl                  ; and get outta here!

; The routine inbufglue should be called when you received a character from
; your port with no error and you want to pass it to the MIDI tools.
InBufGlue      pea $0400            ; push on the long address of the
               phd                  ; direct page and a proc status byte
               RTL                  ; and jump back to the MIDI tools

; The routine inErrGlue should be called when you received a character from
; your port and an error has occurred. In this case, it should still be passed
; to the MIDI driver, as it may still be useful
inErrGlue      pea $0500            ; push on the long address of the
               phd                  ; direct page and a proc status byte
               RTL                  ; and jump back to the MIDI tools

; The routine OutBufGlue should be called when you are ready to send a char
; out your port. The MIDI tools will will return with the character to send
; in <A>. If the MIDI tools have no more characters to send then OutBufGlue
; will return with the carry set.
OutBufGlue     pea $8400            ; push on the long address of the
               phd                  ; direct page and a proc status byte
               RTL                  ; and jump back to the MIDI tools
               EndP
```

```
               EJECT
******************************************************************************
*
*
DRVRPollRecv    PROC
*
* Description:    This routine is called by the MIDI tools when it wants to
*                 pool the port for data instead of waiting for an interrupt.
*                 its function is similar to that of the our interrupt handler
*                 except that it only does input.
*
* Inputs:     None
*
* Outputs:    Carry set if interrupt not serviced
*
* External Refs:
        Import InBufGlue
        Import InErrGlue
*
* Entry Points:    None
*
******************************************************************************
*

               phd                     ; first, save the current dpage
               tcd                     ; and use the MIDI DPage
               php
               SEI

ServicePort                            ; the interrupt was ours, continue

; This routine should test the port too see if the port has any data for use
; to receive. If it does, it calls the MIDI tools and hands it off. Also note
; this routine will turn off interrupts, since we wouldn't want any stray
; receiver interrupts to spoil our fun and grab the data from us. (This is
; very important for certain types of ports which may signal that the port
; is ready and the generate an interrupt, thus leaving us in a situation where
; our interrupt routines could steal the interrupt right out from under us
; before we fetched it, thus allowing us to possibly double post a character.

; *** Insert code here to test for received data ***

               bne ServiceRecv         ; if chars waiting try receive it

; If no more data is waiting  exit this routine.
               plp
               pld                     ; restore the direct page
               clc                     ; clear the carry no errors possible
               RTL                     ; and return

; The following routine ServiceRecv will be called when a character is waiting
; It should retrieve that character, pass it to the MIDI drivers, and then
; branch back to the beginning of ServicePort, to see if any more chars are
; waiting.
ServiceRecv

; *** Place code here that retrieves a byte of data from the port ***
```

```
; Call MIDI tools this way if no error has occurred on receive (<A> contains
; the data read)
RecvOK
               jsl InBufGlue        ; call the MIDI tools
               bra ServicePort      ; and check for more data in or out

; Call MIDI this way if a reception error has occurred (<A> contains the
; data read)
RecvErr
               ldy #miDevReadErr    ; load Y with the error
               jsl InErrGlue        ; call the midi tools
               bra ServicePort
               EndP
               EJECT

******************************************************************************
*
*
DRVRPollXMit    PROC
*
* Description:    This routine is called when the MIDI tools wants to start
*                 an output stream. The tool set calls this routine for the
*                 first character of data, and then this routine is
*                 responsible for enabling transmitter interrupts and sending
*                 the character.
*
*
* Inputs:     None
*
* Outputs:    Carry set if interrupt not serviced
*
* External Refs:    None
        Import OutBufGlue
        Import DRVRXmitIntOn
*
* Entry Points:    None
*
******************************************************************************
*

               phd                  ; first, save the current dpage
               tcd                  ; and use the MIDI DPage
               php                  ; disable interrupts as we are now going
               SEI                  ; to turn on xmitter interrupts.

; First see if the port is ready to send any data, if not simply exit

; *** Insert code here to test if output is ready ***

               bcs Done             ; if not, then simply end

; The port is ready to accept a character for output so, call MIDI tools
; to get the next character

               jsl OutBufGlue       ; get the next character
               bcs Done             ; if carry set, no chars to xmit so end
```

```
               pha                     ; save the character to send
               phd                     ; push the direct page reg on the stack
               jsl DRVRXmitIntOn       ; enable xmit interrupts
               pla                     ; retrieve the character to send

; *** Insert code here to transmit a character ***
Done
               plp                     ; get the old interrupt status
               pld                     ; get the old direct page
               lda #0                  ; no errors are possible
               clc
               rtl

               EndP

               EJECT

******************************************************************************
*
*
DRVRXmitIntOn    PROC
*
* Description:    This routine will be called when the MIDI tools need to
*                 enable transmitter interrupts on your device.
*
*
* Inputs:     None
*
* Outputs:    None
*
* External Refs:
*
* Entry Points:
*
******************************************************************************
*

               php                     ; save proc status/interrupt state
               phd                     ; save the old direct page
               tcd                     ; use the MIDI tools DPage
               SEI                     ; disable interrupts

; *** Insert code here to enable transmitter interrupts on your device

               pld                     ; recover old direct page
               plp                     ; recover old interrupt state
               lda #0                  ; and return no-error (none possible)
               clc
               rtl
               EndP

******************************************************************************
*
*
DRVRXmitIntOff    PROC
*
* Description:    This routine will be called when the MIDI tools need to
```

```
*                  Disable transmitter interrupts on your device.
*
*
* Inputs:      None
*
* Outputs:     None
*
* External Refs:
*
* Entry Points:
*
******************************************************************************
*

               php                   ; save proc status/interrupt state
               phd                   ; save the old direct page
               tcd                   ; use the MIDI tools DPage
               SEI                   ; disable interrupts

; *** Insert code here to Disable transmitter interrupts on your device

               pld                   ; recover old direct page
               plp                   ; recover old interrupt state
               lda #0                ; and return no-error (none possible)
               clc
               rtl
               EndP

               EJECT

******************************************************************************
*
*
DRVRRecvIntOn    PROC
*
* Description:     This routine will be called when the MIDI tools need to
*                  enable receiver interrupts on your device.
*
*
* Inputs:      None
*
* Outputs:     None
*
* External Refs:
*
* Entry Points:
*
******************************************************************************
*

               php                   ; save proc status/interrupt state
               phd                   ; save the old direct page
               tcd                   ; use the MIDI tools DPage
               SEI                   ; disable interrupts

; *** Insert code here to enable receiver interrupts on your device

               pld                   ; recover old direct page
```

```
               plp                    ; recover old interrupt state
               lda #0                 ; and return no-error (none possible)
               clc
               rtl
               EndP


*******************************************************************************
*
*
DRVRRecvIntOff    PROC
*
* Description:    This routine will be called when the MIDI tools need to
*                 Disable receiver interrupts on your device.
*
*
* Inputs:     None
*
* Outputs:    None
*
* External Refs:
*
* Entry Points:
*
*******************************************************************************
*

               php                    ; save proc status/interrupt state
               phd                    ; save the old direct page
               tcd                    ; use the MIDI tools DPage
               SEI                    ; disable interrupts

; *** Insert code here to Disable receiver interrupts on your device

               pld                    ; recover old direct page
               plp                    ; recover old interrupt state
               lda #0                 ; and return no-error (none possible)
               clc
               rtl
               EndP


*******************************************************************************
*
*
DRVRNotImplemented    PROC
*
* Description:    Dummy routine, should leave the stack alone and return
*                 no error
*
*
* Inputs:     None
*
* Outputs:    None
*
* External Refs:
*
* Entry Points:
```

```
*
*****************************************************************************
*
               lda #0
               clc
               RTL
               EndP

               END
```

Further Reference:

- o   Apple IIGS Toolbox Reference Update

END OF FILE TN.IIGS.054

##
FILE: TN.IIGS.055
##

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#55: Avoiding ClrHeartBeat

Written by: Matt Deatherage July 1989

This Technical Note lists changes to the description for ClrHeartBeat. This information supersedes the description in the Apple IIGS Toolbox Reference Manual.

The Apple IIGS Toolbox Reference Manual gives the following cautionary note in the description for the call ClrHeartBeat:

> "A desk accessory may have installed tasks in the Heartbeat Interrupt Task queue. If you make a ClrHeartBeat call, you will remove those tasks. Therefore, under normal circumstances you should not make this call."

This isn't rude enough to get the point across to some people, so we'll try again:

The Heartbeat Interrupt Task queue does not belong to the application. Different portions of System Software can, and will, install Heartbeat Tasks. If these tasks are removed, anything from a system crash to media corruption may result. Nothing but System Software should make this call.

Further Reference

o Apple IIGS Toolbox Reference Manual

END OF FILE TN.IIGS.055

##
FILE: TN.IIGS.056
##

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#56: Managing Dynamic Segments

Written by: Eric Soldan July 1989

This Technical Note discusses application difficulties when transferring control to dynamic segments during low-memory conditions.

Dynamic segments have a drawback in low-memory situations. If the Loader cannot load the dynamic segment because there is not enough memory, it cannot just return to the application with an out-of-memory error since your program may have pushed parameters onto the stack which the dynamic segment would have removed. If this is the case, the Loader does not have a valid return address, so it cannot return to the application; therefore, it gives a fatal error and dies. Because of this problem, an application must make sure that there is sufficient memory available before calling the dynamic segment.

Just checking the amount of free memory does not guarantee that the dynamic segment will load. If memory is fragmented, the Loader may not be able to allocate a block large enough to load the segment, even if the total amount of free memory is greater than the size of the segment. Just checking MaxBlock is not a good solution either, since it can indicate that there is not enough free memory to load the segment when it is actually available. However, calling MaxBlock is preferable to just checking the amount of free memory, since not attempting to load the dynamic segment will not cause a fatal error.

Using the method of checking for out-of-memory conditions outlined in Apple IIGS Technical Note #51, Reserving Memory for the Toolbox, guarantees that there is sufficient space for the dynamic segment. This method involves maintaining a purgeable handle to a segment of memory at least as large as the dynamic segment and relocation dictionary. Before loading the dynamic segment, check to see if the handle has been purged. If it has not been purged, then you can load the dynamic segment without worrying about a fatal error from the Loader due to an out-of-memory condition. If it has been purged and you cannot reallocate it, then you know that there is not enough free memory available to load the dynamic segment.

Further Reference

- o GS/OS Reference, Volume 2
- o Apple IIGS Technical Note #51, Reserving Memory for the Toolbox

END OF FILE TN.IIGS.056

##
FILE: TN.IIGS.057
##

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#57: Preventing Memory Compacting and Purging

Written by: Dave Lyons July 1989

This Technical Note discusses a flag byte at location $E100CB that debugging utilities can use to temporarily prevent the Memory Manager from moving or purging memory.

If the byte at location $E100CB is non-zero, the Memory Manager will not move any memory blocks, and it will not purge any blocks while trying to allocate memory (PurgeHandle and PurgeAll will still purge blocks).

Debugging utilities may temporarily increment this byte to allocate memory in situations when it is not safe for existing memory blocks to be moved or purged.

This flag byte is for use only by debugging aids and System Software. It would be mind-numbingly stupid for an application to use this flag instead of using HLock and HUnlock, since the advantages of a Memory Manager architecture with relocatable blocks would be lost.

END OF FILE TN.IIGS.057

###
FILE: TN.IIGS.058
###

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#58: Keyboard Modifiers Register Anomaly

Written by: Dave Lyons July 1989

This Technical Note discusses an anomaly with the keyboard modifiers register at location $C025 which prevents it from always properly reflecting the state of the Control and Shift keys.

There are two cases where pressing the Control key turns on the Shift bit instead of the Control bit in the keyboard modifiers register:

- o An arrow key (or a Control key equivalent to an arrow key) is being held down and is repeating
- o The Space bar or Delete key is being held down and repeating with the Fast Space/Delete option selected in the Control Panel

Since the Event Manager reads the modifiers byte, desktop applications may be affected by this anomaly.

Further Reference

- o Apple IIGS Hardware Reference

END OF FILE TN.IIGS.058

##
FILE: TN.IIGS.059
##

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#59: Do Not Create Zero-Length Text Scraps

Written by: Dave Lyons July 1989

This Technical Note describes a problem with PutScrap if used to create a zero-length text scrap.

Do not create a text scrap (scrap type $0) with length zero. LEFromScrap in System Software 5.0 and earlier does not expect an existing text scrap to be empty, and it will trash random memory if it is. It is okay for there to be no text scrap, but if one exists it must have a non-zero length.

Even if your application does not cause a call to LEFromScrap, other applications and desk accessories have to live with what you put on the clipboard, so do not allow this condition to arise.

Further Reference

- Apple IIGS Toolbox Reference, Volume 2

END OF FILE TN.IIGS.059

##
FILE: TN.IIGS.060
##

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#60: Care and Feeding of NewMenu

Written by: Dave Lyons July 1989

This Technical Note discusses NewMenu in System Software 5.0 and earlier, where it does not expect its string parameter to cross a bank boundary. Make sure it doesn't.

NewMenu takes a pointer to a string; this string must not cross a bank boundary. If it does, a menu containing random garbage may result.

If your NewMenu strings are contained in your code segments, everything is fine--code segments cannot cross bank boundaries. Depending on your development environment, strings that are not in a code segment may or may not be allowed to cross bank boundaries. If you can find no other way to guarantee the strings will not cross a bank boundary, use NewHandle to allocate blocks with attributes $4010 (fixed, no bank cross) and copy the strings to these blocks.

If you create menus from resources, be sure the resources have their noBankCross attribute bits set. Note that a memory block that can cross a bank boundary usually does not, so your application may be working by accident.

Further Reference

- Apple IIGS Toolbox Reference, Volume 1

END OF FILE TN.IIGS.060

###
FILE: TN.IIGS.061
###

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#61: Window Title Handles

Written by: Dave Lyons July 1989

This Technical Note discusses extensions to SetWTitle and GetWTitle in System Software 5.0 and later which allow handles to be used as window titles.

Prior to System Software 5.0, window titles were pointers to Pascal-style strings (with a leading length byte), but now window titles can be stored in handles, with bit 31 of titlePtr set to indicate that the parameter is actually a handle.

Once you call SetWTitle with a handle for the title parameter, the handle belongs to the Window Manager,which will dispose of it when the window is closed or retitled. You must not dispose of the handle yourself, and you must not change the data it contains.

Further Reference

- o Apple IIGS Toolbox Reference, Volume 2

END OF FILE TN.IIGS.061

```
#####################################################################
### FILE: TN.IIGS.062
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#62: No Non-Solid Window Background Patterns

Written by: Dave Lyons July 1989

This Technical Note discusses why window background patterns should always be solid; non-solid patterns are not always drawn with the expected alignment.

When the Window Manager erases part of a window's content area to its port's background pattern, it is not always aligned with already-drawn parts of the window. With a solid background pattern, this has no visible effect; however, if you try to use a grid, for example, the effect is obvious.

To simulate a non-solid background pattern, just erase the desired area to the pattern you want in your update routine. For best results, use a solid background pattern of the color most common in the pattern you really want.

For example, if you want a white grid on a black background, give the window a solid black background pattern, and use FillRect during the update routine to draw the grid. If you keep the default white background pattern, the end result will be the same, but your window content will briefly be solid white before your update routine fills it with your pattern.

Further Reference

- Apple IIGS Toolbox Reference, Volume 2

END OF FILE TN.IIGS.062

###
FILE: TN.IIGS.063
###

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#63: Master Color Values

Written by: Jim Luther July 1989

This Technical Note documents master color values used for the Apple IIGS text, text background, and border colors.

There are times when you may want to make parts of the IIGS Super Hi-Res screen the same color as the text, text background, and border colors. This is particularly useful when using the Apple II Video Overlay Card. Table 1 lists each color using the names from the Control Panel CDA, the color register values used for that color by the color registers, and the master color value used for that color by the Super Hi-Res screen.

Color Name	Color Register Value	Master Color Value
Black	$0	$0000
Deep Red	$1	$0D03
Dark Blue	$2	$0009
Purple	$3	$0D2D
Dark Green	$4	$0072
Dark Gray	$5	$0555
Medium Blue	$6	$022F
Light Blue	$7	$06AF
Brown	$8	$0850
Orange	$9	$0F60
Light Gray	$A	$0AAA
Pink	$B	$0F98
Light Green	$C	$01D0
Yellow	$D	$0FF0
Aquamarine	$E	$04F9
White	$F	$0FFF

Table 1-Master Color Values

The Apple IIGS Hardware Reference documents the color registers at $C022 and $C034, and the Apple IIGS Toolbox Reference, Volume 2 documents the master color values.

Further Reference

- Apple IIGS Hardware Reference, pp. 58, 76-78

- o   Apple IIGS Toolbox Reference, Volume 2, pp. 16-31

END OF FILE TN.IIGS.063

##
FILE: TN.IIGS.064
##

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#64: Apple IIGS Installer and Installer Scripts

Revised by: Jim Luther September 1989
Written by: Jim Luther & "Jay" Schaffer July 1989

This Technical Note describes how the Apple IIGS Installer program executes script files and documents how to write script files for it. Note that some of the information in this Note is specific to Installer V1.10.
Changes since July 1989: Changed the sourcePrefix and sourcePathname field descriptions, since sourcePrefix must not be empty if any sourcePathname fields are partial pathnames.

Introduction

The Apple IIGS Installer, a utility program that is included with Apple IIGS System Software, can be used to install System Software or applications on a given volume. "Scripts" control the Installer, and they are simply lists of files with information about where and how to install those files. The user interface of the Installer is described in the Apple IIGS System Tools Manual. This Note describes how the Installer executes scripts and how to write scripts to install your applications.

Installer Setup on Disk

Setting up the Installer on your application disk is a simple procedure.

1. Copy the Installer program to your application disk.
2. Create a subdirectory (folder) named Scripts at the same directory level as the Installer program.
3. Copy your scripts into the Scripts subdirectory.

How the Installer Processes Scripts

The Installer reads script files into memory in their entirety, parses them, strips them of all comments, compacts them, then verifies them. It then checks the scriptFlags field to see if a Caution alert should be displayed. This facility permits the script writer to force the user to read the script's help message and make a choice to either continue with file manipulations or skip the installation altogether, which is especially useful when a script installation would be inappropriate on a certain volume.

The Installer then executes the script in two passes. The first pass determines if the update can be completed by calculating the total size of the files to be deleted from the destination volume and of the files to be installed. If there is not sufficient room on the destination volume, the Installer determines the amount of additional space required to complete installation (number of blocks needed divided by two, plus one), reports this result to the user in terms of kilobytes, then terminates execution of the script. It is impossible to determine directory block requirements with complete accuracy. The Installer's space calculation algorithms are good, but they are not perfect.

If the first pass determines that there is sufficient room for the complete update, the Installer continues with the second pass, deleting and copying files in accordance with the instructions contained in the script flags. The Installer "blindly" unlocks locked files and folders, creates necessary subdirectories if they do not already exist, and replaces requested files without regard to version numbers or creation dates of existing files.

The user may terminate execution of any script (and of those which follow) by pressing the Open Apple-Period key combination. The Installer checks for key-down events between every file transfer and at the end of the first pass. If the user requests termination, the Installer warns of the possibility of leaving an unknown mix of file versions on the volume and gives the user the opportunity to continue with the installation or to terminate as requested. (See the "Error Handling" section for more details.)

Scripts are typically written with the ability to remove all of their related files from a particular volume (i.e., in case of an accidental installation); however, they do not have the ability to remove directories which contain files (even if the script installed them), and they can neither recover nor list files which were deleted during the installation process.

After processing all the instructions in a script, the Installer checks to see if additional scripts are selected, and, if they are, it executes them in the order in which they appear in the update selection window until all scripts are successfully completed. Once all selected scripts are completed, the Installer notifies the user that the installation or removal process was successful.

It is important to note several facts about script execution:

- Each script is processed from beginning to end as if it were the only script selected.
- If the execution of a script generates an error, or if the user terminates further processing of a script, the queue is cleared of any additional scripts waiting to be executed and control returns to the user.
- It is possible for the Installer to execute several scripts successfully before encountering one which cannot be executed due to insufficient space on the destination volume.
- All selected scripts use the folder that the user selects as the "Application Folder."

If a user installs or removes system files (i.e., tools, fonts, drivers, etc.) from the boot volume, it may create problems. Therefore, whenever a system level update occurs on a boot volume, the Installer disables all desk accessories and closes the Sys.Resources file. When the user quits the Installer after a system level update, it alerts the user of the need to restart the system, and the default response to this alert is to restart.

Error Handling

User Cancel Request

If the user cancels script execution any time after it has started (i.e., by pressing the Open-Apple-Period key combination), the Installer treats it as an error condition since there is likely an unknown mix of file versions on the volume. In this case, the Installer gives the user the opportunity to continue with the installation or to terminate as requested. A user-initiated cancel request is not acknowledged until the current file copy or delete request is complete. Terminating script execution also clears the queue of other scripts waiting for execution and returns control to the user.

Non-Recoverable Errors

Some errors are simply fatal. If a directory or file is corrupted, the media is bad, or the selected script is longer than 65,535 bytes, the Installer halts execution of the script and alerts the user that a fatal error has occurred with a Stop alert box. Clicking the OK button in this alert box clears the queue of other scripts waiting for execution and returns control to the user.

Script Errors and File Not Found Errors

When the Installer detects a script error or a File Not Found error, it reports the name of the source file and destination file it was processing with the normal error message. This additional information should help script writers find the offending fileSpecification field. If the error is associated with the header, no filename is reported. This condition clears the queue of other scripts waiting for execution and returns control to the user.

Volume Not Found Errors

Volume Not Found errors produce a dialog box prompting the user to insert the missing volume. If the user clicks the OK button, the Installer attempts the file access call again, but if the user clicks the Cancel button, the Installer flags it as an error condition, clears the queue of other scripts waiting for execution, and returns control to the user.

Script File Composition

A script is simply a list of instructions for the Installer, and it can specify that files be copied from a source volume to a destination volume (or directory, when applicable) or that files be removed from a destination volume. Script files are ASCII files (file type $04) containing printable ASCII characters (i.e., with the high-bit clear). The directory in which the Installer resides must contain a directory named Scripts, in which all script files visible to that copy of the Installer must be located. Script files may not exceed 65,535 bytes in length. Any attempt to execute a script larger than this size produces a non-recoverable error.

A script consists of a header field followed by any number of fileSpecification and comment fields. These fields are separated by tildes (~). Two consecutive tildes signal the end of the script, and any additional

characters past the end of script marker are ignored. Figure 1 shows the
syntax diagram for a script.

```
       ________                                                       ____
  __>| header |_____________________________________________>( ~~ )__>
     |________|  \     ___      ___________________       /   (____)
                  \__>( ~ )___>| fileSpecification |_     |
                  /   (___) \  |___________________| \    |
                  |         |     _________          |    |
                  |         \__>| comment |_________\____/
                  |             |_________|           \
                  \______________________________________/
```

Figure 1-Script Syntax Diagram

header Field

The header field consists of the scriptIdentifier, scriptVersion, scriptFlag,
scriptName, and scriptHelp fields, and it may also contain an optional
sourcePrefix field. These fields supply the installer with general
information about the script file. No comments are permitted within the
header field. Figure 2 shows the syntax diagram for the header field.

```
            __________________
     ___>| scriptIdentifier |__
         |__________________|  \
      _________________________/
     /    __________________
     \__>| scriptVersion    |__
         |__________________|  \
      _________________________/
     /    __________________
     \__>| scriptFlag       |__
         |__________________|  \
      _________________________/
     /    __________________
     \__>| scriptName       |__
         |__________________|  \
      _________________________/
     /    __________________
     \__>| scriptHelp       |________>
         |__________________|  \  /
      _________________________/  |
     /    __________________      |
     \__>| sourcePrefix     |_____/
         |__________________|
```

Figure 2-header Field Syntax Diagram

The scriptIdentifier field identifies the text file as a script file, and it
consists of eight characters ("SCRIPT" followed by two carriage returns, or 53
43 52 49 50 54 0D 0D in hexadecimal). Figure 3 shows the syntax diagram for
the scriptIdentifier field.

```
        ___________________________________________________
  __>( "SCRIPT" followed by 2 carriage return characters )__>
     (___________________________________________________)
```

Figure 3-scriptIdentifier Field Syntax Diagram

The scriptVersion field defines the minimum version of the Installer program that can read and execute the instructions in this script file. It should normally consist of seven characters ("V1.10" followed by two carriage returns, or 56 31 2E 31 30 0D 0D in hexadecimal).

Version 1.0 of the Installer moves only the data fork and does not return an error. For compatibility with the original release of the Installer, the value of scriptVersion is V1.00. Scripts which move extended files (i.e., files with resource forks) or work with an AppleShare volume must have a scriptVersion of V1.10. Figure 4 shows the syntax diagram for the scriptVersion field.

```
           __________________________________________________
        __>( "V1.10" followed by 2 carriage return characters )__>
           (__________________________________________________)
```

Figure 4-scriptVersion Field Syntax Diagram

The scriptFlag field defines the directory requirements of the script file. The first character of the scriptFlag field must be either the uppercase character "R" (indicating that the installation must occur at the root directory, such as in a System Software update) or the uppercase character "X" (indicating that the user must specify the directory where installation should take place).

The second character of the scriptFlag field must be either an uppercase or lowercase character "R" (indicating that the Remove command is valid for this script) or an uppercase or lowercase character "N" (indicating that the Remove command is not valid and the button should be dimmed and inactive). If this character is lowercase, before any file manipulations begin, the Installer displays a Caution alert with the contents of the scriptHelp field and button controls to permit the user to choose whether to execute the script or to skip it and go to the next script, if any.

For example, a scriptFlag field might contain the following four characters: "Rr" followed by two carriage returns, or 52 52 0D 0D in hexadecimal. Figure 5 shows the syntax diagram for the scriptFlag field.

```
         ___            ___           ______________________________
    _____>( R )_________>( R )______>( 2 carriage return characters )__>
      \   (___)  /  \    (___)   /   (______________________________)
      |    ___   |  |     ___    |
      \__>( X )__/  |\__>( r )__/|
          (___)     |    (___)   |
                    |     ___    |
                    |\__>( N )__/|
                    |    (___)   |
                    |     ___    |
                    \___>( n )___/
                         (___)
```

Figure 5-scriptFlag Field Syntax Diagram

The scriptName field defines the name of the script as it appears in the Installer's script selection window. It is recommended that care be taken to use a name that fits within the display window. This field consists of any number of characters ending with a carriage return and may not include a tilde or carriage return. An example of scriptName might be: "Example Script" followed by a carriage return, or 45 78 61 6D 70 6C 65 20 53 63 72 69 70 74 0D in hexadecimal. Figure 6 shows the syntax diagram for the scriptName field.

```
                 ____________________________________________
    ________>( any character except ~ and carriage return )_____
      \  /   (____________________________________________)  \  \
      |  \___________________________________________________/  |
      | ________________________________________________________/
      |/
      |                                         _________________
      \______________________________________>( carriage return )__>
                                              (_________________)
```

Figure 6-scriptName Field Syntax Diagram

The scriptHelp field defines the text which appears when the user clicks the Help button. It is recommended that care be taken to ensure the text fits within the help window. This field consists of any number of characters ending with two backslashes (\\) and a carriage return. It may not include two consecutive backslashes or a tilde; however, it may include carriage returns. An example of scriptHelp might be: "Help\\" followed by a carriage return, or 48 65 6C 70 5C 5C 0D in hexadecimal. Figure 7 shows the syntax diagram for the scriptHelp field.

```
                 _______________________________
    ________>( any character except \\ and ~ )_____
      \  /   (_______________________________)  \  \
      |  \______________________________________/  |
      | ___________________________________________/
      |/
      |          __________________________________
      \________>( \\ followed by a carriage return )__>
                (__________________________________)
```

Figure 7-scriptHelp Field Syntax Diagram

The optional sourcePrefix field is the prefix used with source files defined by partial pathnames. Either slashes (/) or colons (:) may be used as the pathname separator character. If there is no sourcePrefix, this entry must be empty. If no sourcePrefix is specified, all sourcePathname fields used within fileSpecification fields must be full pathnames. An example of a sourcePrefix might be: ":System.Disk:System", or 3A 53 79 73 74 65 6D 2E 44 69 73 6B 3A 53 79 73 74 65 6D in hexadecimal. Figure 8 shows the syntax diagram for the sourcePrefix field. GS/OS Reference defines legal pathnames and prefixes.

```
       ____________________
 __>( legal GS/OS prefix )__>
    (____________________)
```

Figure 8-sourcePrefix Field Syntax Diagram

fileSpecification Field

A fileSpecification field contains the instructions to copy a file to or remove a file from the destination volume (or directory, when applicable). A fileSpecification field is composed of the fileSpecWorkspace, fileSpecFlags, fileTypeAuxType, createDateTime, sourcePathname, and destinationPathname fields. The script may contain as many fileSpecification fields as necessary. Figure 9 shows the syntax diagram for the fileSpecification field.

```
      _____________________
___>| fileSpecWorkspace   |__
    |_____________________|  \
 ____________________________/
/     _____________________
\__>| fileSpecFlags       |__
    |_____________________|  \
 ____________________________/
/     _____________________
\__>| fileTypeAuxType     |__
    |_____________________|  \
 ____________________________/
/     _____________________
\__>| createDateTime      |__
    |_____________________|  \
 ____________________________/
/     _____________________
\__>| sourcePathname      |________>
    |_____________________|  \  /
 ____________________________/  |
/     _____________________     |
\__>| destinationPathname |_____/
    |_____________________|
```

Figure 9-fileSpecification Field Syntax Diagram

The fileSpecWorkspace field is 16 bytes that the Installer uses for work space, it can contain any character except a tilde, and it may not begin with a tilde or an asterisk (*). It is suggested that 15 readable characters followed by a carriage return might be easiest to see and count. An example of fileSpecWorkspace might be: ":::Workspace:::" followed by a carriage return, or 3A 3A 3A 57 6F 72 6B 73 70 61 63 65 3A 3A 3A 0D in hexadecimal. Figure 10 shows the syntax diagram for the fileSpecWorkspace field.

```
     ________________________________
__>( any 1 character except ~ and * )__
   (________________________________)  \
      _________________________________/
     /    ____________________________
     \__>( any 15 characters except ~ )__>
         (____________________________)
```

Figure 10-fileSpecWorkspace Field Syntax Diagram

The fileSpecFlags tell the Installer what this fileSpecification does. The fileSpecFlags field consists of the requiredFlag field followed by the optionalFlags field and a carriage return. Figure 11 shows the syntax diagram for the fileSpecFlags field.

```
     ______________    _______________    _________________
__>| requiredFlag |__>| optionalFlags |__>( carriage return )__>
   |______________|   |_______________|   (_________________)
```

Figure 11-fileSpecFlags Field Syntax Diagram

The requiredFlag field tells the Installer what to do with this fileSpecification when the Install or Remove buttons are used. The requiredFlag field must start with only one of the following characters: 1, 2, 3, or 4, and it must end with a carriage return. Any number of characters (except tilde and carriage return) may fall between the flag character and the ending carriage return. These additional characters are ignored by the Installer, making it possible to place comments within a requiredFlag field. Figure 12 shows the syntax diagram for the requiredFlag field.

The four requiredFlag characters tell the installer the following:

1 If the user clicks the Install button, delete the destinationPathname from the destination volume, if it exists, and copy the file from the source volume. If the user clicks the Remove button, delete the destinationPathname from the destination volume, if it exists.
2 If the user clicks the Install button, delete the destinationPathname from the destination volume, if it exists, and copy the file from the source volume. If the user clicks the Remove button, do nothing.
3 If the user clicks the Install button, delete the destinationPathname from the destination volume, if it exists. If the user clicks the Remove button, delete the destinationPathname from the destination volume, if it exists.
4 If the user clicks the Install button, delete the destinationPathname from the destination volume, if it exists. If the user clicks the Remove button, do nothing.

```
            ___
    ______>( 1 )______
      \    (___)   /  \
      |     ___    |  |
      |\__>( 2 )__/|  |
      |    (___)   |  |
      |     ___    |  |
      |\__>( 3 )__/|  |
      |    (___)   |  |
      |     ___    |  |
      \___>( 4 )___/  |
           (___)      |
      _______________/|
     /                |
     |                \_______________________________________
     |    ____________________________________________        \
     \__>( any character except ~ and carriage return )______ |
     /   (____________________________________________)  \  \|
     \_______________________________________________________/  |
                                        ______________________/
                                       /    _________________
                                       \__>( carriage return )__>
                                           (_________________)
```

Figure 12-requiredFlag Field Syntax Diagram

The optionalFlags field gives the Installer additional duties to perform with this fileSpecification when the Install or Remove buttons are used. The five option fields, B, C, D, F, and U (must be uppercase), within the optionalFlags field are formatted the same as the requiredFlag field. Figure 13 shows the syntax diagram for the optionalFlags field.

The five optionalFlags characters tell the installer the following:

- B This flag instructs the Installer to replace the boot code on blocks zero and one of the destination volume. The boot code replacement fileSpecification is reserved for use by Apple Computer, Inc.
- C The creation date and time of the file designated by the sourcePathname field must match the createDateTime entry in this fileSpecification field.
- D The designated destinationPathname should be deleted if, and only if, it has a creation date and time that is older than createDateTime. This flag must be used with a "4" requiredFlag.
- F The file type and auxiliary type of the file designated by the sourcePathname must match the fileTypeAuxType field in this fileSpecification field.
- U Update (replace) the existing destinationPathname only if it exists. This flag must be used with a "1" or a "2" requiredFlag.

```
        ___
______>( B )___________________________________________________________
  \    (___) \     ____________________________________________   /   \
  |           \__>( any character except ~ and carriage return )__/   |
  |           /   (____________________________________________)  \   |
  |           \___________________________________________________/   |
  |                                        ___________________________/
  |                                       /    _________________
  |                                       \__>( carriage return )__________
  |     ___                                   (_________________)          \
  |\__>( C )____________________________________________________________   |
  |    (___)  \     ____________________________________________   /   \   |
  |            \__>( any character except ~ and carriage return )__/   |   |
  |            /   (____________________________________________)  \   |   |
  |            \___________________________________________________/   |   |
  |                               _________________                    |   |
  | _____________________________( carriage return )<__________________/   |
  |/    ___                      (_________________)                       |
  |\__>( D )____________________________________________________________   |
  |    (___)  \     ____________________________________________   /   \   |
  |            \__>( any character except ~ and carriage return )__/   |   |
  |            /   (____________________________________________)  \   |   |
  |            \___________________________________________________/   |   |
  |                               _________________                    |   |
  | _____________________________( carriage return )<__________________/   |
  |/    ___                      (_________________)                       |
  |\__>( F )____________________________________________________________   |
  |    (___)  \     ____________________________________________   /   \   |
  |            \__>( any character except ~ and carriage return )__/   |   |
  |            /   (____________________________________________)  \   |   |
  |            \___________________________________________________/   |   |
  |                               _________________                    |   |
  | _____________________________( carriage return )<__________________/   |
  |/    ___                      (_________________)                       |
  |\__>( U )____________________________________________________________   |
  |    (___)  \     ____________________________________________   /   \   |
  |            \__>( any character except ~ and carriage return )__/   |   |
```

```
 |            /   (____________________________________________)  \   |  |
 |            \____________________________________________________/   |  |
 |                                  _________________                  |  |
 | _____________________________( carriage return )<__________________/  |
 |/                              (_________________)                      |
 |                                                                        |
 \________________________________________________________________________\__>
```

Figure 13-optionalFlags Field Syntax Diagram

The fileTypeAuxType field is used if the "F" optionalFlags field is present in the fileSpecification field. If the fileTypeAuxType field is used, it must start with a fileType field and an auxType field and must end with a carriage return. Any number of characters (except tilde and carriage return) may fall between the auxType field and the ending carriage return. These additional characters are ignored by the Installer, making it possible to place comments within the fileTypeAuxType field. If the "F" optionalFlags field is not used, then the fileTypeAuxType field must be only a carriage return. For a list of current file types and auxiliary types, see the Apple II File Type Notes. Figure 14 shows the syntax diagram for the fileTypeAuxType field.

```
                                                     _________________
 _____________________________________________>( carriage return )__>
   \        __________     _________        /   (_________________)
    \__>| fileType |__>| auxType |________/
        |__________|   |_________|  \  /
           ________________________/  \________________________
          /    ____________________________________________    \
          \__>( any character except ~ and carriage return )__/
          /   (____________________________________________)  \
          \____________________________________________________/
```

Figure 14-fileTypeAuxType Field Syntax Diagram

The fileType part of the fileTypeAuxType field consists of four, and only four, hexadecimal digits. These four digits identify a GS/OS file type if the "F" optionalFlags field is present in the fileSpecification field. An example of fileType might be: "00B3", or 30 30 42 33 in hexadecimal. Figure 15 shows the syntax diagram for the fileType field.

```
    _______________________________________
 __>( four and only four hexadecimal digits )__>
    (_______________________________________)
```

Figure 15-fileType Field Syntax Diagram

The auxType part of the fileTypeAuxType field consists of eight, and only eight, hexadecimal digits. These eight hexadecimal digits identify a GS/OS auxiliary type if the "F" optionalFlags field is present in the fileSpecification field. An example of auxType might be: "00000000", or 30 30 30 30 30 30 30 30 in hexadecimal. Figure 16 shows the syntax diagram for the auxType field.

```
    _________________________________________
 __>( eight and only eight hexadecimal digits )__>
    (_________________________________________)
```

Figure 16-auxType Field Syntax Diagram

The createDateTime field is used if the "C" or "D" optionalFlags fields are present in the fileSpecification field. If the createDateTime field is used, it must start with a date field, a single space and a time field and must end with a carriage return. Any number of characters (except tilde and carriage return) may fall between the time field and the ending carriage return. These additional characters are ignored by the Installer, making it possible to place comments within the createDateTime field. If the "C" or "D" optionalFlags fields are not used, then the createDateTime field must be only a carriage return. Figure 17 shows the syntax diagram for the createDateTime field.

```
                                                         _________________
    ______________________________________________>( carriage return )__>
      \      ______     _______     ______       /   (_________________)
       \__>| date |__>( space )__>| time |__     |
           |______|   (_______)   |______|  \    |
               ______________________________/    \___________________
              /    ____________________________________________       \
              \__>( any character except ~ and carriage return )__/
              /   (____________________________________________)  \
              \___________________________________________________/
```

Figure 17-createDateTime Field Syntax Diagram

The date subfield of the createDateTime field is nine ASCII characters consisting of the day of the month, a space, a three-character month abbreviation, a space, and the year. The day of the month is a two-character number between 01 and 31. The month abbreviation may be "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", or "Dec" in any combination of uppercase and lowercase characters. The year is a two-character number between 00 and 99. An example of the date subfield might be: "31 Mar 57", or 33 31 20 4D 61 72 20 35 37 in hexadecimal. Figure 18 shows the syntax diagram for the date subfield.

```
         _________________________________      _______
    __>( two and only two decimal digits )__>( space )__
       (_________________________________)   (_______)  \
       _________________________________________________/
      /    ____________________________________      _______
      \__>( three character month abbreviation )__>( space )__
          (____________________________________)   (_______)  \
       _______________________________________________________/
      /    _________________________________
      \__>( two and only two decimal digits )__>
          (_________________________________)
```

Figure 18-date Field Syntax Diagram

The time subfield of the createDateTime field is five ASCII characters consisting of the military format hour of the day, a colon, and the minute of the hour. The hour of the day is a two-character number between 00 and 23. The minute of the hour is a two-character number between 00 and 59. An example of the time subfield might be: "08:30", or 30 38 3A 33 30 in hexadecimal. Figure 19 shows the syntax diagram for the time subfield.

```
         _________________________________
    __>( two and only two decimal digits )__
```

```
           (_________________________________)  \
           _____________________________________/
          /    _______     _________________________________
          \__>( colon )__>( two and only two decimal digits )__>
              (_______)   (_________________________________)
```

Figure 19-time Field Syntax Diagram

The sourcePathname field describes the name and location of the source file. The sourcePathname field consists of a valid GS/OS pathname followed by a carriage return. If the sourcePathname is a partial pathname, the sourcePrefix in the header field is used to complete the full pathname. If no sourcePrefix is specified in the header field, all sourcePathname fields must be full pathnames. If the fileSpecFlags indicate removal only, then the sourcePathname is a carriage return only. No optional comments are permitted in this field. Figure 20 shows the syntax diagram for the sourcePathname field. GS/OS Reference defines legal pathnames and prefixes.

```
                                              _________________
    _____________________________________>( carriage return )__>
      \      ______________________     /   (_________________)
       \__>( valid GS/OS pathname )__/
           (______________________)
```

Figure 20-sourcePathname Field Syntax Diagram

The destinationPathname field describes the name and location of the destination file. The destinationPathname field consists of a valid GS/OS partial pathname (the prefix has already been set by the Installer to the location of the destination directory, either the root directory or a user selected directory) followed by a carriage return. No optional comments are permitted in this field. Figure 21 shows the syntax diagram for the destinationPathname field. GS/OS Reference defines legal pathnames and prefixes.

Note that GS/OS now allows filenames to contain both uppercase and lowercase characters. Although filenames are not case sensitive, you should be consistent in your use of uppercase and lowercase usage in the destinationPathname field. Whatever you use here is what everyone sees.

```
        ______________________________     _________________
    __>( valid GS/OS partial pathname )__>( carriage return )__>
       (______________________________)   (_________________)
```

Figure 21-destinationPathname Field Syntax Diagram

comment Field

The comment field allows commenting script files. A comment field must begin with an asterisk. The Installer ignores all characters within a comment field, except tilde, and the comment field ends at the first tilde encountered. Figure 22 shows the syntax diagram for the comment field.

```
                ___
            __>( * )_______________________________________>
               (___)  \       ________________________    /
                       \__>( any character except ~ )__/
                       /   (________________________)  \
                       \________________________________/
```

Figure 22-comment Field Syntax Diagram

Examples

Now that the script language is described, it's time to look at a couple of example scripts. The first example, CD-ROM from the System.Tools disk, installs the files necessary for you to use CD-ROM drives. The CD-ROM script is an example of using the Installer to install or update existing software. The second example, Advanced Disk Utility from the System.Tools disk, installs the files necessary to update the Advanced Disk Utility program. The Advanced Disk Utility script is an example of using the Installer to install an application in any directory on the destination volume. In both examples (Examples 1 and 2), carriage returns are shown with a paragraph mark ([p]) since they are used as delimiters within scripts.

The CD-ROM Script

The header field starts with "SCRIPT" to identify this text file as a script file. The scriptVersion is "V1.10" because this script may have to copy the resource fork of a file. The scriptFlag field is "RR", which tells the Installer to install at the root directory level and that the Remove button is valid for this script. The second "R" character in the scriptFlag field is uppercase, which tells the Installer not to display a Caution alert with the contents of the scriptHelp field. The scriptName field is "CD-ROM". The scriptName is shown in the Installer's list of scripts. The scriptHelp field (everything between the scriptName field and the "\\" delimiter) is the text that will be displayed if the Installer's Help button is used. The sourcePrefix is ":SYSTEM.TOOLS". That is the name of the volume where the source files for this update are found.

After the header field, there is a single comment field and then five fileSpecification fields. The comment field starts at the asterisk after the first tilde and ends at the next tilde. All five fileSpecification fields start with the suggested 16-byte fileSpecWorkSpace (":::WorkSpace:::[p]") and end at the next tilde.

The first, fourth, and fifth fileSpecification fields use the "1" requiredFlag. This flag tells the Installer to copy the sourcePathname to the destinationPathname if the Install button is used, or to delete the destinationPathname if the Remove button is used. Notice the three blank lines after the "1" requiredFlag. The first blank line marks the end of the fileSpecFlags. The fileTypeAuxType field, the second blank line, is blank because the "F" optionalFlags field is not used. The createDateTime field, the third blank line, is blank because the "C" and "D" optionalFlags are not used.

The second fileSpecification field uses the "3" requiredFlag to tell the Installer to delete the destinationPathname, "System:Drivers:SCSI.Driver", if either the Install or the Delete button is used. SCSI.Driver is the interim SCSI driver from System Software 4.0. The sourcePathname field, the fourth blank line after the "3" requiredFlag, is not needed since the "3" requiredFlag is used.

The third fileSpecification field uses the "2" requiredFlag to tell the Installer to delete the destinationPathname, "System:Drivers:SCSI.Manager" if the Install button is used. The Installer does not delete the destinationPathname if the Remove button is used. The "2" requiredFlag prevents this script from removing SCSI.Manager, which might have been installed by another script.

Two consecutive tildes after the fifth fileSpecification field signal the end of this script.

```
SCRIPT[p]
[p]
V1.10[p]
[p]
RR[p]
[p]
CD-ROM[p]
This script installs the files necessary for you to use CD-ROM drives.  The
selected disk must be a startup disk.\\[p]
:SYSTEM.TOOLS~*[p]
 This is the Installer script necessary to move the CD-ROM files from
:SYSTEM.TOOLS to the user's startup disk.[p]
~:::Workspace:::[p]
1[p]
[p]
[p]
[p]
System:FSTs:HS.FST[p]
System:FSTs:HS.FST[p]
~:::Workspace:::[p]
3[p]
[p]
[p]
[p]
[p]
System:Drivers:SCSI.Driver[p]
~:::Workspace:::[p]
2[p]
[p]
[p]
[p]
System:Drivers:SCSI.Manager[p]
System:Drivers:SCSI.Manager[p]
~:::Workspace:::[p]
1[p]
[p]
[p]
[p]
System:Drivers:SCSICD.Driver[p]
System:Drivers:SCSICD.Driver[p]
~:::Workspace:::[p]
1[p]
[p]
[p]
[p]
System:Desk.Accs:CDRemote[p]
System:Desk.Accs:CDRemote[p]
~~
```

Example 1-CD-ROM Script

The Advanced Disk Utility Script

The header field starts with "SCRIPT" to identify this text file as a script file. The scriptVersion is "V1.10" because this script may have to copy the resource fork of a file. The scriptFlag field is "XR", which tells the Installer the user must specify the directory where the installation should take place and that the Remove button is valid for this script. The second character (R) in the scriptFlag field is uppercase, which tells the Installer not to display a Caution alert with the contents of the scriptHelp field. The scriptName field is "Advanced Disk Utility". The scriptName will be shown in the Installer's list of scripts. The scriptHelp field (everything between the scriptName field and the "\\" delimiter) is the text that will be displayed if the Installer's Help button is used. The sourcePrefix is ":SYSTEM.TOOLS". That is the name of the volume where the source files for this update are found.

After the header field, there is a single comment field then one fileSpecification field. The comment field starts at the asterisk after the first tilde and ends at the next tilde. The fileSpecification field starts with the suggested 16-byte fileSpecWorkSpace (":::WorkSpace:::[p]") and ends at the next tilde.

The fileSpecification field uses the "1" requiredFlag. This tells the Installer to copy the sourcePathname to the destinationPathname if the Install button is used or to delete the destinationPathname if the Remove button is used.

Two consecutive tildes signal the end of this script.

```
SCRIPT[p]
[p]
V1.10[p]
[p]
XR[p]
[p]
Advanced Disk Utility[p]
This script installs the files necessary to update the Advanced Disk Utility
program.  These files will be installed on the selected disk.\\[p]
:SYSTEM.TOOLS~*[p]
 This is the Installer script necessary to update the Advanced Disk Utility
file from :SYSTEM.TOOLS to the user's disk.[p]
~:::Workspace:::[p]
1[p]
[p]
[p]
[p]
Adv.Disk.Util[p]
Adv.Disk.Util[p]
~~
```

Example 2-Advanced Disk Utility Script

Further Reference

- Apple IIGS System Tools Manual

- o GS/OS Reference

END OF FILE TN.IIGS.064

```
#####################################################################
### FILE: TN.IIGS.065
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#65: Control-^ is Harder Than It Looks

Written by: Dave Lyons September 1989

This Technical Note describes a problem using Control-^ to change the text cursor with programs that use GETLN.

On the Apple IIGS, typing Control-^ changes the cursor to the next character typed. This feature works properly from the keyboard, but there is a problem when programs print the control sequence. Try entering the following from AppleSoft to demonstrate this problem:

```
NEW
PRINT CHR$(30);"_"
```

It changes the cursor into a blinking underscore, as expected. But now enter the following:

```
12345 HOME
LIST
```

You should see 2345 HOME, which demonstrates that the first character is ignored. This is a problem with GETLN, which AppleSoft uses to read each line of input. Even if your program does not use this routine, you should be aware of this problem since it will occur the next time another program uses GETLN.

Since changing the cursor works fine when done from the keyboard, the way to work around this problem is to have your program simulate the appropriate keypresses for GETLN.

```
301: CLD                      ; required by BASIC.SYSTEM
302: STA ($28),Y              ; remove cursor if present
304: LDY $0300                ; get index into simulated-keys list
307: LDA $310,Y               ; get a simulated keypress
30A: INC $0300                ; point to the next key for next time
30B: RTS                      ; return the key to GETLN

310: 9E DF 8D                 ; Ctrl-^, underscore, return

100 POKE 768,0:PRINT CHR$(4);"IN#A$301":REM Start getting simulated keys
110 INPUT "";A$
120 PRINT CHR$(4);"IN#0":REM Get real keys again
```

From an assembly-language program, the equivalent of IN#A$301 is storing $01 and $03 in locations $38 and $39, while the equivalent of INPUT is JSR $FD6A

(GETLN). (Store a harmless prompt character, like $80, into location $33 first.)

Further Reference

---

o Apple IIGS Firmware Reference, p. 77

END OF FILE TN.IIGS.065

###
FILE: TN.IIGS.066
###

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#66: ExpressLoad Philosophy

Written by: Matt Deatherage September 1989

This Technical Note discusses the ExpressLoad feature and how it relates to the standard Loader and your application.

Speedy the Loader Helper

ExpressLoad is a GS/OS feature which is usually present with System Software 5.0. The system does not load it on machines with 512K or less RAM, and there is always the chance that someone has removed it from the System directory.

ExpressLoad operates on Object Module Format (OMF) files which have been "expressed," using either the APW tool Express (or it's MPW counterpart, ExpressIIGS) or created that way by a linker. Expressed files contain a dynamic data segment named either ExpressLoad or ~ExpressLoad at the beginning of the file. (Current versions of Express and ExpressIIGS create ~ExpressLoad segments, which is the preferred naming convention; older versions created ExpressLoad segments, and should be re-Expressed for future compatibility.) This segment contains information ExpressLoad uses to load the files more quickly than the System Loader, including such things as file offsets to segment headers, mappings of old segment numbers to new segment numbers (these files may have their segments rearranged for optimal performance), and file offsets to relocation dictionaries.

Two Loaders, Two Missions, One Function

The System Loader's function is to interpret OMF. It takes files on disk (or in memory) and transforms them from load files into relocated 65816 code. It does this very well, but in a very straightforward way. For example, when the System Loader sees the instruction to right-shift a value n times, it loads a register with the value and performs a right-shift n times.

ExpressLoad has a different mission. It relies upon the System Loader to handle OMF in a straightforward fashion so it can concentrate upon handling the most common OMF cases in the fastest possible way. For example, when asked for a specific segment in a load file, the System Loader "walks" the OMF until it finds the desired segment. ExpressLoad, however, goes directly to the desired segment since an Expressed file contains precalculated offsets to each segment in the ExpressLoad segment.

Since ExpressLoad focuses on the common things performed by the majority of applications, it may not support those applications which rely upon certain features of OMF or the System Loader. In these cases, the System Loader loads the file as is expected.

ExpressLoad always gets first crack at loading a file, and if it is an Expressed file that ExpressLoad can handle, it loads it. If the file is not an Expressed file, the System Loader loads it instead. It is the same process when working with a file that has already been loaded (i.e., loading or unloading segments).

Because an Expressed file is a standard OMF file with an additional segment, Expressed files are almost fully compatible with the System Loader (although it cannot load them any faster than before). Refer the following section for potential problems.

Working With ExpressLoad

As ExpressLoad is intimate in its relationship with the System Loader, most applications work seamlessly with it; however, there are some potential problems about which you should be aware.

- Don't mix Expressed files and normal OMF files with the same user ID. For example, if your application uses InitialLoad with a separate file, make sure that if it and your main application share the same user ID that they are both either Expressed files or normal OMF files.

- Don't use a user ID of zero. In the past, use of zero told the System Loader to use the current user ID; however, now both the System Loader and ExpressLoad have a current user ID. Be specific about user IDs when loading.

- Avoid loading and unloading segments by number. Since Expressed files may have their segments rearranged, if an Expressed file is loaded by the System Loader, references to segments by number may be incorrect.

- Avoid using GetLoadSegInfo. This call returns System Loader data structures which are not supported by ExpressLoad.

- Don't try to load segments in files which have not been loaded with the call InitialLoad. This process was never a very good idea, and it is now apt to cause problems.

- Don't close files with a reference number of zero. ExpressLoad (and now, the System Loader) keep your file open if there are dynamic segments, so the file won't have to be opened again to load them. Closing a file with a reference number of zero may close your application behind the Loader's back. (It also closes the resource fork of your application, which is another good reason not to do it.)

- Don't have segments that link to other files. ExpressLoad does not support this type of link.

Further Reference

o GS/OS Reference

END OF FILE TN.IIGS.066

###
FILE: TN.IIGS.067
###

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#67: LaserWriter Font Mapping

Written by: Suki Lee & Jim Luther September 1989

This Technical Note discusses the methods used by the Apple IIGS Print Manager to map IIGS fonts to the PostScript(R) fonts available with an Apple LaserWriter printer.

Version 2.2 and earlier of the Apple IIGS LaserWriter driver depend solely upon font family numbers as unique font identifiers. There is a table built into the driver which maps the known font family numbers to the built-in LaserWriter family fonts. Any fonts which are not built-in are created in the printer from its bitmap font strike. Under this implementation, all font family numbers not known at the time the driver was written print using bitmap fonts. This driver knows nothing of any other fonts which may reside in the printer.

There have been many requests for the driver to take advantage of other available PostScript fonts to get high quality output from the LaserWriter. PostScript fonts from Adobe's font library, or from other PostScript font manufacturers, can be downloaded to the printer from a Macintosh and remain in the printer for use until power off. Currently there is no means to download a PostScript font with an Apple IIGS.

The Apple IIGS LaserWriter driver version 3.0 makes use of most resident PostScript fonts in the LaserWriter when requested. If the font is not available, then the bitmap font is used. The driver queries the printer at the start of a job for the font directory listing. The listing consists of names of all the fonts in the printer, built-in or downloaded. This information is kept locally for look up using the name of the requested font.

Issues

All Apple IIGS fonts contain a family name and a family number. The Apple IIGS currently identifies fonts using the family number; however, this identification method may change in the future, due to the complexity of tracking unique matches between font family names and font family numbers.

PostScript identifies its fonts by name (case sensitive) and knows nothing of any font family numbering system, Macintosh or Apple IIGS, which might be attached to a particular font. Most PostScript font families include plain, bold, italic and bold italic fonts. Some fonts families may also have serif and sans serif fonts or fonts of different weights (line thickness). These fonts are generally named by adding a style suffix to the base family name.

Unfortunately, there is no uniform method for naming fonts, since most fonts were named by their designers and many of the names have historical significance.

The three examples shown in Table 1 show three variations of the plain font, two variations of the bold style, three variations of the italic style, and three variations of the bold italic style. There are others such as ZapfChancery-MediumItalic, Korinna-KursivRegular, and LetterGothic-Slanted which all denote the italic style of the respective font family.

Style	Font names		
plain	Helvetica	Times-Roman	AvantGarde-Book
bold	Helvetica-Bold	Times-Bold	AvantGarde-Demi
italic	Helvetica-Oblique	Times-Italic	AvantGarde-BookOblique
bold italic	Helvetica-BoldOblique	Times-BoldItalic	AvantGarde-DemiOblique

Table 1-Example Font Names

The Macintosh LaserWriter driver uses a mapping scheme to compose a full PostScript font name. It relies on the Font Family Definition Record 'FOND' resource to provide a style mapping table containing the appropriate suffixes.

Since there are no similar resource on the Apple IIGS, the Apple IIGS LaserWriter driver adopts the following approach. The driver has full knowledge of all LaserWriter family built-in fonts (see Table 2 for a list of these built-in fonts) and uses the correct name for all style variations of the fonts. For all other fonts, the driver uses a standard set of suffixes for the style modifications. These suffixes are -Bold, -Italics, and -BoldItalics. The appropriate suffix is appended to the family name of the font, and this name is used to search the font directory table obtained from querying the printer. If a match is found, the document is printed using the corresponding PostScript font. If no match is found, then the driver tries to find the plain form of the font and creates the style modification in PostScript. A bitmap of the font is downloaded to the printer if these two searches fail.

If you are shipping your application with the intention of taking advantage of PostScript fonts when printing to a LaserWriter, please be sure to provide an Apple IIGS font whose family name is identical to the PostScript font family name.

LaserWriter	LaserWriter Plus and LaserWriter II
Courier	AvantGarde(R)
Helvetica(R)	Bookman(R)
Symbol	Courier
Times(R)	Helvetica
	Helvetica-Narrow
	NewCenturySchlbk
	Palatino(R)
	Symbol
	Times
	ZapfChancery(R)
	ZapfDingbats(R)

Table 2-Built-in LaserWriter Fonts

Further Reference

- Apple IIGS Toolbox Reference, Volumes 1 & 2
- Apple LaserWriter Reference

END OF FILE TN.IIGS.067

##
FILE: TN.IIGS.068
##

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#68: Tips for I/O Expansion Slot Card Design

Written by: Rob Moore & Jim Luther September 1989

This Technical Note points out several potential problem areas developers should know about when designing I/O expansion slot cards for the Apple IIGS.

This Note is written for experienced design engineers. It is not intended to be a tutorial on Apple IIGS I/O expansion card design techniques, but rather to point out possible problem areas and pitfalls to help developers produce successful and reliable expansion cards.

The 65C816 PH2 Clock versus the Expansion Slot PH0 Clock

It is important to understand the timing of the 65C816 Phase 2 clock (PH2) on the IIGS, because several of the expansion slot signals are actually related to the PH2 clock timing, rather than the 1 MHz Phase 0 clock (PH0) available at the expansion slots. Unlike the Apple IIe, the PH2 clock at the CPU is not the same as the PH0 clock found at the expansion slots. The PH2 clock runs at a variety of periods, depending on whether the CPU is doing a normal 350 nanosecond 2.8 MHz cycle, a extended 700 nanosecond RAM refresh cycle, an isolated slow cycle, or consecutive 980 nanosecond 1.024 MHz slow cycles. During isolated slow cycles, or the first of a series of consecutive slow cycles, the fast side of the system must wait to synchronize with the 1 MHz side of the system. This synchronization results in an average cycle time of about 1.5 microseconds.

Cycle Type	Low	High	Period
Normal 2.8-MHz cycle	140ns	210ns	350ns
Refresh extended cycle	140ns	560ns	700ns
Isolated 1-MHz cycle	140ns typ.	1.33 msecs avg.	≈1.5 msecs
Consecutive 1-MHz cycles	140ns	840(980)ns	980ns

Table 1-PH2 Clock Times

The Mega II Select Signal

On the Apple IIGS, the Mega II select signal (/M2SEL) is used as the enable to the slower, 1 MHz side of the system. It goes active (low) whenever the 1 MHz side RAM or I/O areas are accessed. Accesses that cause /M2SEL to be asserted include shadowed video writes, any accesses to internal I/O or expansion card slots, and accesses to banks $E0 and $E1. Accesses to any expansion card ROM areas that are set to Internal ROM with the Slot register do not assert the /M2SEL signal and run at the 2.8 MHz speed rather than the normal 1 MHz expansion card speed. Also, accesses to the Shadow register ($C035), CYA register ($C036), or DMA bank register ($C037), and reads from the Slot register ($C02D) or State Register ($C068) run at full speed since they are done wholly on the fast side of the system.

/M2SEL can be viewed as an extension of the address bus on the expansion slots. When it is active, it indicates that the CPU is running synchronized with the 1 MHz side of the system and the address on the address lines is a valid Apple II address in the 128K main or auxiliary memory space.

The Mega II Bank 0 Signal

The Mega II bank 0 signal (M2B0) provides the least significant bit of the CPU or DMA bank address to the 1 MHz side of the system. It is normally tri-stated and goes active for 140 nanoseconds, starting 140 nanoseconds after the PH0 clock falls. During the 140 nanosecond active period, M2B0 will be high whenever the CPU is accessing bank $E1 (with the exceptions noted previously) or doing a shadowed video write or I/O access in bank $01. Note that M2B0 does not reflect the state of the RAMRD, RAMWRT, ALTZP, 80STORE, or PAGE2 soft switches that allow access to the auxiliary 64K through bank $00. It only indicates accesses to bank $E1 or shadowed accesses through bank $01.

It is generally safe to latch the state of M2B0 by using the falling edge of the Q3 clock. Even though M2B0 will be tri-stated at the about the same time as Q3 falls, the turn-off and float time on M2B0 will generally provide sufficient hold time provided that there is not more than 1 LS TTL load on M2B0.

```
           ____                          ________________________
    PH0        \________________________/                        \_______


                _______________|         _______________          _______
    Q3     ____/               |________/               \________/
                               |
                               |
                         ______|_
    M2B0   _____________/      | \_______________________________________
                        \______|_/
                               |
                               |<- Latch state of M2B0 here
```

Figure 1-When to Latch State of M2B0

The Apple Video Overlay card uses M2B0 to detect writes to main and auxiliary RAM so that it can capture writes to the Apple IIGS video display buffers into its on-card display buffer. M2B0 is designed for this sort of thing and isn't of much use in most other applications. Note that M2B0 is only available on slot 3.

Using the Ready Signal

The Ready (RDY) input to the 65C816 is used to prevent a CPU cycle from completing until the expansion card has accepted the data output or has its input data available.

When the RDY input to a 65C02 or 6502 is held low, the processor continues to output the same address until RDY is released and the CPU completes the current cycle.

In the Apple IIGS, the 65C816 samples the RDY input when the PH2 clock goes low, and if RDY is low, the current CPU cycle does not complete and the address continues to be emitted. However, the bank address is not emitted while the clock is low if RDY is held low. To deal with this situation, the FPI (Fast Processor Interface) custom IC in the Apple IIGS uses a transparent latch to capture the bank address from the CPU. The latch is transparent while the PH2 clock is low and holds the bank address while the PH2 clock is high. If RDY is low, the CPU emits an invalid bank address, so the FPI holds the latch closed while RDY is low. This action is normally completely transparent to cards in the Apple IIGS expansion slots, but if an expansion card asserts RDY while the PH2 clock is low, it is likely to cause the FPI to latch an invalid bank address, because the latch could close before the bank address from the CPU is available on the data lines.

To avoid unpredictable results, RDY should only be asserted or deasserted when /M2SEL is low and when PH0 is high, or when /DEVSEL, /IOSEL or /IOSTRB are active. When /M2SEL, /DEVSEL, /IOSEL or /IOSTRB are active, you are guaranteed that the 65C816 is running at 1 MHz and is properly synchronized to the 1 MHz side of the system. RDY should be stable at least 60 nanoseconds before the falling edge of PH0 to allow for about a 25 nanosecond skew between the PH0 slot clock and the PH2 CPU clock. Figure 2 shows where it is safe to assert or deassert RDY. Limiting changes to RDY to the time when PH0 is high guarantees that it does not change while the CPU is outputting the bank address.

The RDY line should be driven with an open-collector driver.

```
                                                      ->|   |<- 35ns min
       ____       ______________________________________|___|
PH2        \_____/                                      |   |_________
                                                        |   |
                                                        | ->||<- 25ns min
       ____                         |___________________|____|
PH0        \________________________|                   |    |________
                                    |                   |    |
       _____________                |                   |    |  ______
/M2SEL              \_______________|___________________|____|_/
                                    |                   |    |
                                    |<----------------->|    |
                                       Safe to assert   |    |
                                        or deassert     |    |
/NMI,  ________________________________________________ |____|________
/IRQ,                                                  \|    |
/RST,  ________________________________________________/|____|________
RDY                                                     |    |
                                                      ->|    |<- 60ns min
```

Figure 2-Control Signal Setup Time

Interrupt Request, Non-Maskable Interrupt, and Reset

The Interrupt Request (/IRQ), the Non-Maskable Interrupt (/NMI) and the Reset (/RST) signals are all interrupt lines that are sampled by the CPU when the PH2 clock falls. If they are valid 30 nanoseconds before the PH2 clock falls, they are recognized on the following cycle. If this setup time is not met, they may not be recognized until the second following cycle . Since there can be up to a 25 nanosecond skew between the PH0 and PH2 clocks, these signals should be valid 60 nanoseconds before PH0 falls if they are to be recognized on the following cycle. Figure 2 shows the correct setup time for these signals.

All three signals are all active-low and must be driven with open-collector drivers.

Note: Interrupt vectors are always pulled from ROM regardless of whether or not the language card soft-switches have ROM enabled, providing that the I/O shadowing for banks $00/01 is enabled--which it always is when running Apple IIGS or Apple II system software.

Direct Memory Access

The Direct Memory Access (/DMA) signal is used to temporarily halt the CPU and allow expansion cards direct access to the system RAM to transfer data at high speeds. Since the 65C816 is fully static while the PH2 clock is high (unlike the 6502), /DMA may be asserted for as long as necessary on the Apple IIGS.

The /DMA signal should be asserted and deasserted within the 100 nanosecond period after PH0 falls, and the DMA address should be emitted by the expansion card about 30 nanoseconds later. In any case, the address should be stable on the address bus no later than 120 nanoseconds after PH0 falls. This guarantees that there is enough time for the address to be decoded and for /M2SEL and M2B0 to be asserted by the FPI chip if the DMA transfer is to the 1 MHz side of the system. The bank address must be stored in the DMA bank register at location $C037 before using DMA.

/DMA is a active-low signal and should be driven with an open-collector driver. The Apple IIGS provides a pullup for /DMA, but since the pullup is a fairly high value, it is a good idea for an expansion card that has asserted /DMA to momentarily pull it high for a few nanoseconds when deasserting it.

Note that there is a minor hardware bug in the Apple IIGS that could cause problems for developers who are unaware of it. If the CPU is currently pulling an interrupt vector when the /DMA signal is asserted, and if the DMA address is accessing the language card ($D000-$FFFF) space in a bank of memory where I/O and language card emulation is enabled (normally banks $00, $01, $E0 and $E1), DMA reads access ROM rather than RAM. This happens because the CPU's Vector Pull (VP) signal is active while the DMA cycle is active. Since most expansion cards that use DMA are also associated with some corresponding firmware or software driver, it's a good idea to disable interrupts prior to doing the DMA transfer, then re-enable interrupts as soon as possible after the transfer is complete. If interrupts are off too long, AppleTalk shuts down any connections to file servers because the system does not respond to AppleTalk "tickle" transactions while interrupts are disabled.

We recommend that the DMA be done with the Apple IIGS running at 1 MHz. If DMA is started during a 1 MHz cycle (/M2SEL asserted), the system continues to run slow while the /DMA signal is active.

Avoiding "Bus Fights"

The data bus on the Apple IIGS (and Apple IIe) expansion slots is a multiplexed bus that is used to carry both CPU and video display data. While PH0 is low, the bus is used to transfer data from the system RAM to the video display circuitry. When PH0 is high, the bus is available for CPU data transfers. To avoid potential (or actual) bus fights, it is helpful to avoid driving read data from an expansion card onto the bus immediately after PH0 rises. Since the video read data is driven out onto the expansion slots, and expansion card read data is driven in from the slots, it takes a finite period of time for the bus buffers to turn around. If a card drives data onto the expansion slot data bus immediately after PH0 rises, there may be a bus fight between the expansion card trying to drive the bus, and the Apple IIGS (or Apple IIe) bus buffers, which may not have turned around yet. A similar problem can occur if an expansion card leaves its read data on the bus too long after PH0 falls.

On the Apple IIGS, the data buffers turn around in 30 nanoseconds or less from the PH0 edges. Developers can avoid bus fights by simply using 74LS or 74HCT series parts and relying upon typical delay stackups to delay driving the data bus for approximately 30 nanoseconds. A more solid technique is using the first rising edge of the 7M clock, after PH0 rises. This method may require an additional flip-flop, but it guarantees the desired delay. On the other hand, expansion card read data buffers should be turned off as soon as possible when PH0 falls to avoid a fight when the data buffers turn back out again. Figure 3 shows the recommended data transfer timing for the data bus.

```
       __                            |___________________________|
PH0      \___________________________|                           |________
                                     |                           |
          ___     ___     ___     ___|   |___     ___     ___    |___
7M     __/   \___/   \___/   \___/   |___|   \___/   \___/   \___|   \___
                                     |   |                       |
                 Recommended Delay ->|   |<-                     |
       __   _________________________|_  |_______________________|| ______
D7-D0    \/        Video Data        | \_|  Data from I/O Card   ||/
       __/\__________________________|_/ |_______________________||\______
                                     |   |                       ||
                              0 to 30ns (as short as possible) ->||<-
```

Figure 3-Recommended Data Transfer Timing

Ground Noise

Since the Apple II expansion slots were designed with only one ground pin, complex expansion cards sometimes have problems with excessive ground noise--especially in the IIGS, where the signals typically have faster rise and fall times. To reduce ground noise as much as possible, it is helpful to bypass all four supply voltages (+5 volt, +12 volt, -5 volt, -12 volt) to ground with electrolytic or solid tantalum capacitors, even if all the available voltages are not used on the expansion card. This additional bypassing has the effect of providing an improved ground by providing additional AC ground paths through the various supply pins.

To maintain a consistent ground quality over the board area on two-layer boards, it is important to properly grid the Vcc and ground traces and to fill in unused areas with ground plane.

Expansion Card Power Consumption

The Apple IIe and Apple IIGS expansion slot specifications indicate a total of 500 mA of +5 volt, 250 mA of +12 volt, 200 mA of -5 volt, and 200 mA of -12 volt power is available to all the expansion slots. With design improvements, the power required by disk drives has been reduced. Also, the Apple IIGS power supply is conservatively designed so there is somewhat more power available than indicated on the original specification. However, there is not unlimited power available, and expansion card developers should minimize power consumption as much as possible. Minimization can be accomplished by using CMOS wherever possible, using ROMs or RAMs with "power-down" mode when they are not enabled, and generally being careful to minimize parts count.

Since the Apple IIGS was released, several "super" expansion cards have become available. These cards typically provide a lot of performance and functionality, but in most cases, the power consumed by one card is more than the specified power available to all the expansion slots. Generally these cards work without problems. However, when several "super" cards are installed in a IIGS system, the total power drawn can exceed the available power supply capacity. This increase in power dissipation within the IIGS case can cause excessive heating and other associated problems when the internal case temperatures exceed the design specifications. This could conceivably damage the IIGS power supply. Please minimize the power requirements of expansion card designs wherever possible to avoid these problems.

Further Reference

- Apple IIGS Hardware Reference
- Apple IIGS Firmware Reference
- Apple IIGS Technical Note #28, Interface Card Design Guidelines
- Apple IIGS Technical Note #32, /INH Line Anomaly

END OF FILE TN.IIGS.068

###
FILE: TN.IIGS.069
###

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#69: The Ins and Outs of Slot Arbitration

Written by: Matt Deatherage September 1989

This Technical Note discusses the concept of a 14-slot Apple IIGS system through dynamic software slot arbitration. It presents concepts of which all IIGS programmers should be aware for full compatibility.

History

The Apple II has always had seven slots. In some cases (e.g., IIe), one of the slots was handled specially by the hardware, or (e.g., IIc) there was no hardware present for peripheral cards at all. But there have always been seven "slots" with firmware at location $Cn00 (where n is the slot number). If there was no firmware, there was no peripheral connected.

With the introduction of the Apple IIGS, the Apple II family saw its first 14-slot system. Seven hardware slots are provided for peripheral cards (like on the IIe), and seven internal "ports" with connectors on the back panel are provided by the system (like on the IIc). Since $C800 and above cannot be used for additional slots (that space is shared between all interface cards), each of the seven internal ports is matched with one of the slots, and either the port or the slot is enabled at any given time. The IIGS hardware allows switching between the two, so all fourteen slots could be used more or less simultaneously.

This situation posed a problem--the Apple II had only a disk operating system, not an overall operating system. Access to non-disk devices (i.e., character devices, like a serial card) was not arbitrated by the system in any way. The world was used to seven, and only seven, slots. Attempting to use more in a shared system such as the IIGS resulted in somebody jumping to slot firmware that somebody else had switched out. This tended to crash the system.

Then came GS/OS. With its centralized mechanism for dispatching to all devices connected to a system, GS/OS provides hope (for the first time) that a central routing mechanism can dynamically arbitrate between slots and ports, allowing the use of all 14 at one time. This is called dynamic slot arbitration, and is handled by a portion of GS/OS referred to as the Slot Arbiter.

Although the Slot Arbiter does not function in System Software 5.0 or earlier, it may function in the future. A skeleton is present in version 5.0 and later that accepts Slot Arbiter calls, but the skeleton does not actually switch any slots. This Note details the Slot Arbiter functionality and shows how to switch slots under System Software 5.0 and later in a way which will not interfere with slot arbitration when it becomes available.

Note: The Slot Arbiter must not be used unless GS/OS is the current operating system.

The Slot Arbiter

The Slot Arbiter is accessed through the GS/OS system service call vector DYN_SLOT_ARBITER ($01FCBC). On ROM 03 and later, the vector is duplicated at $E10208. Entry to the Slot Arbiter is via a JSL instruction, and exit is via RTL. The parameters are as follows:

```
Entry:
    A = Slot to be selected (defined below)
    X = Undefined (or Bit Encoded Slot Configuration)
    Y = Undefined
    B = Undefined
    D = Undefined
    P = N V M X D I Z C  E
        x x 0 0 0 x x x  0

Exit:
    A = Error Code
    X = Bit Encoded Slot Configuration
    Y = Undefined
    B = Unchanged
    D = Undefined
    P = N V M X D I Z C  E
        x x 0 0 0 x x 0  0    If A = $0000 (no error)
        x x 0 0 0 x x 1  0    If A = $0010 (slot not available)
```

The slot number in the A register tells the Slot Arbiter what you are requesting. Bits 0-2 are the slot number in the range 0 through 7. Bit 3 is set if you are requesting an external slot and clear if you are requesting an internal port. Taken together, bits 0-3 give slot numbers of $0-$7 for internal ports and $9-$F for external slots. This is the same way that slot numbers are returned by the GS/OS DInfo command.

Bits 8 and 9 of the slot number indicate the action you wish the Slot Arbiter to take. A value in these two bits of 00 asks the Slot Arbiter to switch in the slot identified in bits 0 through 3. If both bits are set to 11, the Slot Arbiter restores all the slots to match the Bit Encoded Slot Configuration present in the X register. Bit Encoded Slot Configurations are discussed in the next section of this Note. Values other than 00 or 11 in bits 8 and 9 are reserved and must not be used by applications.

Bit 15 of the slot number is set if the slot selection has no slot dependencies. When the Slot Arbiter is asked to switch in a slot with no slot dependencies, it does no actual switching, although it returns a Bit Encoded Slot Configuration in the X register. The slot number and the definitions of the individual bits are illustrated in Figure 1.

```
     ___________________|___________
    |F|E|D|C|B|A|9|8|7|6|5|4|3|2|1|0|
    |_|__________|___|________|_|_____|
     |     |       |      |     |   |___ Slot
     |     |       |      |     |________ 0 = Internal; 1 = External
```

```
     |     |      |     |____________ Zero
     |     |      |__________________ Call Type Identifier
     |     |                            00 = Slot Request
     |     |                            11 = Select by Bit Encoded
     |     |                                 Slot Configuration
     |     |_________________________ Zero
     |_______________________________ Slot Dependent or Slot Independent
```

Figure 1-Slot Number and Bit Definitions

Bit Encoded Slot Configurations

Every call to the Slot Arbiter returns (on exit) a miniature picture of the slot configuration in the X register (as it was on entry). This picture has one bit set for each of the 14 slots; if the bit is set, then the corresponding slot is switched in. Bits 0 and 8 are reserved and are always clear. This picture is called a Bit Encoded Slot Configuration.

Since each external slot has the same number as an internal port (with bit 3 set), and since such pairs share the same address space, it follows that both of them may not be enabled at the same time. For example, port 5 and slot 5 ($D) both may not be enabled. This makes the high byte of the Bit Encoded Slot Configuration the eXclusive-OR of the low byte (excluding bits 0 and 8, which are always clear). Figure 2 illustrates the Bit Encoded Slot Configuration.

```
 _______________________________
|F|E|D|C|B|A|9|8|7|6|5|4|3|2|1|0|
 |_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_ 0
 | | | | | | | | | | | | | | |___ 1 = slot 1 active
 | | | | | | | | | | | | | |_____ 1 = slot 2 active
 | | | | | | | | | | | | |_______ 1 = slot 3 active
 | | | | | | | | | | | |_________ 1 = slot 4 active
 | | | | | | | | | | |___________ 1 = slot 5 active
 | | | | | | | | | |_____________ 1 = slot 6 active
 | | | | | | | | |_______________ 1 = slot 7 active
 | | | | | | | |_________________ 0
 | | | | | | |___________________ 1 = slot 9 active
 | | | | | |_____________________ 1 = slot 10 active
 | | | | |_______________________ 1 = slot 11 active
 | | | |_________________________ 1 = slot 12 active
 | | |___________________________ 1 = slot 13 active
 | |_____________________________ 1 = slot 14 active
 |_______________________________ 1 = slot 15 active
```

Figure 2-Bit Encoded Slot Configuration

By fully using the slot number parameter, the Slot Arbiter returns any aspect of the current slot configuration. Following are a few examples:

Slot number	Action Taken by Slot Arbiter
$8000	Returns current Bit Encoded Slot Configuration in the X register. This number asks the Slot Arbiter to switch in with no slot dependencies (no switching), so it just returns the Bit Encoded Slot Configuration.

$0300	Restore from Bit Encoded Slot Configuration. This command, when paired with the one above, can be used to save and restore a slot environment.
$0005	Asks the Slot Arbiter for internal port 5.

The Impact on Applications and Drivers

Applications which correctly do all input and output through GS/OS are affected by slot arbitration, except that they find more devices available. GS/OS uses the slot number parameter in the Device Information Block to call the Slot Arbiter, making sure the slot is available for the device before it gets control. However, there are some applications (such as peripheral card configuration programs) which go directly to firmware or hardware, not using GS/OS. Perhaps the card has no ROM, so there is no generated driver, or perhaps there is no loaded driver and the generated driver does not control certain aspects of the hardware. In any case, such applications are directly impacted by slot arbitration.

Slot Searching

The first problem is finding the hardware. In a 14-slot system, it's not suitable to just look for ID bytes between $C100 and $C700--two peripherals may be sharing each of those pages of slot ROM space. Drivers must examine all 14 slots, with the aid of the Slot Arbiter. The following sample code demonstrates this technique:

```
find_slot          lda    #$8000           ; request current Bit Encoded Slot
Configuration
                   jsl    slot_arbiter
                   phx                     ; save it on the stack

                   lda    #$000F           ; start with slot 15
                   sta    slot_number      ; be sure of the data bank when
                                           ; doing this!

slot_search        lda    slot_number      ; get the slot number to examine
                   jsl    slot_arbiter     ; and ask for it
                   bcs    continue_search  ; if an error, then don't look here
                   jsr    check_for_hw     ; this routine looks for your
hardware
                   bcc    found_my_hw      ; if found it, we're done searching
continue_search    dec    slot_number      ; try the next lower slot
                   bpl    slot_search      ; (if there are any left, of course)

found_my_hw        plx                     ; get Bit Encoded Slot Configuration
                                           ; from stack
                   lda    #$0300           ; and tell the Slot Arbiter to
                                           ; restore from it
                   jsl    slot_arbiter

; We're done.  Our slot number is in the location slot_number.
```

Note: You must restore the previous slot configuration when searching for a slot. This is vital to device drivers during the Drvr_Startup call, and failure to do so at other times may break older, seven-slot applications.

The Slot Arbiter attempts to maintain a static seven-slot system for applications as reflected by the user's Control Panel settings. This system allows older applications to continue to work, as something they find in an older, seven-slot scan is still present. Newer applications may wish to consider implementing a 14-slot scan, but any slot not present in the static seven-slot environment requires a call to the Slot Arbiter before and after every access to that device. The overhead in such instances may be intolerable. Apple recommends that if an application requires hardware that cannot be found in a seven-slot scan, it request the user to set the Control Panel to make the hardware available and restart the system.

Using Slot-Dependent Hardware

Applications which have slot dependencies must call the Slot Arbiter before each use of the slot in question. Since Slot Arbitration changes the environment to which Apple IIGS programs have become accustomed, everyone has a better chance of working by sticking to the general Apple IIGS rule of "put back what you use when you're done with it." Ask for the slot, use it, then restore the previous Bit Encoded Slot Configuration. (If you use multiple slots, you might wish to get the Bit Encoded Slot Configuration, save a copy, modify it to reflect the slots you want, and restore from the modified version.)

Note: Peripherals accessed through GS/OS do not have to call the Slot Arbiter; GS/OS handles this task automatically.

There are certain applications with more specialized needs, such as high-speed, single character input or output. In such cases, the Slot Arbiter may be a bottleneck. When a slot is not switched, the Slot Arbiter returns quickly, but when a slot must be switched, it takes a significant amount of time. Doubling that significant time for switching in and restoring gives a substantial overhead for each hardware access, which may be too much for some applications.

Note: It is far better to write a GS/OS driver to deal with hardware than to write a slot-dependent application to control it. A slot-dependent application must deal with the Slot Arbiter, and the user must quit the current application to run your application just to change some aspect of the hardware. Writing a GS/OS driver lets any application, desk accessory, or CDev control your hardware with regular GS/OS calls.

Problems with Slot-Dependent Tools

Code designed before the Slot Arbiter may have slot-dependencies that cause unexpected problems when dynamic slot arbitration is fully implemented. This list includes some of the Apple IIGS System Software. Specifically, the Text Tools and the FWEntry call in the Miscellaneous Tools present problems with dynamic slot arbitration.

Text Tools

When using the Text Tools to specify a device for input, output, or error, the value specified (a four-byte parameter) is assumed to be a slot number if it is in the range 0-7. The Text Tools were not designed to use Slot Arbiter-style slot numbers, and this causes a compatibility problem.

The Text Tools were modified in System Software 5.0 to recognize Slot Arbiter-style slot numbers where possible. The trick is that it's not possible as often as we'd like. External slots are specified by using slot numbers 9 through 15; if such a slot number is used as input to a Text Tools call, the appropriate Slot Arbiter call is made and that external slot is used if it can be made available. However, internal port numbers are in the range 1-7--the same range used by the old Text Tools to indicate which of two peripherals was switched in for a particular slot. The Text Tools cannot assume that you are requesting an internal slot when using a slot number between one and seven.

For example, your old application might do a seven-slot search and find a parallel printer card in slot 1 (where the Control Panel setting for that slot is "Your Card"). If the Text Tools assumed all slot numbers in the range one through seven meant internal ports, your application would actually access the internal port 1 firmware every time it tried to access the parallel card it found in slot 1; this problem occurs since old applications don't know and don't care about internal or external slots.

The Text Tools may be used to access any external slot (if available), but they may only be used to access internal ports that are set to internal in the Control Panel. The Text Tools slot numbers zero through seven always match the Control Panel settings.

Apple strongly recommends that the Text Tools not be used. GS/OS character-based drivers are preferable for standard character input and output. The Text Tools may be used for specialized purposes; however, you cannot access some internal ports and other components of the system that are not well-behaved. Doing so could cause your application to trash memory or media. You must assume these risks when using the Text Tools.

FWEntry

The Miscellaneous Tools call FWEntry should not be used to access entry points on a peripheral card (entry points in the $Cxxx range). As discussed, a poorly-behaved routine could switch the slot from one you've identified to something else between the time you identify the slot and issue the FWEntry call. Furthermore, the space between $C800 through $CFFF cannot be identified as belonging to any given slot, and the Slot Arbiter more or less guarantees that it won't be what you expect. Accesses to peripheral card ROM space ($Cxxx) should only be made by GS/OS drivers. FWEntry must not be used to access $Cxxx addresses.

FWEntry is still safe to use for addresses in the $D000-$FFFF range.

Further Reference

- Apple IIGS Toolbox Reference, Volume 2
- Apple IIGS Firmware Reference
- Apple IIGS Hardware Reference
- GS/OS Reference

END OF FILE TN.IIGS.069

\##
\### FILE: TN.IIGS.070
\##

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#70: Fast Graphics Hints

Written by: Don Marsh & Jim Luther September 1989

This Technical Note discusses techniques for fast animation on the Apple IIGS.

QuickDraw II gives programmers a very generalized way to draw something to the Super Hi-Res screen or to other parts of Apple IIGS memory. Unfortunately, the overhead in QuickDraw II makes it an unacceptable tool for all but simple animations. If you bypass QuickDraw II, your application has to write pixel data directly to the Super Hi-Res graphics display buffer. It also has to control the New-Video register at $C029, and set up the scan-line control bytes and color palettes in the graphics display buffer. Chapter 4 of the Apple IIGS Hardware Reference documents where you can find the graphics display buffer in memory and how the scan-line control bytes, color palettes, and pixel data bytes are used in Super Hi-Res graphics mode. The techniques described in this Note should be used with discretion--we do not recommend bypassing the Apple IIGS Toolbox unless it is absolutely necessary.

Map the Stack Onto Video Memory

To achieve the fastest screen updates possible, you must remove all unnecessary overhead from the instructions that perform graphics memory writes. The obvious method for achieving sequential writes to the graphics memory uses an index register, which must be incremented or decremented between writes. These operations can be avoided by using the stack. Each time a byte or word is pushed onto the stack, the stack pointer is automatically decremented by the appropriate amount. This is faster than doing an indexed store followed by a decrement instruction.

But how is the stack mapped onto the graphics memory? The stack can be located in bank $01 instead of bank $00 by writing to the WrCardRAM auxiliary-memory select switch at $C005. Bank $01 is shadowed into $E1 by clearing bit 3 of the Shadow register at $C035. Under these conditions, if the stack pointer is set to $3000, the next byte pushed onto the stack is written to $013000, then shadowed into $E13000. The stack pointer is automatically decremented so the stage is set for another byte to be written at $E12FFF.

Warning: While the stack is mapped into bank $01, you may not call any firmware, toolbox or operating system routines (ProDOS 8 or GS/OS). Don't even think about it.

Unroll All Loops

Another source of overhead is branching instructions in loops. By "straightlining" the code to move up a scan-line's worth of memory at one time, branch instructions are avoided. Following is an example of this technique.

```
    lda    |164,y              ; accumulator is 16 bits for
    pha                        ; best efficiency
    lda    |162,y
    pha
    lda    |160,y
    pha
```

In this example, the Y register is used to point to data to be moved to the graphics memory, and hard-coded offsets from the Y register are used to avoid register operations between writes.

Hard-Code Instructions and Data

In desperate circumstances, it is necessary to remove overhead from the previous code example. This can be accomplished by hard-coding pixel data into your code instead of loading pixel values from a separate data space and transferring them to the graphics memory (as in the example). If you are writing an arbitrary pattern of three or fewer constant values to the screen, for example, the following method is the fastest known:

```
    lda    #val1
    ldx    #val2
    ldy    #val3
    pha                        ; arbitrary pattern of pushes
    phx
    phy
    phy
    phx
```

In cases where many different values must be written to the screen, pixel data can be written to the screen using immediate push instructions:

```
    pea    $5389               ; some arbitrary pixel values
    pea    $2378
    pea    $A3C1
    pea    $39AF
```

Your program can generate this mixture of PEA instructions and pixel data itself, or it could load pixel data that already has PEA instructions intermixed (thus increasing the data size by one half).

Be Aware of Slow-Side and Fast-Side Synchronization

Estimating execution speed by counting instruction cycles is always a challenging task on the IIGS, but it is particularly tricky when one is writing to the graphics memory. The graphics memory resides in the side of the IIGS system controlled by the 1 MHz Mega II chip, which means that during all writes to this memory, the fast side of the system controlled by the Fast Processor Interface (FPI) chip must be synchronized with slow side of the system controlled by the Mega II, even if the system is running code at full native speed. This synchronization is performed automatically and transparently by the FPI in the IIGS, and it isn't normally of concern to the programmer. Animation programmers must worry about synchronization delays, however, because slight changes in graphics update code may change the frequency of these delays, and hence the speed of the program. In practical terms, this means that one loop writing data to the graphics memory may run at the same speed as a second loop with a higher cycle count.

A careful analysis of the synchronization problem leads to the following tables, which are useful as a rough estimate of the speed attained by different pieces of code. Each entry is based on the number of cycles consumed during consecutive write instructions. For example, a series of PEA instructions requires five cycles for each 16-bit write. A short PHA instruction followed by a branch requires six cycles for each 8-bit write.

Fast Cycles per Write (byte)	Actual Speed (microseconds/byte)
3 to 5	2.0
6 to 8	3.0
9 to 11	4.0

Fast Cycles per Write (word)	Actual Speed (microseconds/word)
4 to 6	3.0
7 to 8	4.0
9 to 11	5.0

The times given in the tables apply only if the same number of fast cycles separate each consecutive write operation. The first write operation in a set of write instructions usually takes longer than subsequent writes, because the potentially long synchronization operation is accomplished at that time. Unpredictable delays caused by memory refresh slow things down further, although refresh delays byte-wide writes more often than word-wide writes. Therefore, it is usually preferable from a speed standpoint to use word-wide writes to the graphics memory.

For more information on synchronization cycle timing within the IIGS, see Chapter 2 of the Apple IIGS Hardware Reference and Apple IIGS Technical Note #68, Tips for I/O Expansion Slot Card Design.

Use Change Lists

The timing data given in the preceding section shows that it is not possible to perform full-screen updates in the time it takes the IIGS to scan the entire screen. In fact, it would be difficult to update more than one-sixth of the screen in one scan time. Therefore, it is necessary to update only those pixels which have actually changed from the previous frame of animation. One method of doing this is to precalculate the pixels which change by comparing each frame against the preceding frame. For interactive animation, fast methods must be developed for predicting which areas of the screen must be updated (a determination of the exact pixels might require more computation than the actual update would require).

Using the Video Counters

To achieve "tear-free" screen updates, it is necessary to monitor the location of the scan-line beam when writing to graphics memory. As described in Apple IIGS Technical Note #39, Mega II Video Counters, the VertCnt and HorizCnt Mega II video counter registers at $C02E-C02F allow you to determine which scan line is currently being drawn.

By using only the VertCnt register and ignoring the low bit of the 9-bit vertical counter stored in HorizCnt, you can determine within 2 scan lines which scan line is currently being drawn. The VertCnt video counter contains the number of the current scan line divided by two, offset by $80. For example, if the scan-line beam was currently refreshing either scan line four or five, VertCnt would contain $82 (4/2 + $80 or 5/2 + $80). Vertical blanking happens during VertCnt values $7D through $7F and $E4 through $FF.

Clever updates can modify twice as many pixels on the screen by sacrificing some smoothness, running at 30 frames per second instead of 60. The technique is as follows:

1. Wait for the scan line beam to reach the first scan line.
2. Start updates from the top of the screen, being careful not to pass the scan line beam.
3. Continue updates while the scan line beam progresses toward the bottom of the screen, then goes into vertical blanking, then restarts at the top of the screen.
4. Finish the update before the scan line beam catches the update point.

Careful use of this method allows a frame to be updated during two scans of the screen instead of just one. If you are not sufficiently careful, tearing results.

Note: The Apple IIGS main logic board Mega II-VGC registers and interrupts are not synchronous to the Apple II Video Overlay Card video and therefore should not be used for time synchronization with the Apple II Video Overlay Card video output. However, they can be used for time synchronization with the Apple IIGS video output. See the Apple II Video Overlay Card Development Kit for more information.

Interrupts

It is not possible to support interrupts while sustaining a high graphics update rate, unless jerkiness or tearing is acceptable. Be aware that many system activities such as GS/OS and AppleTalk depend on interrupts and do not function if interrupts are disabled.

Further Reference

- Apple IIGS Firmware Reference
- Apple IIGS Hardware Reference
- Apple II Video Overlay Card Development Kit
- Apple IIGS Technical Note #39, Mega II Video Counters
- Apple IIGS Technical Note #40, VBL Signal
- Apple IIGS Technical Note #68, Tips for I/O Expansion Slot Card Design

END OF FILE TN.IIGS.070

```
#####################################################################
### FILE: TN.ImWr.001
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

ImageWriter
#1: Custom Font Selection

Revised by: Matt Deatherage November 1988
Written by: Rilla Reynolds October 1986

This Technical Note documents an ImageWriter II firmware bug which affects custom font selection.

Due to an ImageWriter II firmware bug, the ESC ' command neither selects nor reselects custom font 1 after custom font 2 is selected, unless you fix an errant pointer with the following command sequence first:

```
    7-bit mode:    ESC  Z  00  20   ESC  D  00  20   ESC  '
    8-bit mode:    ESC  Z  00  20   ESC  '
```

The ESC ' command works correctly on an ImageWriter I, but the sequence above is also acceptable; therefore, it is in your best interest to always utilize the given sequence to select custom font 1. It is possible that the printer was initialized and custom font 2 was selected long before your program was launched.

END OF FILE TN.ImWr.001

```
#####################################################################
### FILE: TN.MemX.001
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Memory Expansion Card
#1: Questions and Answers

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

This Technical Note documents many of the questions and answers concerning the Memory Expansion Card which are not covered in its manual.

Question: What screen holes does the Memory Expansion Card firmware use?
Answer: The Memory Expansion Card uses the following screen holes:

```
$478 + slot  numbanks number of 64K banks (256K = $04, 512 = $08)
$4F8 + slot  powerup  powerup byte ($A5)
$578 + slot  power2
```

These screen holes are not cast in concrete and may change with a new revision of the firmware.

Question: Why does RESET turn off the Memory Expansion Card registers until an access to the $Cn00 space?
Answer: The reason $Cn00 enables the registers was to optimize speed and the number of pins and logic on the custom gate array. The boot scan hits $Cn00 anyway and enables the registers.

Question: Will any access (read, write, or status) to the firmware cause the Memory Expansion Card to format itself?
Answer: Yes, any access to the firmware will cause it to format itself to the current operating system (DOS 3.3, Pascal, or ProDOS), assuming it is not already formatted.

Question: Why isn't the Memory Expansion Card marked as a non-interruptible device? What if an interrupt occurred during access to the card and the interrupt handler also accessed the card?
Answer: The Memory Expansion Card is not marked as a non-interruptible device because it would not be fatal to have an interrupt occur during an access to the device. Obviously, the interrupt handler would have to save and restore the registers as well as update the "free block" bitmap, so when the handler returns control the program does not overwrite the new data. The reason other devices are marked as non-interruptible is due to timing dependent read and write requirements.

Question: Why does the Memory Expansion Card fail to format if the powerup screen hole contains the value $A0?

Answer: The firmware checks the screen holes for $A0 values, and if they are all $A0, it assumes that someone made a mistake and cleared the screen improperly, filling the screen holes with spaces. In this case, the firmware does not want to reformat and lose all the files on the RAM disk.

Question: The code at $Cn5A has the following sequence, and does not seem to make sense:

```
LDA #$1
LDY $42
CMP #4
BCS Cn8E
```

Shouldn't the CMP #4 be a CPY #4?

Answer: Yes, this is a known bug that will be fixed if the ROMs are ever revised. The bug by itself was not considered significant enough to justify a revision. Note that this is corrected in the Memory Expandable Apple IIc.

Question: If DOS formats the Memory Expansion Card, ProDOS cannot reformat it without a power down or using a ProDOS application which formats disks. In other words, it does not reformat itself when I boot into a new operating system. Isn't that a bit severe?

Answer: This is no different than any other disk device. ProDOS does not have a format command, so you cannot just format from ProDOS without having the formatter installed and some means for calling it. Additionally this was done intentionally so that you could load DOS files into the RAM card and be able to boot ProDOS and use the CONVERT program to convert the DOS files to ProDOS.

END OF FILE TN.MemX.001

```
#####################################################################
### FILE: TN.Misc.001
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple II Miscellaneous
#1:   80-Column Screen Dump

Revised by: Pete McDonald   November 1988
Written by: Greg Seitz   December 1984

This Technical Note presents an example assembly language program which dumps the contents of the 80-column text screen to whatever is connected to COUT.

```
0000:                1 *
0000:                2 *  80-column screen dump
0000:                3 *
0000:                4 *  By
0000:                5 *     Greg Seitz
0000:                6 *     12-Jul-84
0000:                7 *
0000:                8 *  This program will allow you to dump the contents
0000:                9 *  of your 80-column text screen to whatever device is
0000:               10 *  connected through COUT.  If it is still connected to
0000:               11 *  the screen, you will obviously be printing back
0000:               12 *  what you were reading.
0000:               13 *
0000:        FBC1   14 BASCALC   EQU   $FBC1          ;convert A reg to line addr
on scrn
0000:        FDED   15 COUT      EQU   $FDED          ;A register out as ASCII
0000:        C001   16 SET80COL  EQU   $C001          ;enable page 1/2 switches to
control aux
0000:        C055   17 TXTPAGE2  EQU   $C055          ;page 2 or Aux depending
0000:        C054   18 TXTPAGE1  EQU   $C054          ;page 1 or main depending
0000:        0028   19 BASL      EQU   $28            ;BASCALC puts base addr. here
0000:        0029   20 BASH      EQU   $29            ;and high byte here.
0000:               21 *
1000:        1000   22           ORG   $1000          ;or anywhere
1000:        1000   23 SCREENDMP EQU   *
1000:A2 00          24           LDX   #0             ;START AT LINE 0
1002:               25 *
1002:8A             26 SCRNLP    TXA                  ;CALL BASCALC
1003:20 C1 FB       27           JSR   BASCALC        ;FOR ADDRESS OF LINE X
1006:A0 00          28           LDY   #00            ;DO 80 CHARS STARTING FROM
CHARACTER 0
1008:               29 *
1008:        1008   30 SCRNLP2   EQU   *
1008:8D 01 C0       31           STA   SET80COL       ;SET UP FOR MAIN/AUX
SWITCHING
100B:8D 55 C0       32           STA   TXTPAGE2       ;START ON AUX
```

```
100E:98              33          TYA                   ;GET CURRENT INDEX FOR DIVIDE
BY 2
100F:48              34          PHA                   ;SAVE ACTUAL COLUMN NUM WE'RE
ON
1010:4A              35          LSR                   ;COLUMN/2=ODD OR EVEN BRANCH
IF EVEN
1011:90 03   1016    36          BCC   SCRNDMP1        ;TAKEN IF EVEN SINCE STATE IS
PROPER
1013:8D 54 C0        37          STA   TXTPAGE1        ;ELSE IF ODD TURN ON MAIN MEM
1016:                38 *
1016:        1016    39 SCRNDMP1 EQU   *
1016:A8              40          TAY                   ;USE COLUMN/2 FOR INDEX NOW
1017:B1 28           41          LDA   (BASL),Y        ;GRAB THE CHARACTER
1019:8D 54 C0        42          STA   TXTPAGE1        ;SEL MAIN SO IT SEES RIGHT
SCREEN HOLES
101C:20 ED FD        43          JSR   COUT            ;PRINT THE CHARACTER
101F:68              44          PLA                   ;RECOVER COLUMN NUM
1020:A8              45          TAY                   ;INTO Y FOR NEXT TRIP
1021:C8              46          INY                   ;NEXT COLUMN NUM
1022:C0 50           47          CPY   #80             ;ANY MORE?
1024:90 E2   1008    48          BCC   SCRNLP2         ;TAKEN IF YES
1026:A9 8D           49          LDA   #$8D            ;ELSE CARRIAGE RETURN
1028:20 ED FD        50          JSR   COUT            ;OUT
102B:A9 8A           51          LDA   #$8A            ;LINE FEED
102D:20 ED FD        52          JSR   COUT            ;OUT
1030:E8              53          INX                   ;NEXT LINE
1031:E0 18           54          CPX   #24             ;ANYMORE?
1033:90 CD   1002    55          BCC   SCRNLP          ;TAKEN IF YES
1035:60              56          RTS

 FBC1 BASCALC       ?  29 BASH            28 BASL           FDED COUT
 C054 TXTPAGE1       C055 TXTPAGE2      ?1000 SCREENDMP      1016 SCRNDMP1
 1008 SCRNLP2        1002 SCRNLP         C001 SET80COL
** SUCCESSFUL ASSEMBLY := NO ERRORS
** ASSEMBLER CREATED ON 15-JAN-84 21:28
** TOTAL LINES ASSEMBLED    56
** FREE SPACE PAGE COUNT   84
```

END OF FILE TN.Misc.001

```
#####################################################################
### FILE: TN.Misc.002
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple II Miscellaneous
#2: Apple II Family Identification Routines 2.1

Revised by: Matt Deatherage & Keith Rollin — November 1988
Revised by: Pete McDonald — January 1986

This Technical Note presents a new version of the Apple II Family Identification Routine, a sample piece of code which shows how to identify various Apple II computers and their memory configurations.

Why Identification Routines?

Although we present the Apple II family identification bytes in Apple II Miscellaneous Technical Note #7, many people would prefer a routine they can simply plug into their own program and call. In addition, this routine serves as a small piece of sample code, and there is no reason for you to reinvent the wheel.

Most of the interesting part of the routine consists of identifying the memory configuration of the machine. On an Apple IIe, the routine moves code into the zero page to test for the presence of auxiliary memory. (A IIe with a non-extended 80-column card is a configuration still found in many schools throughout the country.)

The actual identification is done by a table-lookup method.

What the Routine Returns

This version (2.1) of the identification routine returns several things:

- A machine byte, containing one of seven values:
  $00 = Unknown machine
  $01 = Apple][
  $02 = Apple][+
  $03 = Apple /// in emulation mode
  $04 = Apple IIe
  $05 = Apple IIc

  In addition, if the high bit of the byte is set, the machine is a IIGS or equivalent. For all current Apple IIGS computers, the value returned in machine is $84 (high bit set to signify Apple IIGS and $04 because it matches the ID bytes of an enhanced Apple IIe).
- A ROMlevel byte, indicating the revision of the firmware in the

machine. For example, there are currently five revisions of the IIc, two of the IIe (unenhanced and enhanced), and two versions of the IIGS ROM (there will always be some owners who have not yet upgraded). These versions are identified starting at $01 for the earliest. Therefore, the current IIc will return ROMlevel = $04, the current IIGS will return ROMlevel = $02, etc. The routine will also return correct values for future versions of the IIGS, as a convention has been established for future ROM versions of that machine.
- A memory byte, containing the amount of memory in the machine. This byte only has four values--0 (undefined), 48, 64, and 128. Extra memory in an Apple IIGS, or extra memory in an Apple IIe or IIc Memory Expansion card, is not included. Programs must take special considerations to use that memory (if available), beyond those considerations required to use the normal 128K of today's IIe and IIc.
- If running on an Apple IIGS, three word-length fields are also returned. These are the contents of the registers as returned by the ID routine in the IIGS ROM, and they indicate several things about the machine. See Apple II Miscellaneous Technical Note #7 for more details.

In addition to these features, most of the addressing done in the routine is by label. If you wish things to be stored in different places, simply changing the labels will often do it.

Limitations and Improvements

As sample code, you might have already guessed that this is not the most compact, efficient way of identifying these machines. Some improvements you might incorporate if using these routines include:

- If you are running under ProDOS, you can remove the section that determines how much memory is in the machine (starting at exit, line 127), since the MACHID byte (at $BF98) in ProDOS already contains this information for you. This change would cut the routine down to less than one page of memory.
- If you know the ROM is switched in when you call the routine, you can remove the sections which save and restore the language card state. Be careful in doing so, however, because the memory-determination routines switch out the ROM to see if a language card exists.
- If you need to know if a IIe is a 64K machine with a non-extended 80-column card, you may put your own identifying routines in after line 284. NoAux is only reached if there is an 80-column card but only 64K of memory.

How It Works

The identification routine does the following things:

- Disables interrupts
- Saves four bytes from the language card areas so they may be restored later
- Identifies all machines by a table look-up procedure
- Calls 16-bit ID routine to distinguish IIGS from other machines of any

kind, and returns values in appropriate locations if IIGS ID routine returns any useful information in the registers

- Identifies memory configuration:
    - If Apple /// emulation, there is 48K
    - If Apple][or][+, tests for presence of language card and returns 64K if present, otherwise, returns 48K
    - If Apple IIc or IIGS, returns 128K
    - If Apple IIe, tries to identify auxiliary memory
        - If reading auxiliary memory, it must be there
        - If reading alternate zero page, auxiliary memory is present
        - If none of this is conclusive:
            - Exchanges a section of the zero page with a section of code that switches memory banks. The code executes in the zero page and does not get switched out when we attempt to switch in the auxiliary RAM.
            - Jumps to relocated code on page zero:
                - Switches in auxiliary memory for reading and writing
                - Stores a value at $800 and sees if the same value appears at $C00. If so, no auxiliary memory is present (the non-extended 80-column card has sparse memory mapping which causes $800 and $C00 to be the same location).
                - Changes value at $C00 and sees if the value at $800 changes as well. If so, no auxiliary memory. If not, then there is 128K available
                - Switches main memory back in for reading and writing
            - Puts the zero page back like we found it
        - Returns memory configuration found (either 64K or 128K)
- Restores language card and ROM state from four saved bytes
- Restores interrupt status
- Returns to caller

```
SOURCE   FILE #01 =>ID2.1

0000:                1          lst   on
----- NEXT OBJECT FILE NAME IS ID2.1.OBJ
2000:        2000    2          org   $2000
2000:                3 *******************************************
2000:                4 *                                         *
2000:                5 *  Apple II Family Identification Program  *
2000:                6 *                                         *
2000:                7 *               Version 2.1               *
2000:                8 *                                         *
2000:                9 *             September, 1988             *
2000:               10 *                                         *
2000:               11 *  Includes support for revisions to IIc   *
2000:               12 *  firmware, and IIgs identification too   *
2000:               13 *                                         *
2000:               14 *******************************************
2000:               15 *
2000:               16 *
2000:               17 *  First, some global equates for the routine:
2000:               18 *
2000:        0001   19 IIplain   equ   $01           ;Apple II
2000:        0002   20 IIplus    equ   $02           ;Apple II+
2000:        0003   21 IIIem     equ   $03           ;Apple /// in emulation mode
2000:        0004   22 IIe       equ   $04           ;Apple IIe
2000:        0005   23 IIc       equ   $05           ;Apple IIc
```

```
2000:                24 *
2000:        0001    25 safe      equ   $0001         ;start of code relocated to
zp
2000:        0006    26 location  equ   $06           ;zero page location to use
2000:                27 *
2000:        00FB    28 xce       equ   $FB           ;65816 XCE instruction
2000:                29 *
2000:        00AA    30 test1     equ   $AA           ;test byte #1
2000:        0055    31 test2     equ   $55           ;lsr of test1
2000:        0088    32 test3     equ   $88           ;test byte #3
2000:        00EE    33 test4     equ   $EE           ;test byte #4
2000:                34 *
2000:        0400    35 begpage1  equ   $400          ;beginning of text page 1
2000:        0800    36 begpage2  equ   $800          ;beginning of text page 2
2000:        0C00    37 begsprse  equ   $C00          ;byte after text page 2
2000:                38 *
2000:        C000    39 clr80col  equ   $C000         ;disable 80-column store
2000:        C001    40 set80col  equ   $C001         ;enable 80-column store
2000:        C002    41 rdmainram equ   $C002         ;read main ram
2000:        C003    42 rdcardram equ   $C003         ;read aux ram
2000:        C004    43 wrmainram equ   $C004         ;write main ram
2000:        C005    44 wrcardram equ   $C005         ;write aux ram
2000:        C013    45 rdramrd   equ   $C013         ;are we reading aux ram?
2000:        C016    46 rdaltzp   equ   $C016         ;are we reading aux zero
page?
2000:        C018    47 rd80col   equ   $C018         ;are we using 80-columns?
2000:        C01A    48 rdtext    equ   $C01A         ;read if text is displayed
2000:        C01C    49 rdpage2   equ   $C01C         ;read if page 2 is displayed
2000:        C050    50 txtclr    equ   $C050         ;switch in graphics
2000:        C051    51 txtset    equ   $C051         ;switch in text
2000:        C054    52 txtpage1  equ   $C054         ;switch in page 1
2000:        C055    53 txtpage2  equ   $C055         ;switch in page 2
2000:        C080    54 ramin     equ   $C080         ;read LC bank 2, write
protected
2000:        C081    55 romin     equ   $C081         ;read ROM, 2 reads write
enable LC
2000:        C08B    56 lcbank1   equ   $C08B         ;LC bank 1 enable
2000:                57 *
2000:        E000    58 lc1       equ   $E000         ;bytes to save for LC
2000:        D000    59 lc2       equ   $D000         ;save/restore routine
2000:        D400    60 lc3       equ   $D400
2000:        D800    61 lc4       equ   $D800
2000:                62 *
2000:        FE1F    63 idroutine equ   $FE1F         ;IIgs id routine
2000:                64 *
2000:                65 *  Start by saving the state of the language card banks and
2000:                66 *  by switching in main ROM.
2000:                67 *
2000:08              68 strt      php                 ;save the processor state
2001:78              69           sei                 ;before disabling interrupts
2002:AD 00 E0        70           lda   lc1           ;save four bytes from
2005:8D F1 21        71           sta   save          ;ROM/RAM area for later
2008:AD 00 D0        72           lda   lc2           ;restoring of RAM/ROM
200B:8D F2 21        73           sta   save+1        ;to original condition
200E:AD 00 D4        74           lda   lc3
2011:8D F3 21        75           sta   save+2
2014:AD 00 D8        76           lda   lc4
2017:8D F4 21        77           sta   save+3
```

```
201A:AD 81 C0                       lda   $C081          ;read ROM
201D:AD 81 C0                       lda   $C081
2020:A9 00                          lda   #0             ;start by assuming unknown
machine
2022:8D E8 21                       sta   machine
2025:8D E9 21                       sta   romlevel
2028:                   *
2028:A5 06              IdStart     lda   location       ;save zero page locations
202A:8D F5 21                       sta   save+4         ;for later restoration
202D:A5 07                          lda   location+1
202F:8D F6 21                       sta   save+5
2032:A9 FB                          lda   #$FB           ;all ID bytes are in page $FB
2034:85 07                          sta   location+1     ;save in zero page as high
byte
2036:A2 00                          ldx   #0             ;init pointer to start of ID
table
2038:BD F7 21           loop        lda   IDTable,x      ;get the machine we are
testing for
203B:8D E8 21                       sta   machine        ;and save it
203E:BD F8 21                       lda   IDTable+1,x    ;get the ROM level we are
testing for
2041:8D E9 21                       sta   romlevel       ;and save it
2044:0D E8 21                       ora   machine        ;are both zero?
2047:F0 1C   2065                   beq   matched        ;yes - at end of list - leave
2049:                   *
2049:E8                 loop2       inx                  ;bump index to loc/byte pair
to test
204A:E8                             inx
204B:BD F7 21                       lda   IDTable,x      ;get the byte that should be
in ROM
204E:F0 15   2065                   beq   matched        ;if zero, we're at end of
list
2050:85 06                          sta   location       ;save in zero page
2052:                   *
2052:A0 00                          ldy   #0             ;init index for indirect
addressing
2054:BD F8 21                       lda   IDTable+1,x    ;get the byte that should be
in ROM
2057:D1 06                          cmp   (Location),y   ;is it there?
2059:F0 EE   2049                   beq   loop2          ;yes, so keep on looping
205B:                   *
205B:E8                 loop3       inx                  ;we didn't match.Scoot to the
end of the
205C:E8                             inx                  ;line in the ID table so we
can start
205D:BD F7 21                       lda   IDTable,x      ;checking for another machine
2060:D0 F9   205B                   bne   loop3
2062:E8                             inx                   ;point to start of next line
2063:D0 D3   2038                   bne   loop           ;should always be taken
2065:                   *
2065:        2065       matched     equ   *
2065:                   *
2065:                   *  Here we check the 16-bit ID routine at $FE1F.  If it
2065:                   *  returns with carry clear, we call it again in 16-bit
2065:                   *  mode to provide more information on the machine.
2065:                   *
2065:38                 idIIgs      sec                  ;set the carry bit
2066:20 1F FE                       jsr   idroutine      ;Apple IIgs ID Routine
```

```
2069:90 03   206E  124            bcc   idIIgs2       ;it's a IIgs or equivalent
206B:4C A2 20      125            jmp   exit          ;nope, go check memory
206E:AD E8 21      126 idIIgs2    lda   machine       ;get the value for machine
2071:09 80         127            ora   #$80          ;and set the high bit
2073:8D E8 21      128            sta   machine       ;put it back
2076:18            129            clc                 ;get ready to switch into
native mode
2077:FB            130            dfb   xce           ;this is a 65816 XCE
instruction
2078:08            131            php                 ;save the processor status
2079:C2 30         132            dfb   $C2,$30       ;REP 30, sets 16-bit
registers
207B:20 1F FE      133            jsr   $FE1f         ;call the ID routine again
207E:8D EB 21      134            sta   IIgsA         ;16-bit store!
2081:8E ED 21      135            stx   IIgsX         ;16-bit store!
2084:8C EF 21      136            sty   IIgsY         ;16-bit store!
2087:28            137            plp                 ;restores 8-bit registers
2088:FB            138            dfb   xce           ;switches back to whatever it
was before
2089:              139 *
2089:AC EF 21      140            ldy   IIgsY         ;get the ROM vers number
(starts at 0)
208C:C0 02         141            cpy   #$02          ;is it ROM 01 or 00?
208E:B0 01   2091  142            bcs   idIIgs3       ;if not, don't increment
2090:C8            143            iny                 ;bump it up for romlevel
2091:8C E9 21      144 idIIgs3    sty   romlevel      ;and put it there
2094:C0 01         145            cpy   #$01          ;is it the first ROM?
2096:D0 0A   20A2  146            bne   IIgsOut       ;no, go on with things
2098:AD F0 21      147            lda   IIgsY+1       ;check the other byte too
209B:D0 05   20A2  148            bne   IIgsOut       ;nope, it's a IIgs successor
209D:A9 7F         149            lda   #$7F          ;fix faulty ROM 00 on the
IIgs
209F:8D EB 21      150            sta   IIgsA
20A2:        20A2  151 IIgsOut    equ   *
20A2:              152 *
20A2:              153 ******************************************
20A2:              154 * This part of the code checks for the   *
20A2:              155 * memory configuration of the machine.   *
20A2:              156 * If it's a IIgs, we've already stored   *
20A2:              157 * the total memory from above.  If it's  *
20A2:              158 * a IIc, we know it's 128K; if it's a    *
20A2:              159 * ][+, we know it's at least 48K and     *
20A2:              160 * maybe 64K.  We won't check for less    *
20A2:              161 * than 48K, since that's a really rare   *
20A2:              162 * circumstance.                          *
20A2:              163 ******************************************
20A2:              164 *
20A2:AD E8 21      165 exit       lda   machine       ;get the machine kind
20A5:30 14   20BB  166            bmi   exit128       ;it's a 16-bit machine (has
128K)
20A7:C9 05         167            cmp   #IIc          ;is it a IIc?
20A9:F0 10   20BB  168            beq   exit128       ;yup, it's got 128K
20AB:C9 04         169            cmp   #IIe          ;is it a IIe?
20AD:D0 03   20B2  170            bne   contexit      ;yes, go muck with aux memory
20AF:4C 4E 21      171            jmp   muckaux
20B2:C9 03         172 contexit   cmp   #IIIem        ;is it a /// in emulation?
20B4:D0 6E   2124  173            bne   exitII        ;nope, it's a ][ or ][+
20B6:A9 30         174            lda   #48           ;/// emulation has 48K
```

```
20B8:4C BD 20                   jmp   exita
20BB:A9 80          exit128     lda   #128          ;128K
20BD:8D EA 21       exita       sta   memory
20C0:AD 00 E0       exit1       lda   lc1           ;time to restore the LC
20C3:CD F1 21                   cmp   save          ;if all 4 bytes are the same
20C6:D0 18   20E0               bne   exit2         ;then LC was never on so
20C8:AD 00 D0                   lda   lc2           ;do nothing
20CB:CD F2 21                   cmp   save+1
20CE:D0 10   20E0               bne   exit2
20D0:AD 00 D4                   lda   lc3
20D3:CD F3 21                   cmp   save+2
20D6:D0 08   20E0               bne   exit2
20D8:AD 00 D8                   lda   lc4
20DB:CD F4 21                   cmp   save+3
20DE:F0 38   2118               beq   exit6
20E0:AD 88 C0       exit2       lda   $C088         ;no match! so turn first LC
20E3:AD 00 E0                   lda   lc1           ;bank on and check
20E6:CD F1 21                   cmp   save
20E9:F0 06   20F1               beq   exit3
20EB:AD 80 C0                   lda   $C080
20EE:4C 18 21                   jmp   exit6
20F1:AD 00 D0       exit3       lda   lc2
20F4:CD F2 21                   cmp   save+1        ;if all locations check
20F7:F0 06   20FF               beq   exit4         ;then do more more else
20F9:AD 80 C0                   lda   $C080         ;turn on bank 2
20FC:4C 18 21                   jmp   exit6
20FF:AD 00 D4       exit4       lda   lc3           ;check second byte in bank 1
2102:CD F3 21                   cmp   save+2
2105:F0 06   210D               beq   exit5
2107:AD 80 C0                   lda   $C080         ;select bank 2
210A:4C 18 21                   jmp   exit6
210D:AD 00 D8       exit5       lda   lc4           ;check third byte in bank 1
2110:CD F4 21                   cmp   save+3
2113:F0 03   2118               beq   exit6
2115:AD 80 C0                   lda   $C080         ;select bank 2
2118:28             exit6       plp                 ;restore interrupt status
2119:AD F5 21                   lda   save+4        ;put zero page back
211C:85 06                      sta   location
211E:AD F6 21                   lda   save+5        ;like we found it
2121:85 07                      sta   location+1
2123:60                         rts                 ;and go home.
2124:               *
2124:AD 8B C0       exitII      lda   lcbank1       ;force in language card
2127:AD 8B C0                   lda   lcbank1       ;bank 1
212A:AE 00 D0                   ldx   lc2           ;save the byte there
212D:A9 AA                      lda   #test1        ;use this as a test byte
212F:8D 00 D0                   sta   lc2
2132:4D 00 D0                   eor   lc2           ;if the same, should return
zero
2135:D0 12   2149               bne   noLC
2137:4E 00 D0                   lsr   lc2           ;check twice just to be sure
213A:A9 55                      lda   #test2        ;this is the shifted value
213C:4D 00 D0                   eor   lc2           ;here's the second check
213F:D0 08   2149               bne   noLC
2141:8E 00 D0                   stx   lc2           ;put it back!
2144:A9 40                      lda   #64           ;there's 64K here
2146:4C BD 20                   jmp   exita
2149:A9 30          noLC        lda   #48           ;no restore - no LC!
```

```
214B:4C BD 20                 jmp   exita          ;and get out of here
214E:                   *
214E:AE 1A C0           muckaux   ldx   rdtext         ;remember graphics in X
2151:AD 1C C0                     lda   rdpage2        ;remember current video
display
2154:0A                           asl   A              ;in the carry bit
2155:A9 88                        lda   #test3         ;another test character
2157:2C 18 C0                     bit   rd80col        ;remember video mode in N
215A:8D 01 C0                     sta   set80col       ;enable 80-column store
215D:08                           php                  ;save N and C flags
215E:8D 55 C0                     sta   txtpage2       ;set page two
2161:8D 51 C0                     sta   txtset         ;set text
2164:AC 00 04                     ldy   begpage1       ;save first character
2167:8D 00 04                     sta   begpage1       ;and replace it with test
character
216A:AD 00 04                     lda   begpage1       ;get it back
216D:8C 00 04                     sty   begpage1       ;and put back what was there
2170:28                           plp
2171:B0 08   217B                 bcs   muck2          ;stay in page 2
2173:8D 54 C0                     sta   txtpage1       ;restore page 1
2176:30 03   217B       muck1     bmi   muck2          ;stay in 80-columns
2178:8D 00 C0                     sta   $c000          ;turn off 80-columns
217B:A8                 muck2     tay                  ;save returned character
217C:8A                           txa                  ;get graphics/text setting
217D:30 03   2182                 bmi   muck3
217F:8D 50 C0                     sta   txtclr         ;turn graphics back on
2182:C0 88              muck3     cpy   #test3         ;finally compare it
2184:D0 2F   21B5                 bne   nocard         ;no 80-column card!
2186:AD 13 C0                     lda   rdramrd        ;is aux memory being read?
2189:30 2F   21BA                 bmi   muck128        ;yup, there's 128K!
218B:AD 16 C0                     lda   rdaltzp        ;is aux zero page used?
218E:30 2A   21BA                 bmi   muck128        ;yup!
2190:A0 2A                        ldy   #done-start
2192:BE BC 21           move      ldx   start-1,y      ;swap section of zero page
2195:B9 00 00                     lda   safe-1,y       ;code needings safe location
during
2198:96 00                        stx   safe-1,y       ;reading of aux mem
219A:99 BC 21                     sta   start-1,Y
219D:88                           dey
219E:D0 F2   2192                 bne   move
21A0:4C 01 00                     jmp   safe           ;jump to safe ground
21A3:08                 back      php                  ;save status
21A4:A0 2A                        ldy   #done-start    ;move zero page back
21A6:B9 BC 21           move2     lda   start-1,y
21A9:99 00 00                     sta   safe-1,y
21AC:88                           dey
21AD:D0 F7   21A6                 bne   move2
21AF:68                           pla
21B0:B0 03   21B5                 bcs   noaux
21B2:4C BA 21           isaux     jmp   muck128        ;there is 128K
21B5:                   *
21B5:                   *  You can put your own routine at "noaux" if you wish to
21B5:                   *  distinguish between 64K without an 80-column card and
21B5:                   *  64K with an 80-column card.
21B5:                   *
21B5:        21B5       noaux     equ   *
21B5:A9 40              nocard    lda   #64            ;only 64K
21B7:4C BD 20                     jmp   exita
```

```
21BA:4C BB 20      287 muck128   jmp   exit128       ;there's 128K
21BD:              288 *
21BD:              289 *  This is the routine run in the safe area not affected
21BD:              290 *  by bank-switching the main and aux RAM.
21BD:              291 *
21BD:A9 EE         292 start     lda   #test4        ;yet another test byte
21BF:8D 05 C0      293           sta   wrcardram     ;write to aux while on main
zero page
21C2:8D 03 C0      294           sta   rdcardram     ;read aux ram as well
21C5:8D 00 08      295           sta   begpage2      ;check for sparse memory
mapping
21C8:AD 00 0C      296           lda   begsprse      ;if sparse, these will be the
same
21CB:C9 EE         297           cmp   #test4        ;value since they're 1K apart
21CD:D0 0E   21DD  298           bne   auxmem        ;yup, there's 128K!
21CF:0E 00 0C      299           asl   begsprse      ;may have been lucky so we'll
21D2:AD 00 08      300           lda   begpage2      ;change the value and see
what happens
21D5:CD 00 0C      301           cmp   begsprse
21D8:D0 03   21DD  302           bne   auxmem
21DA:38            303           sec                 ;oops, no auxiliary memory
21DB:B0 01   21DE  304           bcs   goback
21DD:18            305 auxmem    clc
21DE:8D 04 C0      306 goback    sta   wrmainram     ;write main RAM
21E1:8D 02 C0      307           sta   rdmainram     ;read main RAM
21E4:4C A3 21      308           jmp   back          ;continue with program in
main mem
21E7:EA            309 done      nop                 ;end of relocated program
marker
21E8:              310 *
21E8:              311 *
21E8:              312 *  The storage locations for the returned machine ID:
21E8:              313 *
21E8:00            314 machine   dfb   $00           ;the type of Apple II
21E9:00            315 romlevel  dfb   $00           ;which revision of the
machine
21EA:00            316 memory    dfb   $00           ;how much memory (up to 128K)
21EB:00 00         317 IIgsA     dw    $0000         ;16-bit field
21ED:00 00         318 IIgsX     dw    $0000         ;16-bit field
21EF:00 00         319 IIgsY     dw    $0000         ;16-bit field
21F1:00 00 00 00   320 save      dfb   0,0,0,0,0,0   ;six bytes for saved data
21F7:01 01 B3 38   321 IDTable   dfb   1,1,$B3,$38,$00 ;Apple ][
21FC:02 01 B3 EA   322           dfb   2,1,$B3,$EA,$1E,$AD,$00 ;Apple ][+
2203:03 01 B3 EA   323           dfb   3,1,$B3,$EA,$1E,$8A,$00 ;Apple ///
(emulation)
220A:04 01 B3 06   324           dfb   4,1,$B3,$06,$C0,$EA,$00 ;Apple IIe
(original)
2211:04 02 B3 06   325           dfb   4,2,$B3,$06,$C0,$E0,$00 ;Apple IIe
(enhanced)
2218:05 01 B3 06   326           dfb   5,1,$B3,$06,$C0,$00,$BF,$FF,$00 ;Apple IIc
(original)
2221:05 02 B3 06   327           dfb   5,2,$B3,$06,$C0,$00,$BF,$00,$00 ;Apple IIc
(3.5 ROM)
222A:05 03 B3 06   328           dfb   5,3,$B3,$06,$C0,$00,$BF,$03,$00 ;Apple IIc
(Mem. Exp)
2233:05 04 B3 06   329           dfb   5,4,$B3,$06,$C0,$00,$BF,$04,$00 ;Apple IIc
(Rev. Mem.
Exp.)
```

```
223C:05 05 B3 06   330           dfb   5,5,$B3,$06,$C0,$00,$BF,$05,$00 ;Apple IIc
Plus
2245:00 00         331           dfb   0,0           ;end of table

 21DD AUXMEM         21A3 BACK           0400 BEGPAGE1       0800 BEGPAGE2
 0C00 BEGSPRSE      ?C000 CLR80COL       20B2 CONTEXIT       21E7 DONE
 20E0 EXIT2          20A2 EXIT           20BB EXIT128       ?20C0 EXIT1
 20F1 EXIT3          20FF EXIT4          210D EXIT5          2118 EXIT6
 20BD EXITA          2124 EXITII         21DE GOBACK        ?2065 IDIIGS
 206E IDIIGS2        2091 IDIIGS3        FE1F IDROUTINE     ?2028 IDSTART
 21F7 IDTABLE          05 IIC              04 IIE            21EB IIGSA
 20A2 IIGSOUT        21ED IIGSX          21EF IIGSY            03 IIIEM
?  01 IIPLAIN       ?  02 IIPLUS        ?21B2 ISAUX          E000 LC1
 D000 LC2            D400 LC3            D800 LC4            C08B LCBANK1
   06 LOCATION       2038 LOOP           2049 LOOP2          205B LOOP3
 21E8 MACHINE        2065 MATCHED        21EA MEMORY         21A6 MOVE2
 2192 MOVE           21BA MUCK128       ?2176 MUCK1          217B MUCK2
 2182 MUCK3          214E MUCKAUX        21B5 NOAUX          21B5 NOCARD
 2149 NOLC          ?C080 RAMIN          C018 RD80COL        C016 RDALTZP
 C003 RDCARDRAM      C002 RDMAINRAM      C01C RDPAGE2        C013 RDRAMRD
 C01A RDTEXT        ?C081 ROMIN          21E9 ROMLEVEL         01 SAFE
 21F1 SAVE           C001 SET80COL       21BD START         ?2000 STRT
   AA TEST1            55 TEST2            88 TEST3            EE TEST4
 C050 TXTCLR         C054 TXTPAGE1       C055 TXTPAGE2       C051 TXTSET
 C005 WRCARDRAM      C004 WRMAINRAM        FB XCE
** SUCCESSFUL ASSEMBLY := NO ERRORS
** ASSEMBLER CREATED ON 15-JAN-84 21:28
** TOTAL LINES ASSEMBLED   331
** FREE SPACE PAGE COUNT   81
```

Further Reference

o   Apple II Miscellaneous Technical Note #7, Apple II Family Identification

END OF FILE TN.Misc.002

##
FILE: TN.Misc.003
##

Apple II
Technical Notes

Developer Technical Support

Apple II Miscellaneous
#3: Super Serial Card Firmware Bug

Revised by: Matt Deatherage November 1988
Written by: Cameron Birse November 1985

This Technical Note documents two bugs in the Super Serial Card firmware.

The Super Serial Card (SSC) firmware does not access location $CFFF to clear the $C800 space before jumping into its bank-switched ROM in that area.

By omitting this access, the Super Serial Card can cause a slot data bus conflict when a ROM of equal or greater strength on another card "owns" the $C800 space when the Super Serial Card wants to use it. For example, the UniDisk 3.5 controller card uses the same 74LS245 octal bus driver as the Super Serial Card. If you are using the UniDisk 3.5 card and switch to the Super Serial Card firmware, there will be a bus conflict . The SSC is trying to switch in its own $C800 space while the UniDisk 3.5 card is trying to keep the $C800 space, since no one cleared it by accessing $CFFF. Since both have the same capability to drive the bus, neither wins the battle.

An easy solution to this problem is to reference $CFFF before calling any of the Pascal entry points on the Super Serial Card. For example:

```
NEWSLOT     STA    $CFFF          ;reset the slot ROM space
            LDA    Char           ;Char = character to output
            LDX    #$Cn           ;n = slot number
            LDY    #$n0
            STX    MSLOT          ;MSLOT = $7F8, always set it up
            JSR    PWRITE         ;now call the Pascal routine of your choice
```

This bug is in the Pascal entry points; the BASIC entry point does not have this problem as there is a STA $CFFF instruction at $Cn1B.

This example code stores the slot number (in the form $Cn) in MSLOT, a screen hole used to tell the system which peripheral card had control when an interrupt occurred. The Super Serial Card firmware does set up MSLOT, but does not do so until long after it has enabled its $C800 space. If an IRQ comes through the system between the call to the card and when the card stores MSLOT, the system will crash.

Both bugs can be avoided by using the sample code to call entry points on the Super Serial Card.

Further Reference

o   Apple IIe Technical Reference Manual

END OF FILE TN.Misc.003

##
FILE: TN.Misc.004
##

Apple II
Technical Notes

Developer Technical Support

Apple II Miscellaneous
#4:   AppleWorks Keys

Revised by: Matt Deatherage   May 1989
Written by: J.D. Eisenberg   June 1985

This Technical Note formerly described information concerning AppleWorks(TM), which is now published by CLARIS.
Changes since November 1988: Updated the CLARIS mailing address.

This Note formerly discussed sections of AppleWorks 1.2 and 1.3 code which checked for keypresses to allow other applications to tap into certain routines. For information on AppleWorks, contact CLARIS at:

CLARIS
5201 Patrick Henry Drive
P.O. Box 58168
Santa Clara, CA 95052-8168

Technical Information
Telephone: (415) 962-0371
AppleLink: Claris.Tech

Non-Technical Information
Telephone: (415) 962-8946
AppleLink: Claris.CR

In addition to the support available from CLARIS, Bob Lissner, the author of AppleWorks, maintains a bulletin board for AppleWorks-related information. You can obtain technical information and file formats from this system as well as submit your comments in writing. You can reach this system at (702) 831-1722.

END OF FILE TN.Misc.004

##
FILE: TN.Misc.005
##

Apple II
Technical Notes

Developer Technical Support

Apple II Miscellaneous
#5:   AppleWorks File Formats

Revised by: Matt Deatherage   May 1989
Revised by: Matt Deatherage   November 1988

This Technical Note formerly documented the file formats for AppleWorks(TM), which is now published by CLARIS.
Changes since November 1988: Updated the CLARIS mailing address.

This Note formerly documented the file formats available in AppleWorks and /// E-Z Pieces (AppleWorks for the Apple ///). This information is now documented in three File Type Notes:

AppleWorks Data Base	$19
AppleWorks Word processor	$1A
AppleWorks Spreadsheet	$1B

For additional information on AppleWorks, contact CLARIS at:

CLARIS
5201 Patrick Henry Drive
P.O. Box 58168
Santa Clara, CA 95052-8168

Technical Information
Telephone: (415) 962-0371
AppleLink: Claris.Tech

Non-Technical Information
Telephone: (415) 962-8946
AppleLink: Claris.CR

In addition to the support available from CLARIS, Bob Lissner, the author of AppleWorks, maintains a bulletin board for AppleWorks-related information. You can obtain technical information and file formats from this system as well as submit your comments in writing. You can reach this system at (702) 831-1722.

Further Reference

- o   Apple II File Type Note $19
- o   Apple II File Type Note $1A
- o   Apple II File Type Note $1B

END OF FILE TN.Misc.005

```
#####################################################################
### FILE: TN.Misc.006
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple II Miscellaneous
#6: IWM Port Description

Revised by: Glenn A. Baxter November 1988
Written by: Cameron Birse February 1986

This Technical Note documents the IWM port pin assignments on various machines.

Apple IIGS Disk Port Pin Assignments

Signal Name	Disk Port Pins (DB-19)
Phase 0	11
Phase 1	12
Phase 2	13
Phase 3	14
/WReq	15
Dr1	17
Rd	18
Wr	19
Wrt Prot	10
Dr2	9
HeadSel	16
Gnd	1,2,3
3.5Disk	4
-12v	5
+5v	6
+12v	7,8

Apple IIe UniDisk 3.5 Controller Disk Port Pin Assignments

Signal Name	Disk Port Pins (DB-19)
Phase 0	11
Phase 1	12
Phase 2	13
Phase 3	14
/WrtReqII	15
/HstEnbl	17
Rd	18
Wr	19
Wrt Prot	10
No Connection	9,16
Gnd	1,2,3,4
-12v	5
+5v	6
+12v	7,8

Apple IIc Disk Port Pin Assignments

Signal Name	Disk Port Pins (DB-19)
Phase 0	11
Phase 1	12
Phase 2	13
Phase 3	14
/WrtReq	15
/Enbl 2	17
Rd	18
Wr	19
Wrt Prot	10
No Connection	16
Gnd	1,2,3,4
-12v	5
+5v	6
+12v	7,8
External Interrupt	9

Note: On the Apple IIc Plus, the disk port pins are driven by a custom ASIC instead of by the IWM chip.

Macintosh Disk Port Pin Assignments

Signal Name	Disk Port Pins (DB-19)
Phase 0	11
Phase 1	12
Phase 2	13
Phase 3	14
/WrtReq	15
/Enbl 2	17
Rd	18
Wr	19
PWM	10
HdSel	16
GND	1,2,3,4
-12v	5
+5v	6
+12v	7,8

END OF FILE TN.Misc.006

\###
\### FILE: TN.Misc.007
\###

Apple II
Technical Notes

Developer Technical Support

Apple II Miscellaneous
#7: Apple II Family Identification

Revised by: Matt Deatherage November 1988
Written by: Cameron Birse December 1986

This Technical Note describes the ROM identification bytes in the Apple II family.

To identify which computer of the Apple II family is executing your program, you must check the following identification bytes. These bytes are in the main bank of main ROM (shadowed on the Apple IIGS), and you should make sure that this bank is switched in before making decisions based on the contents of these locations.

Machine	$FBB3	$FB1E	$FBC0	$FBBF
Apple][	$38		[$60]	[$2F]
Apple][+	$EA	$AD	[$EA]	[$EA]
Apple /// (emulation)	$EA	$8A		
Apple IIe	$06		$EA	[$C1]
Apple IIe (enhanced)	$06		$E0	[$00]
Apple IIc	$06		$00	$FF
Apple IIc (3.5 ROM)	$06		$00	$00
Apple IIc (Org. Mem. Exp.)	$06		$00	$03
Apple IIc (Rev. Mem. Exp.)	$06		$00	$04
Apple IIc Plus	$06		$00	$05
Apple IIGS (See below)				

Note: Values listed in square brackets in the table are provided for your reference only. You do not need to check them to conclusively identify an Apple II.

The ID bytes for an Apple IIGS are not listed in the table since they match those of an enhanced Apple IIe. Future 16-bit Apple II products may match different Apple II identification bytes for compatibility reasons, so to identify a machine as a IIGS or other 16-bit Apple II, you must make the following ROM call:

```
    SEC                         ;Set carry bit (flag)
    JSR $FE1F                   ;Call to the monitor
    BCS OLDMACHINE              ;If carry is still set, then old machine
    BCC NEWMACHINE              ;If carry is clear, then new machine
```

In all the current, standard Apple II ROMs, $FE1F contains an RTS. In the Apple IIGS, there is a routine that returns compatibility information in the A, X, and Y registers:

Bit	Accumulator	X Register	Y Register
Bit 15	Reserved	Reserved	Machine ID Number (0 = Apple IIGS)
Bit 14	Reserved	Reserved	Machine ID Number
Bit 13	Reserved	Reserved	Machine ID Number
Bit 12	Reserved	Reserved	Machine ID Number
Bit 11	Reserved	Reserved	Machine ID Number
Bit 10	Reserved	Reserved	Machine ID Number
Bit 9	Reserved	Reserved	Machine ID Number
Bit 8	Reserved	Reserved	Machine ID Number
Bit 7	Reserved	Reserved	ROM version number
Bit 6	1 if system has memory expansion slot	Reserved	ROM version number
Bit 5	1 if system has IWM port	Reserved	ROM version number
Bit 4	1 if system has a built-in clock	Reserved	ROM version number
Bit 3	1 if system has desktop bus	Reserved	ROM version number
Bit 2	1 if system has SCC built-in	Reserved	ROM version number
Bit 1	1 if system has external slots	Reserved	ROM version number
Bit 0	1 if system has internal ports	Reserved	ROM version number

Note: In emulation or eight-bit mode, only the lower eight bits are returned.

This ROM call is enough to determine if a machine is an Apple IIGS or equivalent.

Note: The original Apple IIGS ROM returns a faulty value in the accumulator. The value returned is $xx1F and should be $xx7F. If you see a $0000 in the Y register (i.e., Apple IIGS, ROM version $00), you should assume that the accumulator value is $xx7F.

The current Apple IIGS ROM (ROM version $01) sets all the registers correctly before returning from this call.

Further Reference

- Miscellaneous Technical Note #2,
  Apple II Family Identification Routines 2.1

END OF FILE TN.Misc.007

##
FILE: TN.Misc.008
##

Apple II
Technical Notes

Developer Technical Support

Apple II Miscellaneous
#8: Pascal 1.1 Firmware Protocol ID Bytes

Revised by: Matt Deatherage November 1988
Written by: Cameron Birse December 1986

This Technical Note documents the Pascal 1.1 Firmware Protocol ID bytes for Apple II peripheral cards and ports.

Background

Apple II Pascal 1.1 introduced a firmware protocol called, not surprisingly, the Pascal 1.1 Firmware Protocol. A card following this protocol could be identified by the following ID bytes, where n is the slot in which the card resides:

Address	Value	Definition
$Cn05	$38	ID byte (from Pascal 1.0)
$Cn07	$18	ID byte (from Pascal 1.0)
$Cn0B	$01	Generic signature of cards with Pascal 1.1 Protocol
$Cn0C	$ci	Device signature byte

$Cn0C was interpreted as two nibbles. The high-order nibble, c, was defined as the device signature. This signature was a pre-defined value determining what kind of device was connected (i.e., printer, modem, joystick, clock, etc.). The low-order nibble, i, was defined as a unique identifier, so you could tell one printer from another, for example.

Developer Technical Support no longer maintains a list of assignments for the i nibble in this protocol. Since, by definition, the Pascal 1.1 Protocol only has room for 16 uniquely identified devices of each signature, it is easy to see that the Apple II family has outgrown the definition.

Following is a table which lists the values of the Pascal 1.1 Firmware Protocol ID bytes for some Apple products which follow the protocol. Previous versions of this Note listed ID bytes for products which did not follow the protocol. Do not attempt to identify devices which do not follow the protocol by checking these ID bytes. This method will not work and should be avoided.

For example, trying to conclusively identify a 3.5" disk drive, SCSI hard drive, memory expansion card, or other SmartPort device using these ID bytes could be disastrous. For any SmartPort device, you should look for the ProDOS Block Device ID bytes ($Cn01 = $20, $Cn03 = $00, $Cn05 = $03), then look for the additional SmartPort ID byte ($Cn07 = $00). Once you have identified

SmartPort, you should make a SmartPort STATUS call to determine the nature and types of connected devices. By this definition, ProDOS block devices and SmartPort devices cannot follow the Pascal 1.1 Firmware Protocol.

Pascal 1.1 Devices

	$Cn05	$Cn07	$Cn0B	$Cn0C
Apple II Peripheral Cards				
Super Serial Card (or port)	$38	$18	$01	$31
Apple 80 Column Card	$38	$18	$01	$88
Apple II Mouse Card	$38	$18	$01	$20
Apple IIc Ports				
1st version $FBBF = $FF				
Slot 1 (Serial Port)	$38	$18	$01	$31
Slot 2 (Serial Port)	$38	$18	$01	$31
Slot 3 (80 Columns)	$38	$18	$01	$88
Slot 4 (Mouse)	$38	$18	$01	$20
2nd version $FBBF = $00				
Slot 1 (Serial Port)	$38	$18	$01	$31
Slot 2 (Serial Port)	$38	$18	$01	$31
Slot 3 (80 Columns)	$38	$18	$01	$88
Slot 4 (Mouse)	$38	$18	$01	$20
Slot 7 (AppleTalk)	$38	$18	$01	$31
3rd version $FBBF = $03, 4th version $FBBF = $04, and 5th version $FBBF = $05				
Slot 1 (Serial Port)	$38	$18	$01	$31
Slot 2 (Serial Port)	$38	$18	$01	$31
Slot 3 (80 Columns)	$38	$18	$01	$88
Slot 7 (Mouse)	$38	$18	$01	$20
Apple IIGS Ports (ROM 1.0 and 2.0)				
Slot 1 (Serial Port)	$38	$18	$01	$31
Slot 2 (Serial Port)	$38	$18	$01	$31
Slot 3 (80 Columns)	$38	$18	$01	$88
Slot 4 (Mouse Port)	$38	$18	$01	$20
Slot 7 (AppleTalk)	$38	$18	$01	$31

ProDOS and SmartPort Devices

	$Cn01	$Cn03	$Cn05	$Cn07
Generic ProDOS Block Device	$20	$00	$03	$xx
SmartPort Device	$20	$00	$03	$00

END OF FILE TN.Misc.008

###
FILE: TN.Misc.009
###

Apple II
Technical Notes

Developer Technical Support

Apple II Miscellaneous
#9: AppleSoft Real Variable Storage

Revised by: Pete McDonald November 1988
Written by: Cameron Birse December 1986

This Technical Note discusses real variable storage in AppleSoft BASIC.

In AppleSoft BASIC, real variables (non-array) are stored sequentially starting at the address pointed to by locations $69 and $6A. The first two bytes are the name of the variable, the third is the exponent, and the fourth through seventh are the mantissa.

Exponent — The top bit of this byte is the sign of the exponent. This sign bit is the opposite of normal sign bits, since zero is negative and one is positive. The remainder of the byte minus one is the value of the exponent (i.e., 84 is a positive exponent of 3).

Mantissa — The mantissa is a binary fraction with the first bit being the sign bit (normal this time with zero being positive and one negative), and the remaining bits are fractional values starting with .5, .25, .125, etc.

The equation which follows is: 2^(Exponent-1) * 1.Mantissa

Examples

A = 3 (real variable equal to 3)

```
The seven bytes look like:    41    00                  Variable name = A
                              82                        Exponent = 1
                              40    00    00    00      Mantissa = .5
```

which works out as: 2^1 * 1.5 = 3

B = 5 (real variable equal to 5)

```
The seven bytes look like:    42    00                  Variable name = B
                              83                        Exponent = 2
                              20    00    00    00      Mantissa = .25
```

which works out as: 2^2 * 1.25 = 5

Further Reference

o   AppleSoft BASIC Programmer's Reference Manual

END OF FILE TN.Misc.009

```
#####################################################################
### FILE: TN.Misc.010
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple II Miscellaneous
#10: 80-Column GetChar Routine

Revised by: Dave Lyons — September 1989
Written by: Cameron Birse — December 1986

This Technical Note presents an 80-column GetChar routine.
Changes since November 1988: Added discussion of single-character input on the unenhanced Apple IIe.

The following is an example of how to display a string on the 80-column screen, reposition the cursor at the beginning of the string, and use the right arrow to get characters which are already there or accept new characters in their place. The routine is a simple BASIC program which displays the string and repositions the cursor before getting incoming characters. If the character input is a right arrow, the program calls the assembly language routine to get the character from screen memory at the current cursor location.

```
10  PRINT  CHR$ (4);"bload getchar.0": REM  first install assembly routine
20  B$ = "hello"
30  PRINT  CHR$ (4);"pr#3"
40  PRINT B$;:B$ = ""
50  A =  PEEK (1403): REM  get horiz location
60  A = A - 5: REM  move cursor to beginning of string
70  POKE 1403,A
80  GET A$: REM  get a character
90  IF A$ =  CHR$ (21) THEN  GOSUB 130: REM   if char is forward arrow,
    handle with assembly routine (GETCHAR)
100  IF A$ =  CHR$ (27) THEN 170: REM   if esc key then we're done
110  PRINT A$;:B$ = B$ + A$
120  GOTO 80
130  CALL 768: REM   GETCHAR
140  A =  PEEK (6)
150  A$ =  CHR$ (A)
160  RETURN
170  PRINT : PRINT : PRINT B$: REM  and we're done
```

An assembled listing of the assembly language GetChar routine follows. It works on the Apple IIe and later.

```
SOURCE   FILE #01 =>GETCHAR
----- NEXT OBJECT FILE NAME IS GETCHAR.0
0300:        0300    1           ORG   $300
0300:        C01F    2 RD80VID   EQU   $C01F         ;80 COLUMN STATE
0300:        C054    3 TXTPAGE1  EQU   $C054         ;TURN OFF PAGE 2 (READ)
```

```
0300:         C055    4 TXTPAGE2  EQU   $C055         ;TURN ON PAGE 2 (READ)
0300:         C000    5 CLR80COL  EQU   $C000         ;TURN OFF 80 STORE (WRITE)
0300:         C001    6 SET80COL  EQU   $C001         ;TURN ON 80 STORE (WRITE)
0300:         0028    7 BASL      EQU   $28           ;BASE ADDRESS OF SCREEN
LOCATION
0300:         0029    8 BASH      EQU   $29
0300:         057B    9 OURCH     EQU   $57B          ;80 COLUMNS HORIZ. POSITION
0300:         05FB   10 OURCV     equ   $5fb          ;80 col vertical pos
0300:         0006   11 char      equ   6             ;place to hand character back
to basic
0300:                12 *
0300:                13
*************************************************************
0300:                14 *   GETCHAR - This routine gets an ascii character from the
*
0300:                15 *   80 column display memory of the Apple IIe. It assumes
*
0300:                16 *   that main memory is switched in and that the base addrs
*
0300:                17 *   of the line has already been calculated and resides
*
0300:                18 *   in BASL and BASH. It is meant to be called from BASIC
*
0300:                19 *   as follows:
*
0300:                20 *                     CALL 768
*
0300:                21 *                     A = PEEK (6)
*
0300:                22 *                     A$ = CHR$(A)
*
0300:                23 *   As you can see, the character is returned in location
*
0300:                24 *   $6 in zero page. This routine is offered as an example.
*
0300:                25 *   No guaranties are made regarding its fitness for any
*
0300:                26 *   purpose.          By Cameron Birse 6/10/86
*
0300:                27
*************************************************************
0300:                28 *
0300:         0300   29 getchr    equ   *             ;get the char at the current
cursor loc.
0300:A9 01           30           lda   #$01          ;mask for horiz test
0302:2C 7B 05        31           bit   OURCH         ;are we in main or aux mem?
0305:D0 17   031E    32           bne   main          ;if bit 0 of OURCH is set,
then main mem
0307:         0307   33 aux       equ   *
0307:AD 7B 05        34           lda   OURCH         ;get horiz pos.
030A:18              35           clc                 ;clear the carry for divide
030B:6A              36           ror   a             ;divide by two
030C:A8              37           tay                 ;put the result in y
030D:8D 01 C0        38           sta   SET80COL      ;turn on 80 store
0310:AD 55 C0        39           lda   TXTPAGE2      ;flip to aux text page
0313:B1 28           40           lda   (basl),y      ;get the character
0315:85 06           41           sta   char
0317:AE 54 C0        42           ldx   TXTPAGE1      ;turn off aux text page
```

```
031A:8D 00 C0                     sta   CLR80COL       ;turn off 80 store
031D:60                           rts
031E:        031E     main        equ   *
031E:AD 7B 05                     lda   OURCH          ;get horiz pos.
0321:18                           clc                  ;clear the carry for divide
0322:6A                           ror   a              ;divide by two
0323:A8                           tay                  ;put the result in y
0324:B1 28                        lda   (basl),y       ;get the character
0326:85 06                        sta   char
0328:60                           rts
```

Reading a Single Character

While the 80-column firmware is active (whether in 40- or 80-column mode), the RDKEY routine on the unenhanced Apple IIe unexpectedly allows the user to press ESC and move the cursor around the screen the same way RDCHAR does.

AppleSoft's GET statement uses RDKEY, so it behaves the same way. The ESC keypress is never returned, so users have problems if you use GET and expect them, for example, to press ESC to return to the previous menu. At this point, the cursor turns into an inverse plus sign (+) and your program is still waiting for a keypress. The user presses ESC a few more times, watching the cursor alternate between an inverse plus sign and an inverse blank, and then turns off the computer in search of a more exciting activity, like throwing darts at your disk.

If your program can run on the unenhanced IIe, either leave the 80-column firmware turned off (PRINT CHR$(21) to make sure it's off), or read keypresses by polling the keyboard register directly:

```
    1000 IF PEEK(-16384)<128 THEN 1000     : REM Wait for a keypress
    1010 A$ = CHR$(PEEK(-16384)-128)       : REM Read the key
    1020 POKE -16368,0                     : REM Clear the keyboard strobe
```

or

```
    0300: LDA $C000                ; check for a keypress
    0303: BPL $0300                ;    keep waiting
    0306: AND #$7F                 ; turn off bit 7
    0308: STA $C010                ; clear the keyboard strobe
```

Note that these code fragments don't display a cursor while waiting for a key.

Further Reference

- Apple IIGS Firmware Reference
- Apple IIe Technical Reference Manual
- Apple IIc Technical Reference Manual

END OF FILE TN.Misc.010

##
FILE: TN.Misc.011
##

Apple II
Technical Notes

Developer Technical Support

Apple II Miscellaneous
#11: Examining the $C800 Space from AppleSoft

Revised by: Matt Deatherage May 1989
Written by: John Bennett August 1987

This Technical Note discusses examining the $C800 space from AppleSoft BASIC with PEEK statements.
Changed since January 1989: Corrected the revision author name.

Both the 6502 and 65816 microprocessors perform a false read during absolute-indexed instructions. When AppleSoft interprets a PEEK statement, it performs an absolute-indexed LDA instruction with a base address such that a false read from $CFxx is performed. This read takes place during the formula translation of the expression passed to PEEK, not during the actual loading of the value.

Some peripheral cards have been designed to deselect their $C800 ROM space any time a $CF value is placed on the high-order address lines of the address bus. Therefore, if you use the AppleSoft PEEK statement to examine an address in the $C800 space of such a peripheral card, the $C800 space will be turned off when the statement is interpreted, and the value returned by the statement will not reflect the actual value in the $C800 ROM.

The 65C02, on the other hand, has been designed so that a false read is not performed for an absolute-indexed LDA instruction. As a result, if the PEEK statement is used to examine the $C800 space of the same peripheral card on an enhanced Apple IIe (or any other Apple II with a 65C02 installed), the $C800 space will not be deselected, and the value returned by the statement will accurately reflect the value in the $C800 ROM.

If it is absolutely necessary to examine the $C800 space from an AppleSoft BASIC program, it is safer to use a assembly-language routine to examine the addresses and pass the results to the BASIC application.

END OF FILE TN.Misc.011

###
FILE: TN.Misc.012
###

Apple II
Technical Notes

Developer Technical Support

Apple II Miscellaneous
#12: The Apple II Firmware WAIT Routine

Revised by: Matt Deatherage — November 1988
Written by: Matt Deatherage — May 1988

This Technical Note expands on the already documented descriptions of the Apple II firmware WAIT routine, which guaranteed a minimum, not an exact, specified delay.

As described in the Apple IIe Technical Reference Manual and the Apple IIc Technical Reference Manual, the WAIT routine located in ROM at $FCA8 waits for a certain amount of time before returning to the calling program. The delay is listed in the IIe manual as being 1/2(26+27A+5A^2), where A is the value in the accumulator when WAIT is called. The value returned by this expression is the number of clock cycles taken by the routine, not the amount of time that passes while it waits. To obtain the elapsed time in microseconds, you must multiply the result by the scaling factor 14 / 14.318181.

Different formulas have appeared in different firmware listings published by Apple in the past, but the above formula is in all current publications, and has been verified as correct by Developer Technical Support. If there were nothing in the system except a 65C02 (or 65816) microprocessor, this formula would be completely accurate. However, this is not the case in an Apple II, as there are interrupts, changing system speeds, fast and slow RAM, and numerous other additions to the system that can cause extra overhead when a routine is executed.

For these reasons, the WAIT routine should be used only as a minimum delay. It should not be expected to wait for exactly the time specified by the WAIT formula.

The Apple IIGS Firmware Reference correctly notes this fact, as well as including the scaling factor (14 / 14.318181) to return the minimum delay in microseconds without further calculation.

Further Reference
- Apple IIGS Firmware Reference
- Apple IIe Technical Reference Manual
- Apple IIc Technical Reference Manual

END OF FILE TN.Misc.012

###
FILE: TN.Misc.014
###

Apple II
Technical Notes

Developer Technical Support

Apple II Miscellaneous
#14: Guidelines for Telecommunication Programs

Written by: Matt Deatherage July 1989

This Technical Note discusses recommended guidelines to ensure future compatibility and maintain workable standards for telecommunication programs.

Telecommunication programs have always been a particularly troublesome area on the Apple II as far as standards are concerned. Exiting from terminal programs often leaves the system in an unbalanced state or leaves strange and unknown things upon the user's disks. Yet complying with standards would not only make life easier for the users, it's not that hard for developers to do. This Note lists the primary guidelines Apple II telecommunication program developers should keep foremost in their minds.

Talking to the Hardware

Communicating with the modem through the interface provided by the user isn't always the easiest task in the world. It often just can't be done at acceptable speeds when using high-level software routines, and sometimes it can't even be done at the firmware level. It's widely known that the Super Serial Card can't keep up with 9600 bps communication unless a low-level driver uses the 6551 chip on the card directly--the firmware just can't do it. The Apple IIGS serial port firmware can easily keep up with 9600 bps, but the GS/OS generated character drivers for those ports can't do single character I/O at that speed.

In general, programs must use the highest level interface available to them that functions to specifications. If dealing with speeds of less than 9600 baud in 16-bit mode, on the Apple IIGS, use the GS/OS drivers. Remember that any GS/OS driver owns the slot or port it controls, and going around the drivers causes problems. High-speed, highly-configurable loaded drivers for the serial ports may ship with the System Software in the future, and it would be unfortunate if your terminal program was the one that caused the driver to break.

For speeds of 9600 bps or higher with System Software 5.0, the driver can't help you. It is necessary to go directly to the firmware or hardware and risk of future incompatibility. Remember that the firmware must be called from bank zero emulation mode. If single character I/O isn't necessary, the driver can handle speeds of 9600 bps when used in multicharacter input or output.

Note: In the future, System Software may include loaded drivers for the serial ports. An application can tell whether a driver is generated or loaded by examining bit 14 of the characteristics word returned by the GS/OS DInfo call--a generated driver has this bit set. A loaded driver may be able to handle 9600 bps single-character I/O, but a generated one may not.

File Transfer Considerations

Transferring files is probably the most important function of a telecommunication program. However, transferring the file's data itself is not always adequate. Telecommunication programs must find a way to transfer a file's attributes as well as a file's contents to keep things running smoothly.

File attributes include the file's type and auxiliary type (necessary fields for most applications to identify their data files), the size of the file, creation and modification dates and times, as well as information about how many forks the file has, what file system it came from, and how the file is stored on disk. In addition, when asked, GS/OS returns in its option_list information about the file that the native file system uses but GS/OS does not (information such as access privileges, native file types and creator types, parent directory IDs, extended attribute records and other information as important to the native file system as file type and auxiliary type are to GS/OS).

Any telecommunication program can devise a way to keep such attributes with a file when the file is transferred between two machines that are both running the program in question. It is a much trickier task to address the issue of keeping all file attributes with files regardless of the programs involved in the transfer. An industry-wide standard is necessary for such integration.

The Binary II standard, devised by Gary B. Little (and documented in the Apple II File Type Note for File Type $E0, Auxiliary Type $8000), has been accepted as a standard for maintaining these attributes for a number of years. Many major telecommunication programs already incorporate support for this standard; Apple urges those that don't to consider doing so at their earliest convenience.

Binary II is designed to keep attributes with files on the fly--it is not an archival standard and should not be used as such. A standard like Binary II should always be used to keep attributes with a file; confusing it with an archival standard can result in files being transferred without their own attributes. Even archival files must be transferred with their attributes. It is never acceptable to transfer a file without it's attributes.

Archival considerations are a completely separate issue. An archival format and program must be carefully designed with archiving considerations in mind, such as manipulating files within the archive, preserving the attributes of the files archived, and allowing for a myriad of compression schemes. The NuFX standard (documented in the Apple II File Type Note for File Type $E0, Auxiliary Type $8002) is such an archival format, which Apple recommends be used for those purposes. The program ShrinkIt is an example of a NuFX archival utility.

In an ideal world, all files would be transferred with their attributes sent transparently by the telecommunication program. The user would select the file to send, and the program would automatically send the attributes. When the program receives a file, it would already have the attributes with the file, so no postprocessing by the user would be necessary to use the file.

Even archival files such as NuFX should be transferred with all attributes intact. Although the archival utility may allow the user to select any file for processing (in case the file's attributes were lost), assuming that this will happen implies that it's acceptable. It is not. No file should ever be transferred without all its attributes, down to, and including the GS/OS option_list, if present.

Apple IIGS Considerations

A few more guidelines for Apple IIGS-specific telecommunication applications follow:

- o Don't ignore slot configurations. Attempting to use a serial port through hardware while an interface card for that slot is switched in will break dynamic slot arbitration if, and when, it becomes available, unless the application does not use the firmware.

- o Be a good neighbor to interrupt handlers. Interrupts will be coming through that you did not enable. (This is also true for Apple IIe computers with Workstation Cards, but is true for IIGS computers even when AppleTalk is not involved.) Programs not prepared for this could bring the system down. Stealing main interrupt vectors is not a good idea.

- o Don't go stepping on things you don't own. It is better to alert the user that a certain resource (like a slot or a port) is not available than to blindly switch it in and crash the system. Never switch slots without using the Slot Arbiter.

- o Behave yourself. Don't make wild assumptions or do things differently just because you're a terminal program and you think you have to do it for speed. Most users won't be impressed by a terminal program that's fast and robust if it breaks every time they activate a desk accessory or if they have to reboot the system when they're done with it. Don't compromise system integrity for superficial functionality.

Further Reference

- o Apple IIGS Firmware Reference
- o Apple IIGS Hardware Reference
- o Apple II File Type Notes, File Type $E0, Auxiliary Type $8000
- o Apple II File Type Notes, File Type $E0, Auxiliary Type $8002

END OF FILE TN.Misc.014

##
FILE: TN.Mous.001
##

Apple II
Technical Notes

Developer Technical Support

Mouse
#1: Interrupt Environment with the Mouse

Revised by: Matt Deatherage November 1988
Revised by: Rilla Reynolds November 1985

This Technical Note describes the interrupt environment one should take into account when programming mouse-based applications on the Apple II family of computers.

Software developers who are writing mouse-based programs in assembly language need to be concerned about the computer's interrupt environment, even if they are using the mouse in passive mode. Listed below are several conditions which assembly language programmers should take into account if their programs are to run on the Apple II family of computers.

- Do not disable interrupts unless absolutely necessary. If you disable them, be sure to re-enable them.
- Disable interrupts when calling any mouse routine. Always use PHP and SEI to disable interrupts, then use PLP to re-enable them. This method preserves the state of interrupts (enabled or disabled).
- Do not re-enable interrupts (PLP) after a call to ReadMouse until X and Y data have been removed from the screen holes.
- Disable interrupts (PHP and SEI) before placing position information in the screen holes (PosMouse or ClampMouse).
- Enter all mouse routines (except ServeMouse) with the X register set to $Cn and Y register set to $n0, where n = the slot number.
- Some programs need to disable interrupts for purposes other than reading the mouse. If interrupts are disabled then re-enabled, the first call to ReadMouse could return incorrect values; subsequent calls to ReadMouse will return correct values until interrupts are disabled and re-enabled again. Disabling interrupts for mouse calls does not create this problem. If you watch numbers from the mouse while moving it in a direction which would increase values, you would see something similar to: 6, 7, 8, 9, 8, 9, 10. In practice, this momentary "glitch" in the stream of data has little importance. If you feel you must avoid this glitch altogether, do not disable interrupts for more than 40 microseconds or make sure that at least one mouse interrupt takes place after re-enabling interrupts.

END OF FILE TN.Mous.001

##
FILE: TN.Mous.002
##

Apple II
Technical Notes

Developer Technical Support

Mouse
#2: Varying VBL Interrupt Rate

Revised by: Matt Deatherage November 1988
Revised by: Rilla Reynolds November 1985

This Technical Note describes a method to make the AppleMouse peripheral card interrupt at a rate other than the default 60 Hz. This method does not work on the Apple IIc or IIGS.

This Technical Note describes a previously undocumented call to the AppleMouse II firmware which allows the user to set the interrupt rate to 50 or 60 Hz. (The default is 60 Hz, which keeps the card-generated VBL interrupts synchronized with the actual VBL rate on standard North American Apples; European Apples use 50 Hz as a standard.)

```
    Call:                TimeData
    Offset Location:     $Cn1C
    Input:               Accumulator bit 0:    0 for 60 Hz
                                               1 for 50 Hz

    Note:    All other accumulator bits are reserved, and must be set to 0.

    Output:              carry bit clear
                         screen holes unchanged
```

You must make this call just prior to calling InitMouse to be effective. If you want to change the interrupt rate in the middle of an application, you must call TimeData with the appropriate value in the accumulator, then call InitMouse (which generates an interrupt). InitMouse resets the mouse position, mode, clamps, etc. to their default values. If you fail to call TimeData, InitMouse will use a default interrupt rate of 60 Hz.

Note: This call exists only on the AppleMouse card for the IIe or][+ and should only be used when you know you are working with a IIe or][+. A user may configure a IIGS to 50 Hz by holding down the Option key while rebooting. The standard North American Apple IIc will not generate 50 Hz VBL interrupts.

END OF FILE TN.Mous.002

##
FILE: TN.Mous.003
##

Apple II
Technical Notes

Developer Technical Support

Mouse
#3: Mode Byte of the SetMouse Routine

Revised by: Matt Deatherage November 1988
Revised by: Rilla Reynolds November 1985

This Technical Note explains the results of turning the mouse on and off through the mode byte of the SetMouse routine.

What Turning the Mouse Off Does

In the description of SetMouse and the mouse mode, the low-order bit of the mouse mode is said to control mouse off and mouse on. This terminology is somewhat misleading. When this bit is set to 0, the mouse is off only in the following respects:

1. The mouse position is not tracked; any mouse motion is ignored.
2. ReadMouse calls do not update the status byte or the screen holes, except on the IIGS, where ReadMouse always functions the same, regardless of mouse on or mouse off.
3. Button and movement interrupts are not generated, regardless of the other mouse mode bits. Pure VBL interrupts can still be generated, however, if bit 3 is set.

What Turning the Mouse Off Does Not Do

Other mouse functions will continue to work as usual when the mouse is off. PosMouse and ClearMouse will change the mouse position, ClampMouse will set new clamp values, etc. In particular:

1. Turning the mouse off and on with the mode byte does not reset any mouse values, including position. The mouse position retains the last values it had before the mouse was turned off until it is turned on again.
2. A mode byte of $08 (mouse off but VBL interrupt on) will generate VBL interrupts.

Further Reference
- o Apple IIGS Firmware Reference
- o Apple IIe Technical Reference Manual
- o Apple IIc Technical Reference Manual

END OF FILE TN.Mous.003

###
FILE: TN.Mous.004
###

Apple II
Technical Notes

Developer Technical Support

Mouse
#4: Mouse Firmware Bug Affecting ServeMouse

Revised by: Matt Deatherage November 1988
Revised by: Rilla Reynolds January 1985

This Technical Note documents a bug in the mouse firmware on the AppleMouse card which affects the way ServeMouse works.

There is a bug in the AppleMouse II 6805 firmware which may affect the way ServeMouse works in an application. If the application takes more than one video cycle (normally about 16 ms) to respond to a mouse-generated interrupt, then ServeMouse will not claim the interrupt. The 6805 returns an interrupt status byte of $00 (i.e., no mouse interrupt pending), and the 6502 firmware sets the carry bit (although the interrupt is also cleared by the ServeMouse call). This situation can be confusing, and under ProDOS or Pascal it can be lethal. We have identified the following solutions, any of which should work:

If you are not working under an established operating system (i.e., ProDOS or Pascal):

1. Do not allow unclaimed interrupts to be fatal to your application. Ignore them.
2. Always service mouse interrupts within 1/60 of a second. If you are forced to disable interrupts for a longer period, first use SetMouse to set the mouse mode to 0, then call ServeMouse to clear any existing mouse interrupt. After interrupts are re-enabled, restore the mouse mode.

If you are working under an established operating system (i.e., ProDOS or Pascal) for which unclaimed interrupts are fatal and the mouse is not the only interrupting device:

1. Write the mouse interrupt handler to claim all unclaimed interrupts and make sure the mouse interrupt handler is installed last, otherwise the interrupt will never get through to any interrupt handlers which follow that of the mouse.

Note: This solution may cause cursor flicker by delaying the application's response to VBL interrupts.

2. Write a spurious interrupt handler (also known as a "daemon"), not associated with any device, which claims all unclaimed interrupts (i.e., clears the carry bit then exits). For the reason just mentioned, this interrupt handler must be installed last.

Note: Under ProDOS, this limits the number if interrupting devices to three.

This bug exists in the AppleMouse card, therefore you must deal with it when you are writing eight-bit programs for the Apple][+, IIe, IIc and IIGS which use the mouse. The Apple IIGS does not have this bug in its internal mouse firmware, so sixteen-bit "native" mode programs are not affected by it.

END OF FILE TN.Mous.004

##
FILE: TN.Mous.005
##

Apple II
Technical Notes

Developer Technical Support

Mouse
#5:   Check on Mouse Firmware Card

Revised by: Matt Deatherage   November 1988
Revised by: Rilla Reynolds   November 1985

This Technical Note formerly described a protocol which allowed applications to check a device which matched the mouse firmware identification for support of interrupts.

The convention formerly described by this Note has been removed since it conflicted with the Pascal 1.1 Firmware Protocol. The conflict could cause Pascal to believe that optional firmware routines were present, when the card being checked was simply stating that it supported interrupts.

Apple recommends that any mouse-type device which matches the mouse ID bytes should support interrupts exactly as the Apple mouse firmware does. Applications which believe they have found an Apple mouse have a reasonable right to expect that the device they actually have found will behave as an Apple mouse.

END OF FILE TN.Mous.005

```
#####################################################################
### FILE: TN.Mous.006
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Mouse
#6: MouseText Characters

Revised by: Matt Deatherage January 1989
Revised by: Rilla Reynolds November 1985

This Technical Note describes the MouseText character set which is available on all currently produced Apple II computers.
Changed since November 1988: Corrected typographical errors in the BASIC and assembly language program examples.

In unenhanced Apple IIe computers, the alternate character set contained two sets of inverse uppercase characters. In the enhanced Apple IIe, and in all Apple IIc and IIGS computers, one set of inverse uppercase characters is replaced by a MouseText character set. MouseText is a set of graphical characters designed to allow Apple II computers to display a desktop metaphor on the text screen. The Apple II Desktop Toolkit uses these characters, as do applications like AppleLink-Personal Edition.

If your program used the set of inverse uppercase characters which were replaced by MouseText (the set mapped to ASCII values $40-$5F), your program will display MouseText characters instead of inverse uppercase characters on all currently-produced Apple II computers. If your program used the other set of inverse uppercase characters (ASCII values $00-$1F), it will display inverse capital characters as expected.

The following will help you identify if the changes affect you or not.

1. If your program is written entirely in BASIC or Pascal or your assembly language program calls the COUT routine to put characters on the screen, you are not affected. The only exception would be if you print (POKE) inverse characters directly to the text screen in BASIC.
2. If your program uses the standard character set (checkerboard cursor) you are not affected.
3. If your program is using the alternate character set (solid cursor) and is directly storing values (via POKE) to the text display area, you will encounter problems if your character values are in the range from 64 ($40) to 95 ($5F). To recreate the original display, use values in the range from 0 ($0) to 31 ($1F) instead. Note that these lower values display as inverse uppercase characters on older machines as well.

Following are the methods recommended for accessing MouseText characters from various languages:

AppleSoft BASIC

1. Turn on the video firmware with PR#3 (if under DOS 3.3 or ProDOS, use PRINT CHR$(4);"PR#3")
2. Enable MouseText characters by printing an ASCII 27 ($1B) to the screen.
3. Set inverse printing mode by printing an ASCII 15 ($0F) to the screen.

To stop displaying MouseText characters:

1. Disable MouseText characters by printing an ASCII 24 ($18) to the screen.
2. Set normal print mode (if desired) by printing an ASCII 14 ($0E) to the screen.

This short BASIC program displays all MouseText characters under DOS 3.3 and ProDOS:

```
    10    D$=CHR$(4)
    20    PRINT D$;"PR#3": REM Turn on the video firmware
    30    PRINT:REM This is so the screen won't be in inverse
    40    PRINT CHR$(15):REM Set inverse mode
    50    PRINT CHR$(27);"ABCEDFGHIJKLMNOPQRSTUVWXYZ@[]^_\";CHR$(24)
    60    PRINT CHR$(14):END
```

Assembly Language

Assembly language programs are expected to follow the same procedure as AppleSoft BASIC. Use calls to COUT to print MouseText characters to the screen. The following is a sample assembly language program which displays two MouseText characters (which create a folder icon), along with their inverse uppercase equivalents:

```
START        LDA #$A0           ;USE A BLANK SPACE TO
             JSR $C300          ;TURN ON THE VIDEO FIRMWARE
             LDY #0             ;INITIALIZE COUNTER
LOOP         LDA STR,Y          ;GET VALUE
             JSR $FDED          ;SEND IT THROUGH THE COUT ROUTINE
             INY
             CPY STRLEN
             BNE LOOP           ;=>NOT DONE YET
             RTS
STR          DFB $1B,$58,$59,$18,$58,$59
                                ;MOUSETEXT ON, SHOW, MOUSETEXT OFF, SHOW
STRLEN       EQU *-STR          ;LENGTH OF STR
```

Note: Using MouseText on the text screen by directly poking or storing MouseText character values into the text buffer is not supported by Apple at this time. Should the MouseText character set require remapping in the future, those programs which use the methods outlined in this Note should still work with any new mapping. Those which directly store MouseText values run the strong risk of display failure under a new mapping.

Apple II Pascal

1. Output a CHR(27), an escape character, to enable MouseText.
2. Output a CHR(15) to turn on inverse video.
3. Output the appropriate capital letter for the desired MouseText character.

A Pascal sample program:

```
    PROGRAM OUTPUT_MOUSETEXT
    VAR CMD:PACKED ARRAY[0..1] OF 0..255
    BEGIN
        CMD[0]:=27; CMD[1]:=15;
        UNITWRITE(1,CMD,2); {turn on MouseText mode}
        {code to display MouseText
                    .
                    .
                    .
        }
        CMD[0]:=24;
        UNITWRITE(1,CMD,1); {turn off MouseText mode}
    END
```

Pictorial descriptions of the MouseText character set may be found in the Apple IIe Technical Reference Manual, the Apple IIc Technical Reference Manual, and the Apple IIGS Hardware Reference.

Note: The pictures of MouseText characters in these manuals differ from early implementations. In early MouseText character sets, the icons mapped to the letters F and G combined to form a "running man." In current production, these letters are different pictures (an inverse carriage return symbol and a window title bar pattern) which form no picture when placed next to each other. Programs should not attempt to use the running man MouseText characters.

Further Reference

---

o Apple IIGS Hardware Reference
o Apple IIe Technical Reference Manual
o Apple IIc Technical Reference Manual

END OF FILE TN.Mous.006

```
#####################################################################
### FILE: TN.Mous.007
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Mouse
#7: Mouse Clamping

Revised by: Matt Deatherage November 1988
Written by: Rilla Reynolds October 1986

This Technical Note describes the different methods available for obtaining mouse clamping values on different Apple II family machines.

AppleMouse Card

The AppleMouse card delivers clamping values on request. There is no specific mouse routine to obtain the clamping values, but an internal routine may be used by the mouse card to return them. The values are returned as minimum and maximum values of X and Y clamps, both low and high bytes.

Note: The following code is the only supported use of the $Cn1A offset into the mouse card firmware, and this entry point is not available in any other mouse firmware implementation.

```
GetClamp    LDA    #$4E
            STA    $478            ;Needed by Mouse Card firmware
            LDA    #$00
            STA    $4F8            ;Needed by Mouse Card firmware
            STA    Tmp             ;Zero-page word for indirect addressing
            LDA    #CN             ;$C<slot>, obtained prior to this rtn
            STA    Tmp+1           ;$C<slot>00, Mouse Card firmware main entry
            STA    ToCard+2
            LDY    #$1A
            LDA    (Tmp),Y
            STA    ToCard+1        ;Mouse Card firmware GetClamp entry
            LDA    #7
            STA    BytePtr
            LDY    #N0             ;$<slot>0, for Mouse Card firmware
GetByte     LDX    #CN             ;$C<slot>, for Mouse Card firmware
            LDA    #0              ;Needed by Mouse Card firmware
            JSR    ToCard
            LDA    $578            ;Clamp byte returned by Mouse Card firmware
            LDX    BytePtr
            STA    Byte,X
            DEC    $478
            DEX
            STX    BytePtr
            BPL    GetByte
            RTS
```

```
ToCard      JMP    $0000          ;Operand modified by rtn
Byte        DS    8,0             ;MinXH,MinYH,MinXL,MinYL,MaxXH,MaxYH,MaxXL,MaxYL
BytePtr     DS    1,0
```

Apple IIc

For the Apple IIc, you can get clamping values by reading the following auxiliary memory screen holes:

```
    $47D MinXL        $67D MaxXL
    $4FD MinYL        $6FD MaxYL
    $57D MinXH        $67D MaxXH
    $5FD MinYH        $6FD MaxYH
```

Apple IIGS

On the Apple IIGS, the Miscellaneous Tool Set call GetMouseClamp returns the mouse clamp values as four words on the stack. This call is documented in the Apple IIGS Toolbox Reference, Volume 1.

Further Reference
o   Apple IIGS Toolbox Reference, Volume 1

END OF FILE TN.Mous.007

##
FILE: TN.Pasc.004
##

Apple II
Technical Notes

Developer Technical Support

Pascal
#4: Pascal Declarations and the Directory Structure of a Blocked Volume

Revised by: Matt Deatherage November 1988
Revised by: Guillermo Ortiz November 1985

This Technical Note formerly described the declarations your Pascal program needs to read an Apple II Pascal disk as well as the actual layout of an Apple-Pascal blocked volume.

The Apple II Pascal 1.3 Manual (pp. IV-14 to IV-16) now documents the information which this Note formerly discussed.

Further Reference
o Apple II Pascal 1.3 Manual

END OF FILE TN.Pasc.004

```
#####################################################################
### FILE: TN.Pasc.010
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Pascal
#10: Configuration and Use of the Apple II Pascal Run-Time Systems

Revised by: Cheryl Ewy — November 1988
Revised by: Cheryl Ewy — June 1985

This Technical Note describes the Apple II Pascal Run-Time Systems which permit the "turnkey" execution of application software which has been developed using Apple Pascal.

System Overview

The Run-Time Systems support only the execution of an application package. Unlike the Pascal Development System, the Run-Time Systems do not contain the Assembler, Compiler, Editor, Filer or Linker, nor even an error reporting mechanism at the system level. System operations such as transferring files, compacting disks (Krunching), and the reporting of and recovery from errors, are all left to the application program. It is the software developer's responsibility to design and implement friendly, entirely self-contained packages for use with the Run-Time Systems. The safest assumption to make when developing such packages is that the user is not only unfamiliar with the facilities of the Pascal Development System, but may also be ignorant of computer operation and use in general.

The three run-time systems currently available are :

- The 48K Run-Time System V1.2 (standard and stripped)
- The 64K Run-Time System V1.3 (standard only)
- The 128K Run-Time System V1.3 (standard only)

The name of each Run-Time System indicates the minimum amount of RAM necessary for proper operation. Any additional RAM available will not be used by the Run-Time Systems.

The 48K Run-Time System has not been updated to version 1.3, as have the 64K and 128K Run-Time Systems. Thus, the changes and improvements made to Pascal for version 1.3 are not available in the 48K Run-Time System. Specifically, the 48K Run-Time System can only use Disk II drives and can only boot from slot 6. See the Apple II Pascal 1.3 Manual for more information on the differences between versions 1.2 and 1.3 of Apple II Pascal.

There are two configurations of the 48K Run-Time System available, one of which provides more free memory for the application package's programs and data than does the other. Except as noted later, the standard configuration of the Run-Time System supports all features of the Pascal Development System that are relevant to turnkey execution of application software. The stripped configuration lacks set operations and floating-point arithmetic.

Contents of the Apple II Pascal Run-Time System Disks

The following files are contained on the Apple II Pascal 1.2 48K Run-Time System disk (RT48:):

o	RTSTND.APPLE	48K Run-time standard P-machine.
o	RTSTRP.APPLE	48K Run-time stripped P-machine.
o	SYSTEM.PASCAL	48K Run-time operating system.
o	RTBSTND.BOOT	Contains the boot code for RTSTND.APPLE.
o	RTBSTRP.BOOT	Contains the boot code for RTSTRP.APPLE.
o	RTBOOTLOAD.CODE	Utility program to load 48K Run-time boot code onto blocks 0 and 1 of Vendor Product disk.

The following files are described below:

- SYSTEM.LIBRARY
- SYSTEM.ATTACH
- RTSETMODE.CODE
- II40.MISCINFO
- II80.MISCINFO
- IIE40.MISCINFO
- SYSTEM.MISCINFO
- SYSTEM.CHARSET

The following files are contained on the Apple II Pascal 1.3 64K Run-Time System disk (RT64:):

o	SYSTEM.APPLE	64K Run-time standard P-machine.
o	SYSTEM.PASCAL	64K Run-time operating system.

The following files are described below:

- SYSTEM.LIBRARY
- SYSTEM.ATTACH
- RTSETMODE.CODE
- II40.MISCINFO
- II80.MISCINFO
- SYSTEM.MISCINFO
- SYSTEM.CHARSET

The following files are contained on the Apple II Pascal 1.3 128K Run-Time System disk (RT128:):

o	SYSTEM.APPLE	128K Run-time standard P-machine.
o	SYSTEM.PASCAL	128K Run-time operating system.

The following files are described below, and are identical to the 64K Run-Time System files:

- SYSTEM.LIBRARY
- SYSTEM.ATTACH
- RTSETMODE.CODE
- SYSTEM.MISCINFO

o SYSTEM.CHARSET

The Development Systems referred to in the following file descriptions are the Apple II Pascal 1.3 Development System when discussing files on the 64K and the 128K Run-Time System disks and the Apple II Pascal 1.2 Development System when discussing files on the 48K Run-Time System disk.

SYSTEM.LIBRARY	contains the run-time versions of the same Intrinsic Units supplied with the Development System. These Units are for use only with the Run-Time System and will not execute properly in the Development environment. Conversely, only the Units in this library, not those on the Development System disks, should be used when executing programs in the Run-time environment. Note that the developer is free to add his own Intrinsic Units to SYSTEM.LIBRARY.
SYSTEM.ATTACH	is a run-time version of the dynamic driver-attachment program described in Apple II Pascal Device and Interrupt Support Tools. This version may only be used with the Run-Time Systems.
RTSETMODE.CODE	is a utility program that permits the vendor to arm or disarm any or all of four system options: Filehandler Overlay, Single Drive System, Ignore External Terminal, and Get/Put and Filehandler Overlay.
MISCINFO	files are identical to those supplied on the Development System disks and are supplied here only for the sake of redundancy.
SYSTEM.CHARSET	is identical to the file supplied with the Development System; it is included here only for the sake of redundancy. This file is needed on the Vendor Product Disk only if TURTLEGRAPHICS is used.

Of the files supplied on the Run-Time System disks, the final Vendor Product Disk should contain only the Run-time P-machine (SYSTEM.APPLE, RTSTND.APPLE, or RTSTRP.APPLE), SYSTEM.PASCAL, SYSTEM.LIBRARY, the appropriate MISCINFO file renamed to SYSTEM.MISCINFO, and, optionally, SYSTEM.CHARSET. SYSTEM.ATTACH, with its attendant data files should be included on the Vendor Product Disk if special device drivers must be bound into the system for use by the Application Package. All other files on the Run-Time System disks are used in creating and configuring the Vendor Product Disk.

Operation

The term Vendor Product Disk, as used throughout this Technical Note, refers to the primary (boot) disk in a turnkey application package, which is assumed to contain the following software: the Run-time P-machine, the Run-time Operating system, a SYSTEM.LIBRARY file, a SYSTEM.MISCINFO file, and the files comprising the application package's programs (and any necessary data). In most instances, the Vendor Product Disk will be the only software disk in the package. Larger systems, however, may also include other disks that contain additional software and data which will not fit on the boot disk.

Note that the main application program must be named SYSTEM.STARTUP, so the Run-Time System can find it when booting.

A two-stage boot process can be used with the 64K and 128K Run-Time Systems if the necessary boot files listed above cannot fit on a single disk. In this case, the primary boot disk would contain only the Run-time P-machine. A second-stage boot disk would contain the remainder of the files. A two-stage boot process cannot be used with the 48K Run-Time System.

The Boot Process

The boot code (contained in blocks 0 and 1 of the boot disk) is loaded into memory by the Autostart ROM. It checks for the P-machine file and loads it into RAM. The P-machine, in turn, brings in and initializes the Run-time operating system. (In the case of a two-stage boot, the message "Insert boot disk with SYSTEM.PASCAL on it, then press RETURN" appears after the P-machine has been loaded. The user should then insert the second-stage boot disk and press the Return key, which results in the Run-time operating system being loaded and initialized.) The first noteworthy action taken by the operating system is to execute SYSTEM.ATTACH, if that utility program is available on the Vendor Product Disk. Remember that SYSTEM.ATTACH must not be present on the Vendor Product Disk unless special, low-level I/O drivers must be bound into the system. As explained more fully in Apple II Pascal Device and Interrupt Support Tools, SYSTEM.ATTACH uses two special data files and will fail if these files are not present on the boot disk. Putting SYSTEM.ATTACH on the Vendor Product Disk without also providing the required data files insures consistent failure of the system boot process. It is possible to include SYSTEM.ATTACH on the Vendor Product Disk, while defeating the automatic execution of it at boot time, by changing its name.

The boot process culminates when the main application program, SYSTEM.STARTUP, is loaded and executed. Any failure during the boot process is fatal. Whenever possible, a failure will display the following message:

SYSTEM FAILURE NUMBER nn. PLEASE REFER TO PRODUCT MANUAL.

Here, nn refers to the actual number reported when the failure occurs. This number corresponds to one of the following failures:

01 Unable to load specified program
02 Specified program file not available
03 Specified program file is not code file
04 Unable to read block zero of specified file
05 Specified code file is un-linked
06 Conflict between user and intrinsic segments
07 UNASSIGNED ERROR CODE
08 Required intrinsics not available
09 System internal inconsistency
10 Can't load required intrinsics/Can't open library file
11 Specified code file must be run under the 128K system
12 Original disk not in boot drive

Clearly, these messages are useful as debugging tools as well as in mechanisms for field failure reporting. The Product Manual mentioned in the bootstrap failure message is, of course, the vendor's own product manual. It is the responsibility of the vendor to enumerate and explain for the user the situations in which bootstrap failures may occur, as well as suggest remedies for these failures.

General Considerations

Once the program is loaded and running, operation proceeds normally and may even include removal of the system disk. (It is, however, the responsibility of the application package to protect itself against the possibility that the system disk will not be on-line when a segment must be loaded or when a specific subprogram must be chained to. At such times, the application software should first determine whether or not the required disk is on-line, and, if not, suspend operation, after giving a suitable prompt, until the user has inserted the disk in the appropriate drive.) Any errors that occur during execution of the application package cause the system to transfer program control to a specific procedure in the currently-executing application program, where code intended to respond to errors is assumed to exist. If any program in the application system terminates without chaining to another one, the Run-time system reboots into SYSTEM.STARTUP.

Specifications

Available Configurations

The memory requirements of different applications impose the need for different Run-Time Systems. The developer should choose one of the systems as the target environment, and keep its limitations and capabilities in mind during design and implementation of the application package. Apple currently supports the following Run-Time Systems:

- The 48K Run-Time System V1.2 (standard and stripped)
- The 64K Run-Time System V1.3 (standard only)
- The 128K Run-Time System V1.3 (standard only)

The difference between the standard and stripped versions of the 48K Run-Time System is that the stripped version does not support set operations or floating point arithmetic, thereby making more memory available for the application.

The chart below summarizes the amount of free memory that is available under the different Run-Time Systems for use by the application package. Note that when swapping is set to level 1, the amount of memory available to the application package is increased by approximately 3660 bytes.

	No Swapping	Swapping on byte level
48K Standard	23372 bytes	27040 bytes
48K Stripped	25676 bytes	29344 bytes
64K	40290 bytes	43958 bytes
128K (code)	40758 bytes	44410 bytes
128K (data)	44502 bytes	44526 bytes

Figure 1-Free Memory in Run-Time Systems

Note: The amount of free memory available with the 64K Run-Time System is reduced by 1024 bytes if it is operating in 40-column mode. Similarly, the amount of free memory available for data in the 128K Run-Time System is reduced by 1024 bytes if the system is operating in 40-column mode.

There is another level of swapping (level 2) which provides an additional 810 bytes of usable memory, however, using GET or PUT to disk will be slow if swapping level 2 is selected since these routines will have to be loaded from disk repeatedly. READ and WRITE to disk will also be slow since they use GET and PUT. BLOCKREAD, BLOCKWRITE, UNITREAD, and UNITWRITE will be unaffected.

Swapping can be set to the desired level by using RTSETMODE (described later) or by calling a procedure in CHAINSTUFF before chaining to another subprogram. See the Apple II Pascal 1.3 Manual for further information on swapping.

Use Environment

The hardware environment must include the following:

48K Run-Time System	An Apple][or][+ with 48K of RAM (minimum), or an Apple IIe, IIc or IIGS.
64K Run-Time System	An Apple][or][+ with 48K of RAM and an Apple Language Card, or an Apple IIe, IIc or IIGS.
128K Run-Time System	An Apple IIe with an Extended 80-Column Text Card, an Apple IIc or an Apple IIGS.
All Run-Time Systems	At least one disk drive in slot 4, 5, or 6. Video screen or external terminal (video screen preferred).

Note that the Run-Time Systems support all standard Apple peripheral cards. Other cards may not operate properly, especially if they include firmware that depends upon specific internal characteristics of the P-machine or operating system. SYSTEM.ATTACH must be used by those vendors who wish to reconfigure the BIOS (Basic I/O Subsystem) to support non-standard peripheral devices. Through the ATTACH facility, it is possible to assign new physical devices to any of the existing logical I/O units in the Pascal system, as well as retain the standard device assignments while adding new devices to the system. Drivers prepared for use with SYSTEM.ATTACH are bound into the system dynamically when it boots. Note that the addition of special I/O drivers to the system will reduce the amount of free memory available for use by the applications code, since drivers are loaded on the Pascal system heap. For more information, see Apple II Pascal Device and Interrupt Support Tools.

Restrictions and Considerations

1. SYSTEM.ATTACH and the CHAINSTUFF, LONGINTIO, and PASCALIO units in SYSTEM.LIBRARY make assumptions about the internal structure of the Pascal operating system. Because the internals of the Run-time operating systems are different from those in the Development System, only the versions of CHAINSTUFF, LONGINTIO, PASCALIO and SYSTEM.ATTACH that are supplied on the Run-Time System disks should be used in the Run-time execution environment. (These special versions should never be used in the Development environment.)

2. The units TRANSCEND and TURTLEGRAPHICS employ floating-point operations, so software intended to be executed under the 48K stripped Run-Time System should not use them. For software that employs the TURTLEGRAPHICS procedure TURNTO, note that turns through right angles and null angles are treated as special cases, and the TURTLEGRAPHICS unit uses only integer arithmetic in calculating the trigonometric values needed to execute them. TURTLEGRAPHICS may be used under the 48K stripped Run-Time System if the turtle is allowed to make only right-angle turns (i.e., the HILBERT demonstration program on the APPLE3: disk). Attempts to draw arbitrary curves, as demonstrated in the GRAFDEMO program on

APPLE3:, will produce execution errors in the 48K stripped Run-time environment.

3. Pascal's special function keys retain their meanings in the Run-Time Systems. The following keys have special meanings:

Control-@	Break
Control-A	Switch to alternate half of screen
Control-F	Flush screen display
Control-S	Freeze (Stop) screen display
Control-Z	Initiate auto-follow mode
Control-W, Control-E	Upper/lower case activation
Control-R, Control-T	Reverse video toggles
Control-K	Left square bracket
Shift-M	Right square bracket

Note: Some of these special function keys are ignored by Pascal if it is running on an Apple IIe, IIc or IIGS. Also, it is possible to disable some of these special key functions. See Apple II Pascal 1.3 Manual for complete details.

4. The Run-Time System will operate correctly only with programs that have been prepared for execution in the Apple II Pascal environment.using Apple's Pascal compiler or Pascal-system assembler on either an Apple II or an Apple ///.

5. The Run-Time System is optimized for operation with Apple's built-in video output screen. There is no easy way for a turnkey package to reconfigure its host Run-Time System to use the random-cursor facilities of any arbitrary external terminal. Therefore, it is expected that users of the system will be operating with the standard Apple video screen and not an external terminal. Any program that makes use of screen control, such as clearing the screen, random cursor addressing, or backspacing, is not likely to work properly on an external terminal. To avoid this problem, the Run-Time System contains a switch which can be set through the RTSETMODE program (explained below). When set, this switch causes the system to ignore an external terminal, if one is connected. Simple programs that do not make use of any screen control may leave the external terminal switched in without any adverse consequences.

Run-Time System Configuration Utilities

RTSETMODE (provided with all Run-Time Systems)

Flags which note the state of four system options are contained within a special part of the directory of any Run-Time System boot disk. (These flags will not normally be present on disks prepared for or used with the Pascal Development System.) When a flag is set (TRUE), the corresponding system option is enabled. The option is disabled when the corresponding flag is reset (FALSE). At boot time, the option flags are checked and are used during a dynamic configuration process which occurs before the application software is executed.

The RTSETMODE utility is used by the application developer to set or reset the option flags, according to the requirements of the application package. In operating RTSETMODE, the developer first selects the Pascal volume to be affected, then answers four yes-or-no questions by pressing the Y or N keys, respectively. Responding to any prompt for input by pressing only the Return key causes immediate termination of the program.

Answering yes to any of the following questions arms the indicated option (setting the corresponding flag), while answering no disarms the option (and resets the corresponding flag).

> Arm Filehandler Overlay Option? Arming this option sets OS swapping to level 1. Operating System code related to disk file opening and closing is swapped into memory as needed by the application software, thus freeing approximately 3660 bytes of RAM for use by the application.
>
> Arm Single-drive System Option? With this option armed, the initial boot process is finished, the Pascal system will not assume the availability of any disk drives other than the boot drive. Specifically, volume searches will be limited to the boot drive. The application may still use Apple Pascal's UNITREAD and UNITWRITE procedures to access any other drives which may be connected to the system.
>
> Arm Ignore External Terminal Option? Arming this option insures that the Pascal system will always operate in 40-column mode, regardless of whether or not an external terminal interface or 80-column card is available.
>
> Arm Get/Put and Filehandler Overlay Option? Arming this option sets OS swapping to level 2. Operating System code related to disk file opening and closing, as well as GET and PUT to disk is swapped into memory as needed. (See above for more information on swapping level 2.)

After the four-question sequence, RTSETMODE asks the user to confirm that all information input to that point is correct and should be used to update the Vendor Product Disk. If so, an attempt is made to update the disk's directory with the new set of option flags, and RTSETMODE finishes by reporting the success or failure of the update operation.

Developers should note that only exact copies of a Run-time boot disk will retain its option flags. Transferring the Run-Time System and applications software from disk to disk on a file-by-file basis will not transfer the option flags between the disks. For this reason, it is recommended that RTSETMODE be applied to the product master of any package based on Run-time immediately prior to releasing that master to production, to insure the correct status of the option flags.

If a two-stage boot will be used for a run-time application, RTSETMODE must be run on both boot disks since the flags are checked by both the P-machine and the operating system.

RTBOOTLOAD (48K Run-Time System only)

This program is used to transfer to the Vendor Product Disk the proper boot code for the chosen 48K Run-time configuration (STND or STRP). Responding to any prompt for input by pressing only the Return key results in immediate termination of the program. RTBOOTLOAD first asks for the name of the file which contains the appropriate boot code (either RTBSTND.BOOT or RTBSTRP.BOOT). The filename must be entered exactly as it appears in the directory (including a volume prefix if the file is not on the default volume), or the program will not be able to find the file, and will repeat its request for a filename. Once it has fetched the boot code, RTBOOTLOAD asks for the volume name of the Vendor Product Disk, then waits for the user to press the space bar (thus providing the user with an opportunity to insert the selected volume, if necessary) before attempting to transfer the boot information. The success or failure of the transfer is reported before RTBOOTLOAD terminates. This program is only supplied on the 48K Run-Time System disk and should never be used to transfer boot information to a disk which contains the 64K or 128K Run-Time Systems, as doing so will prevent the systems from booting correctly.

Error Handling

If an error in execution or I/O occurs during program operation, the Run-Time System attempts to let the application package itself acknowledge, and if possible, recover from the error condition. As with the Pascal Development environment, the application developer is free to use the $I- and $R- compiler options to assume localized, programmatic control of the corresponding error situations.

When the Run-Time System detects an error, it stores the error number in IORESULT and calls PROCEDURE NUMBER TWO of the currently-executing program. This is the procedure in segment number 1 that has been given the procedure number 2 by the compiler. In other words, it is the first one declared after the program heading that is not itself a unit or segment procedure, or within a unit or segment procedure. In a compiler listing, PROCEDURE NUMBER TWO may be identified as those lines whose S (segment) number is 1, and whose P (procedure) number is 2.

PROCEDURE NUMBER TWO may be declared as a forward procedure since the procedure number is assigned at the forward declaration.

From now on, PROCEDURE NUMBER TWO will usually be called the error handler, since it must always be reserved by the application programmer for the sole purpose of handling errors. The error handler may not have any parameters, and must always be declared as a PROCEDURE, never as a FUNCTION.

The error handler can determine what kind of error has occurred by checking the value of the IORESULT function. In the Development System, this function is restricted to containing the codes for any I/O errors that might occur during execution. In the Run-Time Systems, IORESULT has been extended to report all system errors, as well as the usual I/O errors.

Here are all the values IORESULT can assume during Run-time execution:

```
    00 No error                     100 Unknown Run-time error
    01 Bad block, parity error      101 Value range error
    02 Illegal unit number          102 No procedure in segment table
    03 Illegal I/O request          103 Exit from uncalled procedure
    04 Data-com timeout             104 Stack overflow
    05 Volume went off-line         105 Integer overflow
    06 File lost in directory       106 Divide by zero
    07 Bad file name                107 Nil pointer reference
    08 No room on volume            108 Program interrupted by user
```

```
    09 Volume not found             109 System I/O error
    10 File not found               110 User I/O error
    11 Duplicate directory entry    111 Unimplemented instruction
    12 File already open            112 Floating point error
    13 File not open                113 String overflow
    14 Bad input format             114 Programmed HALT
    16 Disk is write-protected      115 Programmed breakpoint
    17 Illegal block number         116 Codespace overflow
    18 Illegal buffer address
    19 Must read a multiple of 512 bytes
    20 Unknown ProFile error
    64 Device error
```

It is recommended that a program's error handler should simply report system error for all cases except those which are relevant to the program. Global state variables in the program may be used to help determine the nature of the problem and report it to the user. Note that a system reboot occurs if an attempt is made to exit the program (without chaining to another).

After the error handler finishes its operation, control returns to the caller of the procedure where the error occurred (unless the error was fatal). In this way, program operation may be continued, cleanly and simply, after an error is handled. The caller of a failure-prone procedure can set and test status flags to determine whether or not the called procedure completed its operation and either repeat the procedure call or perform an alternative action.

In developing particularly large systems where program chaining is used, the application programmer should remember that each chained program must reserve PROCEDURE NUMBER TWO as an error handler.

Following are two programming examples. The first shows a typical error handler routine, and the second is a program fragment that demonstrates an error recovery technique.

```
(* EXAMPLE #1 -- ERROR HANDLER *)

    (* THE FOLLOWING PROCEDURE IS ONLY *)
    (* CALLED BY THE OPERATING SYSTEM *)

    PROCEDURE ErrorHandler;

        PROCEDURE Message(Space: Boolean; S: String);
        VAR Ch : Char;
        BEGIN (* Message *)
            WriteLn;
            WriteLn('*** ',S);
            IF Space THEN
                BEGIN
                    Write('*** Press SPACE-BAR to continue');
                REPEAT
                        Read(Keyboard, Ch)
                    UNTIL ((Ch = ' ') AND (NOT EoLn));
                END;
            END   (* Message *);

        BEGIN (* ErrorHandler *)
            IF (IOResult = 14) THEN
```

```
            Message(True,'That is not a legal integer!')
        ELSE IF (IOResult = 106) THEN
            Message(True,'Division by zero is impossible!')
        ELSE BEGIN
            Message(False,'System error.  Please reboot.');
            WHILE True DO (* Hang *);
        END;
    END   (* ErrorHandler *);

    (* END OF EXAMPLE #1 *)


    (* EXAMPLE #2 -- ERROR RECOVERY USING ERROR HANDLER OF EXAMPLE #1 *)

    PROCEDURE Calculator;
    (* Features recovery from input or arithmetic error. *)
     TYPE Order = (First, Second);
    VAR A,B : Integer;
         Flag : Boolean;

        PROCEDURE GetNumber(Which: Order; VAR Number: Integer);
        BEGIN
            Write('Input the');
            IF (Which = First) THEN
                 Write(' first')
            ELSE Write(' second');
            Write(' number: ');
             Read(Number);  ReadLn;
             Flag := True;
        END   (* GetNumber *);

         PROCEDURE Answer;
            VAR R : Real;
            BEGIN
                R := A / B; (* Bombs if B=0 *)
                WriteLn;
                WriteLn(A,' divided by ',B,' is ',R);
             END   (* Answer *);

        BEGIN (* Calculator *)
            REPEAT
                Flag := False;
                WriteLn;
                WriteLn;
                REPEAT
                    GetNumber(First,A)
                UNTIL Flag;
                Flag := False;
                WriteLn;
                REPEAT
                    GetNumber(Second,B)
                UNTIL Flag;
                Answer;
            UNTIL Eof;
        END   (* Calculator *);

        (* END EXAMPLE #2 *)
```

To illustrate the effect of the Run-Time System's error handling mechanism, here is the interaction between user and machine during a typical run of the above Calculator program. User-input is terminated by a press of the Return key in all cases except the first and last. In the first case, the error handler is invoked during the erroneous numeric input. In the last case, the system accepts and acts upon a Control-C signal before the user has a chance to press any other keys.

```
Input the first number: N

*** That is not a legal integer!

Input the first number: 16

Input the second number: 0

*** Division by zero is impossible!

Input the first number: 16

Input the second number: 2

16 divided by 2 is 8

Input the first number: <Control-C>
```

As soon as the user presses Control-C, the Run-time system detects the end of the standard input file (EOF), and reboots (right back into Calculator).

Differences between the Pascal Development Systems
and the Run-Time Systems

Although the Run-Time Systems will run most Pascal code files exactly as does the Pascal Development System, the application developer must be aware of important differences between the two environments. As mentioned above, there is no system-level handling of any type of error that may occur, including stack overflow, arithmetic errors, or bad disk reads. It is left to the application package to respond to all error conditions. The typical user will not have access to (nor knowledge of) the Pascal Formatter or Filer.

Many programs which fit comfortably in the 64K Development System environment may fail to execute at all under the 48K Run-Time System due to the difference in available user memory. Similarly, programs developed with the 128K Development System may fail to execute under the 64K Run-Time System for the same reason. While large systems can be made to fit within the confines of a particular Run-time environment, this is possible only through use of Apple Pascal's program segmentation (overlay) and chaining facilities. It is suggested, however, that much thought and care be taken when using chaining and segmentation in software design, since these facilities, by their very nature, involve time-consuming disk accesses. Application software that abuses chaining or segmentation, or employs them in a careless fashion, may easily waste a large amount of time in disk thrashing, especially if swapping is being used. Finally, an application package runs the risk of massive failure unless calls to program overlays and chaining are preceded by checks that the expected disk is in the appropriate drive. This is especially important when the target machine includes only one disk drive (as is frequently the case).

The following items are never present in the Run-Time Systems:

- o System HOMECURSOR, CLEARSCREEN, and CLEARLINE functions
- o System prompt function
- o Compiler, Assembler, Linker, Editor, and Filer
- o IDSEARCH and TREESEARCH procedures

Programs that make use of information stored in specific memory locations within the Development System P-machine or that make assumptions about static or dynamic memory allocation at the operating system level (i.e., for the purpose of accessing system data structures) are likely to function incorrectly when executed in the Run-time environment. This failure to run is due to the code reorganization, compaction, and optimization that was necessary to produce the Run-Time Systems.

Creation of Vendor Product Disks

The following steps can be used as a guide for creating a Vendor Product Disk:

1. Format a disk using the Pascal Development System Formatter.
2. Transfer the files SYSTEM.APPLE (or RTSTND.APPLE or RTSTRP.APPLE), SYSTEM.PASCAL, SYSTEM.LIBRARY, SYSTEM.MISCINFO, and SYSTEM.CHARSET (if needed) from the Run-Time System disk to the Vendor Product Disk.
3. Transfer the code file or files for the application to the Vendor Product Disk. The main code file for the application must be named SYSTEM.STARTUP.
4. Run the Pascal Development System Library program to add any needed library units to SYSTEM.LIBRARY on the Vendor Product Disk.
5. Run RTBOOTLOAD to load the appropriate bootstrap code from RT48: onto the Vendor Product Disk. (48K Run-Time Systems Only)
6. Run RTSETMODE if you wish to arm the Filehandler Overlay option, the Single-Drive System option, the Ignore External Terminal option, or the Get/Put and Filehandler Overlay option.

Vendor Product Disks, or other disks which contain 48K Run-Time System software should be copied using only whole-volume transfer mechanisms, such as that provided by the Pascal system Filer. A succession of individual file transfers, or a wildcard transfer (such as transferring #4:= to #5:$), will only copy files from one disk to another. They will not copy the crucial 48K Run-time boot code between disks. Only whole-volume transfers (such as #4: to #5:, or SOUP: to NUTS:) will result in complete copies, containing the proper boot information.

Vendor Product Disks, or other disks which contain 64K or 128K Run-Time System software can be copied using either whole volume or individual file transfers since they do not contain special bootstrap information.

Apple FORTRAN and the Run-Time Systems

Apple FORTRAN programs will execute correctly under the Apple II Pascal Run-Time Systems (48K and 64K only), as long as no execution errors or untrapped I/O errors occur. Using only FORTRAN, it is impossible to produce object code that contains the specially-placed error-handling procedure to which control is transferred in the event of an untrapped error during Run-time execution. Furthermore, the FORTRAN Run-Time Support Library includes system-level code for handling FORTRAN I/O errors independently of the Apple Pascal system's own error-handling facilities. Execution of this special code will always lead to a system reboot in the Run-time environment.

Users who wish to provide turnkey packages based on FORTRAN object-code are advised to link the FORTRAN object-code to a Pascal host, as explained in the Apple FORTRAN Language Reference Manual. The only live code which the Pascal host must contain is the error-handling procedure that the Run-Time Systems require for robust execution of turnkey software.

Further Reference
o   Apple II Pascal 1.3 Manual
o   Apple II Pascal Device and Interrupt Support Tools
o   Apple FORTRAN Language Reference Manual

END OF FILE TN.Pasc.010

```
#####################################################################
### FILE: TN.Pasc.012
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Pascal
#12: Disk Formatter Routine

Revised by: Cheryl Ewy & Dan Strnad — November 1988
Revised by: Cheryl Ewy — June 1985

This Technical Note documents the Apple II Pascal 1.3 Disk Formatter routine.

Introduction

Integrating the Pascal Disk Formatter utility into your application program will free the user from having to format Pascal disks prior to running your program. Error codes that specify any problems encountered during the formatting process are returned. The disk contains the following files:

FORMATTER.TEXT is a sample Pascal host program that illustrates the use of the formatter routine.

FORMDISK.TEXT is an assembly language function that is linked to your Pascal host program. It contains the code to format disks in ProDOS blocked devices and calls the ASMFORMAT function to format disks in Disk II drives.

ASMFORMAT.TEXT is the Disk II formatter, an assembly language procedure that must be specially handled (see below).

BOOTTRACKS.DATA is a data file that is used to create the formatter data file. It contains boot blocks for both Disk II drives and ProDOS blocked devices and a blank disk directory.

MAKEFMT.TEXT, MAKEFMT.CODE are a Pascal program that will create the required formatter data file.

FORMATTER.DATA is a complete formatter data file (identical to that supplied with the Apple II Pascal 1.3 Development System).

FORMATTER.CODE is the formatter program supplied with the Apple II Pascal 1.3 Development System.

All programs are supplied in source (and where appropriate, as code files) so that you may modify them for your own particular purposes.

ASMFORMAT - The Disk II Formatter Routine

The file ASMFORMAT.TEXT contains a proprietary subroutine that performs the actual formatting of Disk II disks. It is written in 6502 assembly language suitable for assembly by the Apple II or Apple /// Pascal Assembler. This code requires special handling by the host program to ensure a reliable format. It contains critical timing code, and because of this, it must be located on a page boundary in memory (a location of the form xx00, e.g., 3D00, 2000, etc.). To do this, it must be assembled ABSOLUTE and you must use ORG to place it on particular page boundary. It comes supplied at location 3D00, which is the location used by the formatter routine supplied with the Apple II Pascal 1.3 Development System (FORMATTER.CODE). If you need to move it to another particular location you must change the .ORG statement in the file to the new address. The formatter will not work reliably if it is not on a page boundary. The code itself is 1082 bytes in length.

Because of the special nature of this code, it must be loaded by the Pascal host program at the chosen location. The following sample code illustrates how this is done:

```
    TYPE MEMARRAY = PACKED ARRAY [0..1535] OF 0..255;

        MEMPTR = RECORD CASE BOOLEAN OF
                TRUE:  (ADDR: INTEGER);
                FALSE: (MEM: ^MEMARRAY);
            END;

    VAR  LOADPTR: MEMPTR;     {this is the pointer to the absolute memory
                              location where the Disk II formatter routine
                              will be loaded.}


        {the following code will load the Disk II formatter routine
        from the formatter data file into memory at a fixed location}

        RESET(DATAFILE, '%FORMATTER.DATA');

        LOADPTR.ADDR := 15616;    {this value is the absolute memory location
                                  where the code is to be loaded.  In this
                                  example, 15316 is the decimal equivalent of
                                  the memory address 3D00.}

        BLOCKSREAD := BLOCKREAD(DATAFILE, LOADPTR.MEM^, 3);
{the above line will load three blocks (the Disk II formatter code) from the data
file into
the memory space specified in LOADPTR}
```

The Disk II formatter routine assumes that the A register has been setup with the slot number and drive number of the disk which is to be formatted. FORMDISK sets up this information before doing a JSR to the Disk II formatter routine. The contents of the A register are defined as follows:

Bit 7	Drive number. 0=Drive 1, 1=Drive 2
Bits 6-4	Slot number. 100=4, 101=5, 110=6. No other slots are supported.
Bits 3-0	Reserved; must be set to zero.

After the Disk II formatter routine is called, it returns an error code in the A register. FORMDISK then returns this error code to the host program. The error codes are listed in the following section.

FORMDISK - The Main Formatter Routine

The file FORMDISK.TEXT is an assembly language function that is assembled and linked to your Pascal host program. This function determines whether the drive containing the disk to be formatted is a Disk II drive or a ProDOS blocked device. If it is a Disk II drive, FORMDISK invokes the Disk II formatter routine with the required parameters as described in the previous section. If the drive is a ProDOS blocked device, FORMDISK sets up the proper parameters and executes a format call to the device. FORMDISK will return an error code back to the Pascal host after the formatting is complete. The call to this function is shown below:

```
     VAR  ERRCODE: INTEGER;            {the error code returned}
          VOLNUM:  INTEGER;            {the volume (unit) number of the disk}
     ERRCODE := FORMDISK(VOLNUM);      {the function call}
```

There are six possible error codes returned by FORMDISK. They indicate problems that may have occurred during the formatting process. They are as follows:

Error code	Error	Possible causes
00	No Error	Formatting successfully completed
39	Unable to format the disk	No disk in drive; drive door not closed; bad media
43	Disk is write-protected	Disk is write-protected; disk is pushed halfway into drive, activating the write-protect switch
47	No disk in drive	The disk drive is empty. This error is only reported for ProDOS block devices. If a Disk II drive is empty, error #39 is returned.
51	Drive speed is too slow	The drive motor speed requires adjustment, it is too slow. This erroris only reported for Disk IIs.
52	Drive speed is too fast	The drive motor speed requires adjustment, it is too fast. This error is only reported for Disk IIs.

To use the FORMDISK function requires that you modify one .EQU statement in the source file (FORMDISK.TEXT) to specify the location of the Disk II formatter routine in memory. Currently, the statement reads as follows:

```
    DO_FORMAT   .EQU   3D00   ;memory address of the Disk II formatter routine
```

If you decide to relocate the Disk II formatter routine, simply change this value to reflect the new memory address, then reassemble FORMDISK. The FORMDISK function does a JSR to this value to invoke the Disk II formatter routine.

Note: The value used in the .ORG in ASMFORMAT and the .EQU in FORMDISK must match.

Making a Formatter Data File

To use the formatter requires a data file that contains three pieces:

1. The Disk II formatter routine code, to be loaded into memory.
2. The boot code that is written to blocks 0 and 1 of the formatted disk.
3. A blank UCSD Pascal directory that is written to block 2 of the formatted disk.

The formatter disk comes with the second and third parts in the file BOOTTRACKS.DATA. This four-block file contains the boot blocks for Disk II drives and ProDOS blocked devices and the blank directory. Once the Disk II formatter routine has been assembled (to ASMFORMAT.CODE) it must be concatenated to the BOOTTRACKS.DATA file to make the formatter data file. The Disk II formatter routine code occupies the first 3 blocks of the formatter data file, which is then followed by the contents of the BOOTTRACKS.DATA file. Because the assembler puts special informational content blocks into a code file, a special program is required to copy only the blocks containing the code of the Disk II formatter routine. This is the program MAKEFMT.CODE. This program copies blocks 1, 2, and 3 of ASMFORMAT.CODE to blocks 0, 1, and 2 of the file FORMATTER.DATA. It then copies blocks 0, 1, 2, and 3 of the file BOOTTRACKS.DATA to blocks 3, 4, 5, and 6 of the file FORMATTER.DATA. This makes the required formatter data file (7 blocks in size) that will be used by the Pascal host program. MAKEFMT requires that the files ASMFORMAT.CODE and BOOTTRACKS.DATA be on the prefix volume. Set the Pascal prefix to this volume and X(ecute MAKEFMT. It will create the file FORMATTER.DATA on the same volume. The source for this program is included so that you may modify it as needed.

The Pascal Host Program

It is up to you to write the Pascal host program. On the disk is a sample program (the Apple II Pascal 1.3 Formatter) that you may study. It illustrates the above techniques. The primary functions of the Pascal host are to:

1. Open the FORMATTER.DATA file.
2. Read blocks 0 - 2 into a memory location that is on a page boundary.
3. Read blocks 3 - 6 into a 2,048 byte buffer.
4. Call the assembly language function FORMDISK with the volume number of the drive containing the disk to be formatted.
5. Examine the error code returned. If there is an error then report it to the user, otherwise continue.
6. Use UNITSTATUS to determine whether the drive is a Disk II or a ProDOS blocked device and how many blocks are on the disk.
7. Use the number of blocks returned by UNITSTATUS to update the maximum block number information in the blank directory.
8. If the drive is a Disk II, use UNITWRITE to write blocks 0 - 2 from the buffer to blocks 0 - 2 on the newly formatted disk.
9. If the drive is a ProDOS blocked device, use UNITWRITE to write block 3 from the buffer to block 0 on the newly formatted disk, then use it again to write block 2 from the buffer to block 2 on the disk.

The following code is an example of how to read in the blocks from the FORMATTER.DATA file, determine the drive type, update the directory, and write the boot blocks and directory to the newly formatted disk:

```
TYPE BYTARRAY = PACKED ARRAY [0..1] OF 0..255;

VAR  BUFFER: PACKED ARRAY [0..2048] OF 0..255;

NUMBLOCKS : INTEGER;

TRIX : RECORD CASE BOOLEAN OF
            TRUE  : (INT : INTEGER);
            FALSE : (BYT : BYTARRAY);
        END;

{read in the boot blocks and directory}
NUMBLOCKS := BLOCKREAD (DATAFILE, BUFFER, 4, 3);

{determine type of disk drive and number of blocks on the disk}
UNITSTATUS (VOLNUM, NUMBLOCKS, 1);

{update maximum number of blocks in blank directory}
TRIX.INT := NUMBLOCKS;
BUFFER[1038] := TRIX.BYT[0];
BUFFER[1039] := TRIX.BYT[1];

{write out the boot blocks and directory to a Disk II disk}
UNITWRITE (VOLNUM, BUFFER, 1536, 0);

{write out the boot block and directory to a ProDOS blocked device disk}
UNITWRITE (VOLNUM, BUFFER[1536], 512, 0);
UNITWRITE (VOLNUM, BUFFER[1024], 512, 2);
```

A dynamic variable can also be used as the buffer so that your program can reclaim the buffer space for its own use after the formatting is completed:

```
TYPE BUFFER = PACKED ARRAY [0..2048] OF 0..255;

VAR BUFPTR : ^BUFFER;
OLDPTR : ^INTEGER;

 {mark the beginning of usable space}
MARK (OLDPTR);
 {allocate space for the buffer}
NEW (BUFPTR);
 {read in the boot blocks and directory}
NUMBLOCKS := BLOCKREAD (DATAFILE, BUFPTR^, 4, 3);
{write out the boot blocks and directory to a Disk II disk}
UNITWRITE (VOLNUM, BUFPTR^, 1536, 0);
{release the space used by the buffer}
RELEASE (OLDPTR);
```

In Review

The following is a step-by-step review of how to use the formatting routine.

1. Determine where in memory you wish to load the Disk II formatter routine. Remember it must be on a page boundary.
2. Edit the file ASMFORMAT.TEXT, and change the value in the .ORG statement to be the memory address chosen.
3. Assemble ASMFORMAT.TEXT to ASMFORMAT.CODE.
4. X(ecute MAKEFMT to make the required FORMATTER.DATA file.
5. Edit the file FORMDISK.TEXT and change the line

```
DO_FORMAT   .EQU   3D00
```

   to reflect the new memory location (same value as in the .ORG statement above).
6. Assemble FORMDISK.TEXT to FORMDISK.CODE.
7. Write the Pascal host program using the above techniques for loading the Disk II formatter routine, calling the FORMDISK function, updating the blank directory, and writing the boot blocks and directory. Remember error reporting.
8. Compile the Pascal host.
9. Link the Pascal host to the file FORMDISK.CODE, thus linking the FORMDISK function into your program.
10. With the linked Pascal host program and the FORMATTER.DATA file you can now format disks.

END OF FILE TN.Pasc.012

```
#####################################################################
### FILE: TN.Pasc.014
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Pascal
#14: Apple Pascal 1.3 TREESEARCH and IDSEARCH

Revised by: Cheryl Ewy November 1988
Written by: Cheryl Ewy June 1985

This Technical Note describes the TREESEARCH and IDSEARCH routines which were built into Pascal 1.2 and earlier, but which are separate entities for Pascal 1.3.

Introduction

In Apple II Pascal versions 1.0 through 1.2, TREESEARCH and IDSEARCH were special-purpose built-in routines which could be called from within a Pascal program. The routines existed primarily for use by the Compiler and Libmap but were also available for use by applications. In Apple II Pascal 1.3, the routines were removed due to space requirements. Since some applications use these routines, they are being supplied as 6502 codefiles which can be linked to Pascal programs. To use the routines, the Pascal host program must declare them as EXTERNAL (see the following sections for details). After compiling the host program, use the Linker to link the file TRS.CODE (TREESEARCH) or the file IDS.CODE (IDSEARCH).

The TREESEARCH Function

TREESEARCH is a fast assembly language function for searching a binary tree with a particular kind of structure. The external declaration is:

```
    FUNCTION TREESEARCH (ROOTPTR : ^NODE;
                        VAR NODEPTR : ^NODE;
                        NAMEID : PACKED ARRAY [1..8] OF CHAR) :INTEGER;
        EXTERNAL;
```

The call syntax is:

```
    RESULT := TREESEARCH (ROOTPTR, NODEPTR, NAMEID);
```

where ROOTPTR is a pointer to the root node of the tree to be searched, NODEPTR is a reference to a pointer variable to be updated by TREESEARCH, and NAMEID contains the eight-character name to be searched for in the tree.

The nodes in the binary tree are assumed to be linked records of the type:

```
NODE = RECORD
```

```
            NAME : PACKED ARRAY[1..8] OF CHAR;
            LEFTLINK, RIGHTLINK : ^NODE;

            {other fields can be anything}

     END;
```

The actual names of the type and the field identifiers are not important; TREESEARCH assumes only that the first eight bytes of the record contain an eight-character name and are followed by two pointers to other nodes.

It is also assumed that names are not duplicated within the tree and are assigned to nodes according to an alphabetical rule; for a given node, the name of the left subnode is alphabetically less than the name of the node, and the name of the right subnode is alphabetically greater than the name of the node. Finally, any links that do not point to other nodes should be NIL.

TREESEARCH can return any of three values:

- 0 The name passed to TREESEARCH (as the third parameter) has been found in the tree. The node pointer (second parameter) now points to the node with the specified name.
- 1 The name is not in the tree. If it is added to the tree, it should be the right subnode of the node pointed to by the node pointer.
- -1 The name is not in the tree. If it is added to the tree, it should be the left subnode of the node pointed to by the node pointer.

The TREESEARCH function does not perform any type checking on the parameters passed to it.

The IDSEARCH Procedure

IDSEARCH is a fast assembly language procedure that scans Apple II Pascal source text for identifiers and reserved words. Note that IDSEARCH recognizes only identifiers and reserved words--you have to scan for special characters and comments yourself.

The external declaration is:

```
    PROCEDURE IDSEARCH (VAR OFFSET:INTEGER;
                       VARBUFFER:BYTESTREAM);
        EXTERNAL;
```

The call syntax is:

```
    IDSEARCH (OFFSET, BUFFER);
```

To use IDSEARCH, you must include the following declarations in your program. Note that the variables (except BUFFER) must be declared in exactly the order and types shown.

```
TYPE

    {SYMBOL is the enumerated type of symbols in the Apple // Pascal
language}
```

```
    SYMBOL =    (IDENT,COMMA,COLON,SEMICOLON,LPARENT,RPARENT,DOSY,TOSY,
                DOWNTOSY,ENDSY,UNTILSY,OFSY,THENSY,ELSESY,BECOMES,LBRACK,
                RBRACK,ARROW,PERIOD,BEGINSY,IFSY,CASESY,REPEATSY,WHILESY,
                FORSY,WITHSY,GOTOSY,LABELSY,CONSTSY,TYPESY,VARSY,PROCSY,
                FUNCSY,PROGSY,FORWARDSY,INTCONST,REALCONST,STRINGCONST,
                NOTSY,MULOP,ADDOP,RELOP,SETSY,PACKEDSY,ARRAYSY,RECORDSY,
                FILESY,OTHERSY,LONGCONST,USESSY,UNITSY,INTERSY,IMPLESY,
                EXTERNLSY,OTHERWSY);

    {The reserved words corresponding to the above symbols are as follows -

    DOSY        - DO        WITHSY      - WITH         RELOP       - IN
    TOSY        - TO        GOTOSY      - GOTO         SETSY       - SET
    DOWNTOSY    - DOWNTO    LABELSY     - LABEL        PACKEDSY    - PACKED
    ENDSY       - END       CONSTSY     - CONST        ARRAYSY     - ARRAY
    UNTILSY     - UNTIL     TYPESY      - TYPE         RECORDSY    - RCORD
    OFSY        - OF        VARSY       - VAR          FILESY      - FILE
    THENSY      - THEN      PROCSY      - PROCEDURE    USESSY      - USES
    ELSESY      - ELSE      FUNCSY      - FUNCTION     UNITSY      - UNIT
    BEGINSY     - BEGIN     PROGSY      - PROGRAM      INTERSY     -.INTERFACE
    IFSY        - IF                      SEGMENT      IMPLESY     -.IMPLEMENTATION
    CASESY      - CASE      FORWARDSY   - FORWARD      EXTERNLSY   - EXTERNAL
    REPEATSY    - REPEAT    NOTSY       - NOT          OTHERWSY    - OTHERWISE
    WHILESY     - WHILE     MULOP       - AND,DIV,MOD
    FORSY       - FOR       ADDOP       - OR           }

    {OPERATOR expands the multiplicative (MULOP), additive (ADDOP) and
    relational (RELOP) operators}

    OPERATOR =     (MUL,RDIV,ANDOP,IDIV,IMOD,PLUS,MINUS,OROP,LTOP,LEOP,
                   GEOP,GTOP,NEOP,EQOP,INOP,NOOP);

    ALPHA = PACKED ARRAY [1..8] OF CHAR;

VAR
    {the next four variables must be declared in the order shown}
    OFFSET : INTEGER;
    SY : SYMBOL;
    OP : OPERATOR;
    ID : ALPHA;
```

IDSEARCH begins by looking for an identifier at the character pointed to by BUFFER[OFFSET] and assumes that this character is alphabetic. IDSEARCH produces the following results:

- BUFFER[OFFSET] points to the character following the identifier just found.
- ID contains the first eight alphanumeric characters of the identifier just found, left justified and padded with spaces as necessary.
- SY contains the symbol associated with the identifier just found if the identifier is a reserved word or IDENT if the identifier is not a reserved word. For example, the identifier MOD translates to MULOP; the identifier ARRAY translates to ARRAYSY, and the identifier MYLABEL translates to IDENT.
- OP contains the operator value which corresponds to the identifier just found if the identifier is an operator, or NOOP if the

identifier is not an operator. For example, the identifier MOD translates to IMOD, the identifier ARRAY translates to NOOP, and the identifier MYLABEL translates to NOOP.

The following is an example of calling IDSEARCH:

```
    BEGIN
        IF BUFFER[OFFSET] IN ['A'..'Z','a'..'z'] THEN
            IDSEARCH (OFFSET, BUFFER);
    END;
```

The following is an algorithmic representation of IDSEARCH:

```
    PROCEDURE IDSEARCH (VAR OFFSET:INTEGER; VAR BUFFER:BYTESTREAM);
    BEGIN
        ID := ScanIdentifier (OFFSET, BUFFER);
        SY := LookUpReservedWord (ID);
        OP := LookUpOperator (ID);
    END;
```

ScanIdentifier increments OFFSET until BUFFER[OFFSET] is no longer part of an identifier, copying the first eight alphanumeric characters passed into ID (left justified, padding with spaces).

LookUpReservedWord translates an identifier into the associated symbol (defaulting to IDENT).

LookUpOperator translates an identifier into the associated operator (defaulting to NOOP).

END OF FILE TN.Pasc.014

##
FILE: TN.Pasc.015
##

Apple II
Technical Notes

Developer Technical Support

Pascal
#15: Apple II Pascal SHORTGRAPHICS Module

Revised by: Cheryl Ewy & Dan Strnad November 1988
Written by: Cheryl Ewy December 1983

This Technical Note describes the Apple II Pascal SHORTGRAPHICS routine, which is available as part of the 48K Run-Time System.

Introduction

Many applications, especially those designed to use the 48K Run-Time System, run out of memory quickly if they use the TURTLEGRAPHICS unit provided with the standard SYSTEM.LIBRARY.

This document describes a library unit called SHORTGRAPHICS which removes the relative polar coordinate features of TURTLEGRAPHICS to save memory.

General Comments

If your application uses (or can be modified to use) only those TURTLEGRAPHICS procedures which refer to absolute screen coordinates, you can use the SHORTGRAPHICS unit. The SHORTGRAPHICS unit has the same segment numbers assigned to it, as does TURTLEGRAPHICS, thus you may not use both in the same program.

Deletions

The following routines are not available in the SHORTGRAPHICS unit:

```
    PROCEDURE TURN(ANGLE: INTEGER);
    PROCEDURE TURNTO(ANGLE: INTEGER);
    PROCEDURE MOVE(DIST: INTEGER);
    FUNCTION  TURTLEANG: INTEGER;
```

Additions

The following definitions have been added to the INTERFACE section of SHORTGRAPHICS:

```
    TYPE
        FONT=PACKED ARRAY[0..127,0..7] OF 0..255;
```

```
    VAR
        FONTPTR:^FONT;
```

The variable FONTPTR is a pointer to the memory area used by the WCHAR and WSTRING procedures to display text on the graphics screen.

Thus, if you have a character set named KATAKANA.FONT, you could load it into memory and use it as follows:

```
    VAR
        SPECIALFONT:^FONT;         (* where the new font goes *)
        SAVEFONT:^FONT;            (* to save pointer to standard font area *)

        PROCEDURE LOADFONT;
        VAR
            F:FILE;
            NIO:INTEGER;
        BEGIN
            NEW(SPECIALFONT);
            RESET(F,'KATAKANA.FONT');
            NIO:=BLOCKREAD(F,SPECIALFONT^,2,0);
            CLOSE(F)
        END;

        PROCEDURE USESPECIAL;
        BEGIN
            SAVEFONT:=FONTPTR;    (* save standard font pointer *)
            FONTPTR:=SPECIALFONT; (* and point to special font *)
        END;

        PROCEDURE USENORMAL;
        BEGIN
            FONTPTR:=SAVEFONT     (* restore pointer to normal font *)
        END;
```

Memory Considerations

When the system is booted, the heap pointer is normally below the start of high-resolution page one. The TURTLEGRAPHICS unit automatically sets the heap pointer above high-resolution page one. This protects the high-resolution page from being overwritten by your program, but it also prevents you from using the space between the original top of the heap and the start of high-resolution page one for your own variables.

SHORTGRAPHICS does not protect the high-resolution page, thus you may use this extra space for yourself. The following code will check to see if you have n bytes available between the top of the heap and high-resolution page one. If the room is not available, the heap pointer will be jumped to the top of the high-resolution page.

```
    PROCEDURE MAKEROOM(N:INTEGER);
    CONST
        BOTTOM=8192;

        TOP=16384;
    VAR
        CHEAT:RECORD CASE BOOLEAN
```

```
                TRUE:(IPART:INTEGER);
                FALSE:(PPART:^INTEGER);
            END;
    BEGIN
        MARK(CHEAT.PPART);
        IF (CHEAT.IPART+N)>=BOTTOM THEN BEGIN
            CHEAT.IPART:=TOP;
            RELEASE(CHEAT.PPART)
        END
    END;
```

Thus, if you wanted to allocate a special font (which requires 1,024 bytes) below the high-resolution page, you could use this code:

```
    MAKEROOM(1024);
    NEW(SPECIALFONT);
```

If there are at least 1,024 bytes beneath the high-resolution page, the new font will be allocated there. If there is not enough space there, the new font will be allocated above the high-resolution page.

All of these heap allocations should be done as the very first actions of your program. When you finish allocating your variables, you should invoke the following procedure to make sure the heap pointer is above high-resolution page one (thus protecting it).

```
    PROCEDURE PROTECT;
    CONST
        TOP=16384;
    VAR
        CHEAT:RECORD CASE BOOLEAN OF
                    TRUE:(IPART:INTEGER);
                    FALSE:(PPART:^INTEGER);
                END;
    BEGIN
        MARK(CHEAT.PPART);
        IF CHEAT.IPART<TOP THEN BEGIN
            CHEAT.IPART:=TOP;
            RELEASE(CHEAT.PPART);
        END;
    END;
```

Warning: Every program written using SHORTGRAPHICS is unprotected from a heap which grows large enough to go into the high-resolution page one area. Therefore, every program using SHORTGRAPHICS should protect page one by using PROCEDURE PROTECT. You should protect page one even if the program does not use the space below it.

Code Length

When you look at TURTLEGRAPHICS with the LIBRARY program, the code segment has a length of 5,230 bytes and the data segment a length of 386 bytes. SHORTGRAPHICS has a code segment 3,140 bytes in length and a data segment of 18 bytes. Thus, in a test of a null program, you have 2,458 bytes more space available.

Files on the Disk

The following files are on the SHORT GRAPHICS disk:

SHORT.CODE	Contains the SHORTGRAPHICS code. You can use it as a library or use the library program to add it to SYSTEM.LIBRARY in place of TURTLEGRAPHICS.
KATAKANA.FONT	A sample font used to demonstrate the use of alternate fonts.
SYSTEM.CHARSET	The letters in this character set are not as wide as those found in the SYSTEM.CHARSET supplied with the Development System and the Run-Time Systems.
TEST.TEXT, TEST.CODE	A sample program showing some of the concepts discussed in this Technical Note.

Interface Listing of the SHORTGRAPHICS Unit:

The following is the interface section of the SHORTGRAPHICS unit:

```
UNIT SHORTGRAPHICS; INTRINSIC CODE 20  DATA 21;

    INTERFACE
        TYPE


SCREENCOLOR=(none,white,black,reverse,radar,black1,green,violet,white1,
                         black2,orange,blue,white2);

            FONT=PACKED ARRAY[0..127,0..7] OF 0..255;

        VAR
            FONTPTR:^FONT;

        PROCEDURE INITTURTLE;
        PROCEDURE MOVETO(X,Y: INTEGER);
        PROCEDURE PENCOLOR(PENMODE: SCREENCOLOR);
        PROCEDURE TEXTMODE;
        PROCEDURE GRAFMODE;
        PROCEDURE FILLSCREEN(FILLCOLOR: SCREENCOLOR);
        PROCEDURE VIEWPORT(LEFT,RIGHT,BOTTOM,TOP: INTEGER);
        FUNCTION  TURTLEX: INTEGER;
        FUNCTION  TURTLEY: INTEGER;
        FUNCTION  SCREENBIT(X,Y: INTEGER): BOOLEAN;
        PROCEDURE DRAWBLOCK(VAR SOURCE; ROWSIZE,XSKIP,YSKIP,WIDTH,
                            HEIGHT,XSCREEN,YSCREEN,MODE:
                            INTEGER);
        PROCEDURE WCHAR(CH: CHAR);
        PROCEDURE WSTRING(S: STRING);
        PROCEDURE CHARTYPE(MODE: INTEGER);
```

END OF FILE TN.Pasc.015

```
#####################################################################
### FILE: TN.Pasc.016
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Pascal
#16: Driver to Have Two Volumes on One 3.5" Disk

Revised by: Guillermo Ortiz, Cheryl Ewy & Dan Strnad November 1988
Written by: Guillermo Ortiz October 1986

This Technical Note discusses how to install a driver to have more than one volume on a 3.5" 800K disk under Apple II Pascal.

For the sake of simplicity,.we will limit the discussion to the following case: we want to have two 400K volumes on the boot 3.5" disk. For such a scenario, Unit #4 occupies the first 800 blocks and Unit #20 uses blocks 800 to 1599 as shown here:

```
           First Volume Unit #4                 Second Volume Unit #4
 __|_________________________________|___________________________________|__
   |       Blocks (0 .. 799)         |       Blocks (800 .. 1599)        |


     +-- Directory Unit #4                +-- Directory Unit #20
     |   blocks (2 .. 5)                  |   blocks (802 .. 805)
     |                                    |
 ___________________________________________________________________________
 | |   |                             | |   |                             |
 |_|___|_____________________________|_|___|_____________________________|__
 | \                                 | \
 |  \ Boot Blocks (0 .. 1)           |  \ Pseudo Boot Blocks (800 .. 801)
```

Figure 1-Block Diagram for 3.5" Disk

There are four calls a device driver has to handle, UNITCLEAR, UNITSTATUS, UNITREAD, and UNITWRITE. For the first one, our driver will only return since the device is already on-line. For a blocked device, UNITSTATUS returns the number of blocks available, in this case UNITSTATUS (20) = 800.

In the case of UNITREAD and UNITWRITE, all the driver has to do is add the offset of 800 to the number of the block requested then jump to the BIOS routine with the unit number set to four. Our driver is basically a dispatcher that directs the disk access to the proper blocks.

When this driver is present, the application must be very careful about making sure the right disk is in the drive when accessing the second volume; any access to Unit #20 could damage a normal volume present in the drive.

Once the driver is ready, it is necessary to format a disk with the special directories. With the listings for the driver we have included the source of a sample formatting program.

Once the disk is ready we proceed to transfer all system files to it including SYSTEM.ATTACH, ATTACH.DRIVERS (containing our driver), and ATTACH.DATA. This last file reflects the following information:

```
    Driver Name - FAKEDISK  - Not Aligned
    Attached to #20                                      {Can change if desired}
    Unit #s to be init at boot time - 20
    This driver CAN be placed in the first HiRes screen {Change if needed}
    This driver CAN be placed in the second HiRes screen{Change if needed}
    This driver does not use interrupts
    Driver does not have transient initialization code
```

The code has comments that explain it fairly well; for more information on drivers in general and how to use the attach tools please refer to Apple II Pascal Device and Interrupt Support Tools.

```
;
;       Disk Driver
;       by Guillermo Ortiz
;       03/25/86
;
;       This driver will allow splitting a 3.5 disk in two pieces of 400K
;       each, therefore permitting more than 77 files per disk. It
;       is required to "format" the disk with two directories, one at
;       block 0 .. 5 and the other at block 800 .. 805, each with a
;       length of 800 blocks. Names must be different!

;       The ancient admonition:
;
;       This is a sample!
;       No claims are made regarding the fitness of this code for
;       any particular purpose.

ROUTINE .EQU    02              ; For indirect jumping
RETURN  .EQU    04              ; Back to Pascal
BUFF    .EQU    06              ; Where to put stuff

        .PROC   FAKEDISK

;       At this level we could have some code to differentiate
;       between different pseudo volumes if we had more than
;       two pseudo-volumes per disk.
;       In this example we use Unit # 20 for the second part.
;       Using units 13 and up let us keep the "standard" drives available
;       In any UNIT call X Register contains the type of call
;       as follows:

        CPX     #04
        BEQ     STATUS          ; X = 4
        CPX     #02
        BEQ     INIT            ; X = 2

        STA     TEMP1
        STY     TEMP1+1         ; Saving A, Y and X
        STX     TEMP1+2         ;   for future use
```

```
;       We make the assumption that the disk split is the
;       System Volume, so we get the logical volume number for
;       Unit # 4 from the DISKNUM table;
;       see Apple // Pascal Device and Interrupt
;       Support Tools manual for details.

        TSX                     ; Gimmie the stack pointer
        LDA     0FEB6           ; Logical volume for boot disk
        STA     109,X           ;   so read from that disk

;       Our fiddling is complete now let's finish checking
;       the call in order to make the jump

        LDA     TEMP1+2         ; X contains the call code
        BEQ     READ            ; X = 0
        CMP     #01
        BEQ     WRITE           ; X = 1

;       Here we could have
;       instructions to report some undefined control code.
;       This driver will only CRASH!!!

        BRK                     ; Bumm!!!

;       Now the real stuff

READ    .EQU    *
        JSR     SETUP           ; Modify the stack
        LDY     #19.            ; Index for Reading from disk
        BNE     GET             ; Nice way of jumping

WRITE   .EQU    *
        JSR     SETUP           ; Modify the stack
        LDY     #16.            ; Index for WRITE to CONSOLE

GET     LDA     @0E2,Y          ; $E2 contains a pointer to the jump vector
        STA     ROUTINE         ; Set low byte of address
        INY
        LDA     @0E2,Y          ; Get high byte of address
        STA     ROUTINE+1       ;   and set it off

        LDX     TEMP1+2         ; Restore
        LDY     TEMP1+1         ;   all registers
        LDA     TEMP1           ;     before jump

        JMP     @ROUTINE        ; and Go!



;       INIT will only pass back the no_error IORESULT

INIT    .EQU    *
        LDX     #00             ; No error
        RTS                     ;   Go back

STATUS  PLA                     ; Get
        STA     RETURN          ;   return
        PLA                     ;     address
```

```
        STA     RETURN+1
        PLA                     ; Get
        STA     BUFF            ;   Pascal
        PLA                     ;     Buffer
        STA     BUFF+1          ;       address
        PLA                     ; Dump control
        PLA                     ;   word
        LDY     #00
        LDA     #20             ; Set
        STA     @BUFF,Y         ;   the number of blocks
        INY                     ;     to
        LDA     #03             ;       800
        STA     @BUFF,Y
        LDX     #00
        LDA     RETURN+1        ; and
        PHA
        LDA     RETURN
        PHA
        RTS                     ;   Return!


;       To any request for READ/WRITE we'll add 800 to the
;       number of the block needed.

SETUP   .EQU    *
        LDA     103,X           ; Get Block number low
        CLC                     ; Set up for addition
        ADC     #20             ;   Offset block count by 800
        STA     103,X           ;     and restore
        LDA     104,X           ; Get Block number high
        ADC     #03             ;   800 = $320
        STA     104,X           ;     and restore
        RTS                     ; Go back

TEMP1   .BLOCK  3               ; Temporary storage area


        .END
```

The driver requires that the disk be formatted in a special way. Run the following program to create your volume.

```
program REFORMAT;

{By Guillermo Ortiz
     03/27/86
}


{This program takes a newly formatted 3.5 disk and lays down two
 directories transforming the volume into two 400K pseudo-volumes to be
 used with the driver FAKEDISK which assigns Unit # 20 to the second
 part of the disk.
}


CONST   MAXDIR  = 77;   {Max number of files per volume}
        VIDLENGTH = 7;  {Max chars in volume name}
```

```
        TIDLENGTH = 15; {Max chars per file ID}
        FBLKSIZE = 512; {Number of bytes per block}
        DIRBLK = 2;     {We are reading the directory}

type    daterec = packed record
                    month:0..12;          {0 --> Meaningless date}
                    day:  0..31;          {Day of month}
                    year:0..100           {100 --> dated volume is temp}
                  end;

        vid = string [vidlength];         {Volume ID}
        dirrange = 0 .. maxdir;           {Number of files on disk}
        tid = string[tidlength];          {File ID}
        filekind = (untypedfile,xdskfile,codefile,textfile,infofile,
                    datafile,graffile,fotofile,securdir);

{Now the real directory layout}
        direntry =
          packed record
            dfirstblk:integer;            {1st physical disk address}
            dlastblock:integer;           {block after last used block}
            case dfkind:filekind of
              securdir,untypedfile:       {Volume info only in dir[0]}
               (filler1: 0..2048;         {Waste 13 bits}
                dvid:    vid;             {Name of volume}
                deovblk: integer;         {Last block in volume}
                dnumfiles:dirrange;       {Number of files in directory}
                dloadtime:integer;        {Time of last access}
                dlastboot:daterec);       {Most recent date setting}
              xdskfile,codefile,textfile,infofile,datafile,
              graffile,fotofile:          {Regular file info}
               (filler2: 0..1024;         {Waste 12 bits}
                status:  boolean;         {For filer wildcards}
                dtid:    tid;             {Name of file}
                dlastbyte:1..fblksize;    {Bytes in last block of file}
                daccess: daterec)         {Date of last modification}
          end;  {Of the whole directory record}

        directory = array [dirrange] of direntry;


var     dirinfo:directory;                {The directory goes here}
        UNITNUM:INTEGER;
        CH:CHAR;


PROCEDURE DOSTUFF;
{Function CHECK will read the directory from a freshly formatted
 3.5 disk, then DOSTUFF will make changes so it has only 800 blocks and
 a name HALFONE: and will write it back to block 2; then we will
 change the name to HALFTWO: and will write to block 802 as
 the directory for our second pseudo-volume.
}

BEGIN
  with dirinfo[0] do
    begin
      deovblk:=800;  {Cut it in half}
```

```
      dvid:='HALFONE';
    end;
  unitwrite(UNITNUM,dirinfo,sizeof(dirinfo),dirblk); {Put back main directory}
  DIRINFO[0].DVID:='HALFTWO';
  unitwrite(UNITNUM,dirinfo,sizeof(dirinfo),dirblk+800) {Write second dir.}
end; {Of DOSTUFF}


FUNCTION CHECK:BOOLEAN;

{Reads the directory from the target disk, if possible, warns the user
 of the certain destruction of the current directory and checks the
 size of the volume so that the program doesn't use other than 3.5
 disks.
 }


BEGIN
  CHECK:=FALSE;
  DIRINFO[0].DLASTBLOCK:=-999;  {Make sure we read from a disk}
  UNITREAD(UNITNUM,DIRINFO,SIZEOF(DIRINFO),DIRBLK);
  IF DIRINFO[0].DLASTBLOCK= 6 THEN  {IS THIS A PASCAL DISK?}
    BEGIN
      IF DIRINFO[0].DEOVBLK <> 1600 THEN
        BEGIN
          WRITELN('SORRY THIS PROGRAM IS INTENDED FOR 3.5 DISKS ONLY');
          EXIT(CHECK)
        END;
      WRITE('WE ARE ABOUT TO PERMANENTLY DESTROY    ');
      WRITELN(DIRINFO[0].DVID,':');
      WRITE('IS IT OK? --> ');
      REPEAT
        READ(KEYBOARD,CH)
      UNTIL CH IN ['Y','N','n','y'];
      WRITELN(CH);
      IF CH IN ['Y','y'] THEN
        CHECK:=TRUE
    END
  ELSE
    BEGIN
      WRITELN;
      WRITELN;
      WRITELN('CAN NOT READ DIRECTORY')
    END
END {OF CHECK};


PROCEDURE GETNUM;

{Prompts the user for the Unit Number of the target disk,
 checks the validity of the input and returns when provided with
 a reasonable value.
 }

VAR     I:INTEGER;

BEGIN
  WRITELN;
```

```
  WRITELN('PLEASE ENTER THE NUMBER OF THE UNIT CONTAINING THE DISK');
  WRITE('TO BE REFORMATTED (PRESS <ESCAPE> TO EXIT) --> ');
  UNITNUM:=0;
  REPEAT
    BEGIN
      WRITE(CHR(5));    {Cursor ON}
      READ(CH);         {For the prompt}
      WRITE(CHR(6));    {and then OFF for speed and elegance(?)}
      IF EOLN THEN
        IF (UNITNUM IN [4,5,9..12]) THEN
          EXIT(GETNUM)
        ELSE
          FOR I:= 1 TO 32 - UNITNUM DO  {Kind of crude but ...}
            WRITE(CHR(8));              {to go back to the same place}
      IF ORD(CH) = 27 THEN
        BEGIN
          WRITELN;
          WRITELN('YOU ASKED FOR IT!!!');
          WRITE(CHR(5));        {Turn cursor ON before we exit}
          EXIT(PROGRAM)
        END;
      IF (ORD(CH) = 8) AND (UNITNUM > 0) THEN
        BEGIN
          IF UNITNUM < 10 THEN
            UNITNUM:=0
          ELSE
            UNITNUM:=UNITNUM DIV 10;
          WRITE(CHR(8),' ',CHR(8))      {To delete previous entry}
        END
      ELSE
        BEGIN
          IF (UNITNUM = 0) AND (CH IN ['1','4','5','9']) THEN
            UNITNUM:=ORD(CH)-ORD('0')
          ELSE
            IF (UNITNUM=1) AND (CH IN ['0','1','2']) THEN
              UNITNUM:=10*UNITNUM+ORD(CH)-ORD('0')
            ELSE
              IF ORD(CH) > 31 THEN
                WRITE(CHR(8),' ',CHR(8))        {Unwanted stuff,so ...}
        END                                     {get rid of it.       }
    END
  UNTIL FALSE;                                  {No Exit here.}
  WRITELN
END {OF GETNUM};


BEGIN                                   {main}
  WRITELN;
  WRITELN;
  WRITELN('WE ARE ABOUT TO REFORMAT A VOLUME SO IT WILL CONTAIN TWO');
  WRITELN('400K PSEUDO-VOLUMES. MAKE SURE YOU MARK THE DISK CLEARLY');
  WRITELN('SO YOU DON''T FORGET');
  WRITELN;
  WRITELN;
  REPEAT
    GETNUM
  UNTIL CHECK;
  DOSTUFF;
```

```
  WRITE(CHR(5));                {Don't forget to turn cursor ON}
  writeln;
  WRITELN('AWAAAAAY!!!')
end.
```

If two volumes are not enough, you can modify this example to support more than two per disk; the key is to keep in mind that when the call comes to the driver, the accumulator contains the number of the Unit the for which the call is intended. After checking this number the driver could decide what offset it has to add to access the correct volume.

Of course the formatter program would have to change accordingly, laying down the directories for the new volumes with the appropriate names and sizes.

The same scheme can be applied to any device that Pascal can directly recognize (i.e., the Apple Memory Expansion Card, ProFile hard disk, etc.).

Further Reference
o   Apple II Pascal Device and Interrupt Support Tools

END OF FILE TN.Pasc.016

##
FILE: TN.PDOS.001
##

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#1: The GETLN Buffer and a ProDOS Clock Card

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note describes the effect of a clock card on the GETLN buffer.

ProDOS automatically supports a ThunderClock(TM) or compatible clock card when the system identifies it as being installed. When programming under ProDOS, always consider the impact of a clock card on the GETLN input buffer ($200 - $2FF). ProDOS can support other clocks which may also use this space.

When ProDOS calls a clock card, the card deposits an ASCII string in the GETLN input buffer in the form: 07,04,14,22,46,57. This string translates as the following:

    07 = The month, July (01=Jan,...,12=Dec)
    04 = The day of the week, Thurs.(00=Sun,...,06=Sat)
    14 = The date (00 to 31)
    22 = The hour, 10 p.m. (00 to 23)
    46 = The minute (00 to 59)
    57 = The second (00 to 59)

ProDOS calls the clock card as part of many of its routines. Anything in the first 17 bytes of the GETLN input buffer is subject to loss if a clock card is installed and is called.

In general, it has been the practice of programmers to use the GETLN input buffer for every conceivable purpose. Therefore, an application should never store anything there. If your application has a future need to know about the contents of the $200 - $2FF space, you should transfer it to some other location to guarantee that it will remain intact, particularly under ProDOS, where a clock card may regularly be overwriting the first 17 bytes.

The ProDOS 8 Technical Reference Manual contains more information on the clock driver, including the necessary identification bytes, how the ProDOS driver calls the card, and how you may replace this routine with your own.

Further Reference
o ProDOS 8 Technical Reference Manual

END OF FILE TN.PDOS.001

###
FILE: TN.PDOS.002
###

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#2: Porting DOS 3.3 Programs to ProDOS and BASIC.SYSTEM

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note formerly described the DOSCMD vector of BASIC.SYSTEM.

This Note formerly described the DOSCMD vector of BASIC.SYSTEM, which can be used to let BASIC.SYSTEM interpret ASCII strings as disk commands in much the same way DOS 3.3 did. The ProDOS 8 Technical Reference Manual now contains this information in Appendix A.

Further Reference
- ProDOS 8 Technical Reference Manual

END OF FILE TN.PDOS.002

###
FILE: TN.PDOS.003
###

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#3: Device Search, Identification, and Driver Conventions

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note formerly described how ProDOS 8 searches for devices and how it deals with devices which are not Disk II drives.

This Note formerly described how ProDOS 8 searches for devices and how it deals with devices which are not Disk II drives; this information is now contained in section 6.3 of the ProDOS 8 Technical Reference Manual.

Note: The information on slot mapping on page 113 of the manual and on page 2 of the former edition of this Technical Note is incorrect. ProDOS itself will mirror SmartPort devices from slot 5 into slot 2 under certain conditions. Devices should not be mirrored into slots other than slot 2. For more information, see ProDOS 8 Technical Note #20, Mirrored Devices and SmartPort.

Further Reference
o ProDOS 8 Technical Reference Manual
o ProDOS 8 Technical Note #20, Mirrored Devices and SmartPort

END OF FILE TN.PDOS.003

###
FILE: TN.PDOS.004
###

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#4: I/O Redirection in DOS and ProDOS

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note discusses I/O redirection differences between DOS 3.3 and ProDOS.

Under DOS 3.3, all that is necessary to change the I/O hooks is installing your I/O routine addresses in the character-out vector ($36-$37) and the key-in vector ($38-$39) and notifying DOS (JSR $3EA) to take your addresses and swap in its intercept routine addresses.

Under ProDOS, there is no instruction installed at $3EA, so what do you do?

You simply leave the ProDOS BASIC command interpreter's intercept addresses installed at $36-$39 and install your I/O addresses in the global page at $BE30-$BE33. The locations $BE30-$BE31 should contain the output address (normally $FDF0, the Monitor COUT1 routine), while $BE32-$BE33 should contain the input address (normally $FD1B, the Monitor KEYIN routine).

By keeping these vectors in a global page, a special routine for moving the vectors is no longer needed, thus, no $3EA instruction. You install the addresses at their destination yourself.

If you intend to switch between devices (i.e., the screen and the printer), you should save the hooks you find in $BE30-$BE33 and restore them when you are done. Blindly replacing the values in the global page could easily leave you no way to restore input or output to the previous device when you are done.

END OF FILE TN.PDOS.004

```
########################################################################
### FILE: TN.PDOS.005
########################################################################
```

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#5: ProDOS Block Device Formatting

Revised by: Matt Deatherage — November 1988
Revised by: Pete McDonald — October 1985

This Technical Note formerly described the ProDOS FORMATTER routine.

The ProDOS 8 Update Manual now contains the information about the ProDOS FORMATTER routine which this Note formerly described. This routine is available from Apple Software Licensing at Apple Computer, Inc., 20525 Mariani Avenue, M/S 38-I, Cupertino, CA, 95014 or (408) 974-4667.

Note: This routine does not work properly with network volumes on either entry point. You cannot format a network volume with ProDOS 8, nor can you make low-level device calls to it (as FORMATTER does in ENTRY2 to determine the characteristics of a volume). As a general rule, it is better to use GET_FILE_INFO to determine the size of the volume since ProDOS MLI calls work with network volumes.

Further Reference
o ProDOS 8 Update Manual

END OF FILE TN.PDOS.005

###
FILE: TN.PDOS.006
###

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#6: Attaching External Commands to BASIC.SYSTEM

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald December 1985

This Technical Note formerly described how to attach an external command to BASIC.SYSTEM.

The ProDOS 8 Technical Reference Manual, Appendix A now documents the information which this Note formerly covered about installing an external command into BASIC.SYSTEM to be treated as a normal BASIC.SYSTEM command.

Further Reference
o ProDOS 8 Technical Reference Manual

END OF FILE TN.PDOS.006

```
#####################################################################
### FILE: TN.PDOS.007
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#7: Starting and Quitting Interpreter Conventions

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald December 1985

This Technical Note formerly described conventions for a ProDOS application to start and quit.

Section 5.1.5 of the ProDOS 8 Technical Reference Manual now documents the conventions a ProDOS application should follow when starting and quitting, which were formerly covered in this Note as well as ProDOS 8 Technical Note #14, Selector and Dispatcher Conventions.

Further Reference
- ProDOS 8 Technical Reference Manual

END OF FILE TN.PDOS.007

###
FILE: TN.PDOS.008
###

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#8: Dealing with /RAM

Revised by: Matt Deatherage November 1988
Written by: Kerry Laidlaw October 1984

This Technical Note formerly described conventions for dealing with the built-in ProDOS 8 RAM disk, /RAM.

Section 5.2.2 of the ProDOS 8 Technical Reference Manual now documents the conventions on how to handle /RAM, including how to disconnect it, how to reconnect it, and precautions you should follow if doing either, which were covered in this Note. The manual also includes sample source code.

Executing the sample code which comes with the manual to disconnect /RAM has the undesired effect of decreasing the maximum number of volumes on-line when used with versions of ProDOS 8 prior to 1.2. This side effect is because earlier versions of ProDOS 8 do not have the capability to remove the volume control block (VCB) entry which is allocated for /RAM when it is installed.

In later versions of ProDOS 8 (1.2 and later), this problem no longer exists, and you should issue an ON_LINE call to a device after disconnecting it. This call returns error $28 (no device connected), but it also erases the VCB entry for the disconnected device.

Further Reference

- o ProDOS 8 Technical Reference Manual
- o ProDOS 8 Update Manual

END OF FILE TN.PDOS.008

###
FILE: TN.PDOS.009
###

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#9: Buffer Management Using BASIC.SYSTEM

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald October 1985

This Technical Note discusses methods for allocating buffers which will not be arbitrarily deallocated in BASIC.SYSTEM.

Section A.2.1 of the ProDOS 8 Technical Reference Manual describes in detail how an application may obtain a buffer from BASIC.SYSTEM for its own use. The buffer will be respected by BASIC.SYSTEM, so if you choose to put a program or other executable code in there, it will be safe.

However, BASIC.SYSTEM does not provide a way to selectively deallocate the buffers it has allocated. Although it is quite easy to allocate space by calling GETBUFR ($BEF5) and also quite easy to deallocate by calling FREEBUFR ($BEF8), it is not so easy to use FREEBUFR to deallocate a particular buffer.

In fact, FREEBUFR always deallocates all buffers allocated by GETBUFR. This is fine for transient applications, but a method is needed to protect a static code buffer from being deallocated by FREEBUFR for a static application.

Location RSHIMEM ($BEFB) contains the high byte of the highest available memory location for buffers, normally $96. FREEBUFR uses it to determine the beginning page of the highest (or first) buffer. By lowering the value of RSHIMEM immediately after the first call to GETBUFR, and before any call to FREEBUFR, we can fool FREEBUFR into not reclaiming all the space. So although it is not possible to selectively deallocate buffers, it is still possible to reserve space that FREEBUFR will not reclaim.

Physically, we place the code buffer between BASIC.SYSTEM and its buffers, in the space from $99FF down.

After creating the protected static code buffer, we can call GETBUFR and FREEBUFR to maintain temporary buffers as needed by our protected module. FREEBUFR will not reclaim the protected buffer until after RSHIMEM is restored to its original value.

The following is a skeleton example which allocates a two-page buffer for a static code module, protects it from FREEBUFR, then deprotects it and restores it to its original state.

```
START    LDA #$02            ;get 2 pages
         JSR GETBUFR
         LDA RSHIMEM         ;get current RSHIMEM
```

```
         SEC                   ;ready for sub
         SBC #$02              ;minus 2 pages
         STA RSHIMEM           ;save new val to fool FREEBUFR
         JSR FREEBUFR          ;CALL FREEBUFR to deallocate.
```

At this point, the value of RSHIMEM is the page number of the beginning of our protected buffer. The static code may now use GETBUFR and FREEBUFR for transient file buffers without fear of freeing its own space from RSHIMEM to $99FF.

To release the protected space, simply restore RSHIMEM to its original value and perform a JSR FREEBUFR.

```
END      LDA RSHIMEM           ;get current val
         CLC                   ;ready for ADD
         ADC #2                ;give back 2 pages
         STA RSHIMEM           ;tell FREEBUFR about it
         JSR FREEBUFR          ;DO FREEBUFR
         RTS
```

You can reserve any number of pages using this method, as long as the amount you reserve is within available memory limits.

Further Reference

o   ProDOS 8 Technical Reference Manual

END OF FILE TN.PDOS.009

```
#####################################################################
### FILE: TN.PDOS.010
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#10: Installing Clock Driver Routines

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note formerly described how to install a clock driver routine other than the default.

Section 6.1.1 of the ProDOS 8 Technical Reference Manual documents how to install a clock driver other than the default ThunderClock(TM) driver or the Apple IIGS clock driver into ProDOS 8, which this Note formerly covered.

Further Reference
- ProDOS 8 Technical Reference Manual
- ProDOS 8 Technical Note #1, The GETLN Buffer and a ProDOS Clock Card

END OF FILE TN.PDOS.010

###
FILE: TN.PDOS.011
###

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#11: The ProDOS 8 MACHID Byte

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note describes the machine ID byte (MACHID) which ProDOS maintains to help identify different machine types.

ProDOS 8 maintains a machine ID byte, MACHID, at location $BF98 in the ProDOS 8 global page. Section 5.2.4 of the ProDOS 8 Technical Reference Manual correctly documents the definition of this byte.

MACHID has become less robust through the years. Although it can tell you if you are running on an Apple][,][+, IIe, IIc, or Apple /// in emulation mode, it cannot tell you which version of an Apple IIe or IIc you are using, nor can it identify an Apple IIGS (it thinks a IIGS is an Apple IIe). However, the byte still provides a quick test for two components of the system which you might wish to identify: an 80-column card and a clock card.

Bit 1 of MACHID identifies an 80-column card. ProDOS 8 Technical Note #15, How ProDOS 8 Treats Slot 3 explains how this identification is determined. Note that on an Apple IIGS, this bit is always set, even if the user selects Your Card in the Control Panel for slot 3. The bit is set since ProDOS 8 versions 1.7 and later switch out a card in slot 3 in favor of the built-in 80-column firmware, unless the card in slot 3 is an 80-column card. ProDOS 8 behaves in the same manner on an Apple IIe as well.

Bit 0 of MACHID identifies a clock card. Note that on an Apple IIGS, this bit is always set since the IIGS clock cannot be switched out of the system. Due to these unchangeable settings, the value of MACHID on the Apple IIGS is always $B3, as it is on any Apple IIe with an 80-column card and a clock card.

Further Reference
- ProDOS 8 Technical Reference Manual
- Apple IIGS Hardware Reference Manual
- ProDOS 8 Technical Note #15, How ProDOS 8 Treats Slot 3
- Miscellaneous Technical Note #7, Apple II Family Identification

END OF FILE TN.PDOS.011

```
#####################################################################
### FILE: TN.PDOS.012
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#12: Interrupt Handling

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note clarifies some aspects of ProDOS 8 interrupt handlers.

Although the ProDOS 8 Technical Reference Manual (section 6.2) documents interrupt handlers and includes a code example, there still remain a few unclear areas on this subject matter; this Note clarifies these areas.

All interrupt routines must begin with a CLD instruction. Although not checked in initial releases of ProDOS 8, this first byte will be checked in future revisions to verify the validity of the interrupt handler.

Although your interrupt handler does not have to disable interrupts (ProDOS 8 does that for you), it must never re-enable interrupts with a 6502 CLI instruction. Another interrupt coming through during a non-reentrant interrupt handler can bring the system down.

If your application includes an interrupt handler, you should do the following before exiting:

1. Turn off the interrupt source. Remember, 255 unclaimed interrupts will cause system death.
2. Make a DEALLOC_INTERRUPT call before exiting from your application. Do not leave a vector installed that points to a routine that is no longer there.

Within your interrupt handler routines, you must leave all memory banks in the same configuration you found them. Do not forget anything: main language card, main alternate $D000 space, main motherboard ROM, and, on an Apple IIe, IIc, or IIGS, auxiliary language card, auxiliary alternate $D000 space, alternate zero page and stack, etc. This is very important since the ProDOS interrupt receiver assumes that the environment is absolutely unaltered when your handler relinquishes control. In addition, be sure to leave the language card write-enabled.

If your handler recognizes an interrupt and services it, you should clear the carry flag (CLC) immediately before returning (RTS). If it was not your interrupt, you set set the carry (SEC) immediately before returning (RTS). Do not use a return from interrupt (RTI) to exit; the ProDOS interrupt receiver still has some housekeeping to perform before it issues the RTI instruction.

Further Reference
o   ProDOS 8 Technical Reference Manual

END OF FILE TN.PDOS.012

##
FILE: TN.PDOS.013
##

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#13: Double High-Resolution Graphics Files

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note formerly described a proposed file format for Apple II double high-resolution graphics images.

The information formerly in this Note, the proposed file format for Apple II double high-resolution graphics images, is now covered in the Apple II File Type Notes, File Type $08.

Further Reference

- o Apple II File Type Notes, File Type $08

END OF FILE TN.PDOS.013

##
FILE: TN.PDOS.014
##

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#14: Selector and Dispatcher Conventions

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald December 1985

This Technical Note formerly described conventions for a ProDOS application to start and quit.

Section 5.1.5 of the ProDOS 8 Technical Reference Manual now documents the conventions a ProDOS application should follow when starting and quitting, which were formerly covered in this Note as well as ProDOS 8 Technical Note #7, Starting and Quitting Interpreter Conventions.

Further Reference
- ProDOS 8 Technical Reference Manual

END OF FILE TN.PDOS.014

###
FILE: TN.PDOS.015
###

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#15: How ProDOS 8 Treats Slot 3

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note describes how ProDOS 8 reacts to non-Apple 80-column cards in slot 3 and how it identifies them.

The ProDOS 8 Update Manual now documents much of the information which was originally covered in this Note about how ProDOS 8 reacts to non-Apple 80-column cards in slot 3. However, since there is still some confusion on the issue, we summarize it again in this Note.

On an Apple][+, ProDOS 8 considers the following four Pascal 1.1 protocol ID bytes sufficient to identify a card in slot 3 as an 80-column card and mark the corresponding bit in the MACHID byte: $C305 = $38, $C307 = $18, $C30B = $01, $C30C = $8x, where x represents the card's own ID value and is not checked. On any other Apple II, the following fifth ID byte must also match: $C3FA = $2C. This fifth ID byte assures ProDOS 8 that the card supports interrupts on an Apple IIe. Unless ProDOS 8 finds all five ID bytes in an Apple IIe or later, it will not identify the card as an 80-column card and will enable the built-in 80-column firmware instead. In an Apple IIc or IIGS, the internal firmware always matches these five bytes (see below).

If you are designing an 80-column card and wish to meet these requirements, you must follow certain other considerations as well as matching the five identification bytes; the ProDOS 8 Update Manual enumerates these other considerations.

The ProDOS 8 Update Manual notes that an Apple IIGS does not switch in the 80-column firmware if it is not selected in the Control Panel. However, due to a bug in ProDOS 8 versions 1.6 and earlier, it switches in the 80-column firmware unconditionally. ProDOS 8 cannot respect the Control Panel setting for 80-column firmware in certain situations; it cannot operate in a 128K machine in a 128K configuration (including /RAM) without the presence of the 80-column firmware, since it must utilize the extra 64K. This is just one of the reasons ProDOS 8 does not recognize a card in slot 3 if it is not an 80-column card, as outlined above.

With ProDOS 8 version 1.7 and later, an Apple IIGS behaves exactly like an Apple IIe with respect to slot 3. If a card is slot 3 is selected in the Control Panel, ProDOS 8 ignores it in favor of the built-in 80-column firmware--unless the card matches the five identification bytes listed above. This works the same on a Apple IIe.

Further Reference
- o ProDOS 8 Technical Reference Manual
- o ProDOS 8 Update Manual
- o ProDOS 8 Technical Note #11, The ProDOS 8 MACHID Byte

END OF FILE TN.PDOS.015

###
FILE: TN.PDOS.016
###

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#16: How to Format a ProDOS Disk Device

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note supplements the ProDOS 8 Technical Reference Manual in its description of the low-level driver call that formats the media in a ProDOS device.

The ProDOS 8 Technical Reference Manual describes the low-level driver call that formats the media in a ProDOS device, but it neglects to mention the following:

1. It does not work for Disk II drives or /RAM, both of which ProDOS treats specially with built-in driver code.
2. ProDOS has no easy way to tell you whether a device is a Disk II drive or /RAM.

Once ProDOS finishes building its device table, which it does when it boots, it no longer cares about what kind of devices exist, so it does not keep any information about the different types of devices available. ProDOS identifies Disk II devices and installs a built-in driver for them. When it has installed all devices which are physically present, ProDOS then installs /RAM, in a manner similar to Disk II drives, by pointing to the driver code which is within ProDOS itself. This method presents a problem for the developer who wishes to format ProDOS disks since the Disk II driver and the /RAM driver respond to the FORMAT request in non-standard ways, yet there is no identification in the global page that tells you which devices are Disk II drives or /RAM.

The Disk II driver does not support the FORMAT request, and the /RAM driver responds by "formatting" the RAM disk and also writing to it a virgin directory and bitmap; neither of these two cases is documented in the ProDOS 8 Technical Reference Manual. To write special-case code for these two device types, you must be able to identify them, and the method for identification is available in ProDOS 8 Technical Note #21: Identifying ProDOS Devices.

You should note, however, that AppleTalk network volumes cannot be formatted; they return a DEVICE NOT CONNECTED error for the FORMAT and any low-level device call. You may access AppleTalk network volumes through ProDOS MLI calls only.

Also note that Apple licences a ProDOS 8 Formatter routine, which correctly identifies and handles Disk II drives and /RAM. You should contact Apple Software Licensing at Apple Computer, Inc., 20525 Mariani Avenue, M/S 38-I, Cupertino, CA, 95014 or (408) 974-4667 if you wish to license this routine.

Further Reference

- ProDOS 8 Technical Reference Manual
- ProDOS 8 Update Manual
- ProDOS 8 Technical Note #21, Identifying ProDOS Devices

END OF FILE TN.PDOS.016

```
#######################################################################
### FILE: TN.PDOS.017
#######################################################################
```

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#17: Recursive ProDOS Catalog Routine

Revised by: Keith Rollin & Matt Deatherage — November 1988
Written by: Greg Seitz — December 1983

This Technical Note presents an assembly language example of a recursive directory reading routine which is AppleShare compatible.

This Note presents a routine in assembly language for recursively cataloging a ProDOS disk, and if you apply this technique to the volume directory of a disk, it will display the name of every file stored on the disk. The routine displays the contents of a given directory, the volume directory in this case, then displays the contents of every directory contained within the initial directory.

Previous versions of the Note recommended reading the directory with READ_BLOCK ProDOS calls. This method will still work in most cases, but Apple Computer no longer endorses it since block-level access is not allowed to AppleShare file servers under ProDOS. (A file server cannot handle one machine connected to it changing files and directories on a block level while it is trying to arbitrate usage between other machines at the file level. A block-level change behind the server's back could easily mess things up catastrophically.)

If you are willing to use a different approach, however, you can read the directory just as easily using READ calls. Instead of using directory pointers to decide which block to read next, we simply read the directory and display filenames as we go, until we reach a subdirectory file. When we reach a subdirectory, the routine saves the necessary variables, plunges down one level of the tree structure, and catalogs the subdirectory. You repeat the process if you find a subdirectory at the current level. When you reach the EOF of any directory, the routine closes the current directory and pops back up one level, and when it reaches the EOF of the initial directory, the routine is finished.

The code example on the following pages includes a simple test of the ReadDir routine, which is the actual recursive catalog routine. Note that the simple test relies upon the GETBUFR routine in BASIC.SYSTEM to allocate a buffer; therefore, as presented, the routine requires the presence of BASIC.SYSTEM. The actual ReadDir routine requires nothing outside of the ProDOS 8 MLI.

```
01 RECURSIVE.S     ProDOS Catalog Routine            27-AUG-88  16:20 PAGE 2
```

```
----- NEXT OBJECT FILE NAME IS CATALOG.OBJ
0800:        0800                org   $800
0800:                  *******************************************************
0800:                  *
0800:                  * Recursive ProDOS Catalog Routine
0800:                  *
0800:                  * by: Greg Seitz 12/83
0800:                  *     Pete McDonald 1/86
0800:                  *     Keith Rollin 7/88
0800:                  *
0800:                  * This program shows the latest "Apple Approved"
0800:                  * method for reading a ProDOS directory. With
0800:                  * the advent of AppleTalk for the Apple II, using
0800:                  * _READBLOCK to read the directory will no longer
0800:                  * work. This routine has been re-written to read
0800:                  * directories by opening them as files, and
0800:                  * performing simple _READ commands.
0800:                  *
0800:                  *******************************************************
0800:                  *
0800:                  * Equates
0800:                  *
0800:                  * Zero page locations
0800:                  *
0800:        0080      dirName   equ   $80           ; pointer to directory name
0800:        0082      entPtr    equ   $82           ; ptr to current entry
0800:                  *
0800:                  *
0800:                  * ProDOS command numbers
0800:                  *
0800:        BF00      MLI       equ   $BF00         ; MLI entry point
0800:        BEF5      GetBufr   equ   $BEF5         ; BASIC.SYSTEM get buffer
routine
0800:                  *
0800:        00C7      GetPCmd   equ   $C7           ; Get Prefix
0800:        00C8      OpenCmd   equ   $C8           ; Open a file command
0800:        00CA      ReadCmd   equ   $CA           ; Read a file command
0800:        00CC      CloseCmd  equ   $CC           ; Close a file command
0800:        00CE      SetMCmd   equ   $CE           ; Set File Position command
0800:                  *
0800:                  *
0800:                  * Offsets into the directory
0800:                  *
0800:        0000      oType     equ   $0            ; offset to file type byte
0800:        0023      oEntLen   equ   $23           ; length of each dir. entry
0800:        0024      oEntBlk   equ   $24           ; entries in each block
0800:        0025      oEntDir   equ   $25           ; entries in entire directory
0800:                  *
0800:                  *
0800:                  * Monitor routines
0800:                  *
0800:        FDED      cout      equ   $FDED         ; output a character
0800:        FD8E      crout     equ   $FD8E         ; output a RETURN
0800:                  *
0800:                  *******************************************************
0800:                  *
0800:        0800      Start     equ   *
0800:                  *
```

```
0800:                  * Simple routine to test the recursive ReadDir

01 RECURSIVE.S     ProDOS Catalog Routine            27-AUG-88  16:20 PAGE 3

0800:                  * routine. It gets an I/O buffer for ReadDir, gets
0800:                  * the current prefix, sets the depth of recursion
0800:                  * to zero, and calls ReadDir to process all of the
0800:                  * entries in the directory.
0800:                  *
0800:A9 04                       lda   #4             ; get an I/O buffer
0802:20 F5 BE                    jsr   GetBufr
0805:B0 17   081E                bcs   exit           ; didn't get it
0807:8D DD 09                    sta   ioBuf+1
080A:                  *
080A:20 00 BF                    jsr   MLI
080D:C7                          db    GetPCmd
080E:EE 09                       dw    GetPParms
0810:B0 0C   081E                bcs   exit
0812:                  *
0812:A9 00                       lda   #0
0814:8D D2 09                    sta   Depth
0817:                  *
0817:A9 F1                       lda   #nameBuffer
0819:A2 0B                       ldx   #<nameBuffer
081B:20 1F 08                    jsr   ReadDir
081E:                  *
081E:        081E      exit      equ   *
081E:60                          rts
081F:                  *
081F:                  ******************************************************
081F:                  *
081F:        081F      ReadDir   equ   *
081F:                  *
081F:                  *  This is the actual recursive routine. It takes as
081F:                  *  input a pointer to the directory name to read in
081F:                  *  A,X (lo,hi), opens it, and starts to read the
081F:                  *  entries. When it encounters a filename, it calls
081F:                  *  the routine "VisitFile". When it encounters a
081F:                  *  directory name, it calls "VisitDir".
081F:                  *
081F:                  ******************************************************
081F:                  *
081F:85 80                       sta   dirName        ; save a pointer to name
0821:86 81                       stx   dirName+1
0823:                  *
0823:8D DA 09                    sta   openName       ; set up OpenFile params
0826:8E DB 09                    stx   openName+1
0829:                  *
0829:        0829      ReadDir1  equ   *              ; recursive entry point
0829:20 54 08                    jsr   OpenDir        ; open the directory as a
file
082C:B0 1F   084D                bcs   done
082E:                  *
082E:4C 48 08                    jmp   nextEntry      ; jump to the end of the loop
0831:                  *
0831:        0831      loop      equ   *
0831:A0 00                       ldy   #oType         ; get type of current entry
0833:B1 82                       lda   (entPtr),y
```

```
0835:C9 00          112            cmp   #0             ; inactive entry?
0837:F0 0F   0848  113            beq   nextEntry      ; yes - bump to next one
0839:29 F0          114            and   #$F0           ; look at 4 high bits
083B:C9 D0          115            cmp   #$D0           ; is it a directory?
083D:F0 06   0845  116            beq   ItsADir        ; yes, so call VisitDir

01 RECURSIVE.S     ProDOS Catalog Routine              27-AUG-88  16:20 PAGE 4

083F:20 9A 08       117            jsr   VisitFile      ; no, it's a file
0842:4C 48 08       118            jmp   nextEntry
0845:        0845  119 ItsADir   equ   *
0845:20 A1 08       120            jsr   VisitDir
0848:        0848  121 nextEntry equ   *
0848:20 68 09       122            jsr   GetNext        ; get pointer to next entry
084B:90 E4   0831  123            bcc   loop           ; Carry set means we're done
084D:        084D  124 done      equ   *
084D:               125 *
084D:20 00 BF       126            jsr   MLI            ; close the directory
0850:CC             127            db    CloseCmd
0851:E7 09          128            dw    CloseParms
0853:               129 *
0853:60             130            rts
0854:               131 *
0854:               132 ******************************************************
0854:               133 *
0854:        0854  134 OpenDir   equ   *
0854:               135 *
0854:               136 *  Opens the directory pointed to by OpenParms
0854:               137 *  parameter block. This pointer should be init-
0854:               138 *  ialized BEFORE this routine is called. If the
0854:               139 *  file is successfully opened, the following
0854:               140 *  variables are set:
0854:               141 *
0854:               142 *     xRefNum     ; all the refnums
0854:               143 *     entryLen    ; size of directory entries
0854:               144 *     entPtr      ; pointer to current entry
0854:               145 *     ThisBEntry  ; entry number within this block
0854:               146 *     ThisEntry   ; entry number within this dir.
0854:               147 *     ThisBlock   ; offset (in blocks) into dir.
0854:               148 *
0854:20 00 BF       149            jsr   MLI            ; open dir as a file
0857:C8             150            db    OpenCmd
0858:D9 09          151            dw    OpenParms
085A:B0 3D   0899  152            bcs   OpenDone
085C:               153 *
085C:AD DE 09       154            lda   oRefNum        ; copy the refnum return-
085F:8D E0 09       155            sta   rRefNum        ; ed by Open into the
0862:8D E8 09       156            sta   cRefNum        ; other param blocks.
0865:8D EA 09       157            sta   sRefNum
0868:               158 *
0868:20 00 BF       159            jsr   MLI            ; read the first block
086B:CA             160            db    ReadCmd
086C:DF 09          161            dw    ReadParms
086E:B0 29   0899  162            bcs   OpenDone
0870:               163 *
0870:AD 14 0A       164            lda   buffer+oEntLen ; init 'entryLen'
0873:8D D7 09       165            sta   entryLen
0876:               166 *
```

```
0876:A9 F5          167            lda   #buffer+4      ; init ptr to first entry
0878:85 82          168            sta   entPtr
087A:A9 09          169            lda   #<buffer+4
087C:85 83          170            sta   entPtr+1
087E:               171 *
087E:AD 15 0A       172            lda   buffer+oEntblk ; init these values based on
0881:8D D5 09       173            sta   ThisBEntry     ; values in the dir header
0884:8D D8 09       174            sta   entPerBlk

01 RECURSIVE.S      ProDOS Catalog Routine              27-AUG-88  16:20 PAGE 5

0887:AD 16 0A       175            lda   buffer+oEntDir
088A:8D D3 09       176            sta   ThisEntry
088D:AD 17 0A       177            lda   buffer+oEntDir+1
0890:8D D4 09       178            sta   ThisEntry+1
0893:               179 *
0893:A9 00          180            lda   #0             ; init block offset into dir.
0895:8D D6 09       181            sta   ThisBlock
0898:               182 *
0898:18             183            clc                  ; say that open was OK
0899:               184 *
0899:         0899  185 OpenDone  equ   *
0899:60             186            rts
089A:               187 *
089A:               188 *******************************************************
089A:               189 *
089A:         089A  190 VisitFile equ   *
089A:               191 *
089A:               192 * Do whatever is necessary when we encounter a
089A:               193 * file entry in the directory. In this case, we
089A:               194 * print the name of the file.
089A:               195 *
089A:20 A7 09       196            jsr   PrintEntry
089D:20 8E FD       197            jsr   crout
08A0:60             198            rts
08A1:               199 *
08A1:               200 *******************************************************
08A1:               201 *
08A1:         08A1  202 VisitDir  equ   *
08A1:               203 *
08A1:               204 * Print the name of the subdirectory we are looking
08A1:               205 * at, appending a "/" to it (to indicate that it's
08A1:               206 * a directory), and then calling RecursDir to list
08A1:               207 * everything in that directory.
08A1:               208 *
08A1:20 A7 09       209            jsr   PrintEntry     ; print dir's name
08A4:A9 AF          210            lda   #'/'|$80       ; tack on / at end
08A6:20 ED FD       211            jsr   cout
08A9:20 8E FD       212            jsr   crout
08AC:               213 *
08AC:20 B0 08       214            jsr   RecursDir      ; enumerate all entries in
sub-dir.
08AF:               215 *
08AF:60             216            rts
08B0:               217 *
08B0:               218 *******************************************************
08B0:               219 *
08B0:         08B0  220 RecursDir equ   *
```

```
08B0:              221 *
08B0:              222 * This routine calls ReadDir recursively. It
08B0:              223 *
08B0:              224 * - increments the recursion depth counter,
08B0:              225 * - saves certain variables onto the stack
08B0:              226 * - closes the current directory
08B0:              227 * - creates the name of the new directory
08B0:              228 * - calls ReadDir (recursively)
08B0:              229 * - restores the variables from the stack
08B0:              230 * - restores directory name to original value
08B0:              231 * - re-opens the old directory
08B0:              232 * - moves to our last position within it

01 RECURSIVE.S     ProDOS Catalog Routine             27-AUG-88  16:20 PAGE 6

08B0:              233 * - decrements the recursion depth counter
08B0:              234 *
08B0:EE D2 09      235          inc   Depth         ; bump this for recursive
call
08B3:              236 *
08B3:              237 * Save everything we can think of (the women,
08B3:              238 * the children, the beer, etc.).
08B3:              239 *
08B3:A5 82         240          lda   entPtr
08B5:48            241          pha
08B6:A5 83         242          lda   entPtr+1
08B8:48            243          pha
08B9:AD D3 09      244          lda   ThisEntry
08BC:48            245          pha
08BD:AD D4 09      246          lda   ThisEntry+1
08C0:48            247          pha
08C1:AD D5 09      248          lda   ThisBEntry
08C4:48            249          pha
08C5:AD D6 09      250          lda   ThisBlock
08C8:48            251          pha
08C9:AD D7 09      252          lda   entryLen
08CC:48            253          pha
08CD:AD D8 09      254          lda   entPerblk
08D0:48            255          pha
08D1:              256 *
08D1:              257 * Close the current directory, as ReadDir will
08D1:              258 * open files of its own, and we don't want to
08D1:              259 * have a bunch of open files lying around.
08D1:              260 *
08D1:20 00 BF      261          jsr   MLI
08D4:CC            262          db    CloseCmd
08D5:E7 09         263          dw    CloseParms
08D7:              264 *
08D7:20 20 09      265          jsr   ExtendName    ; make new dir name
08DA:              266 *
08DA:20 29 08      267          jsr   ReadDir1      ; enumerate the subdirectory
08DD:              268 *
08DD:20 56 09      269          jsr   ChopName      ; restore old directory name
08E0:              270 *
08E0:20 54 08      271          jsr   OpenDir       ; re-open it back up
08E3:              272 *
08E3:              273 * Restore everything that we saved before
08E3:              274 *
```

```
08E3:68                      pla
08E4:8D D8 09                sta   entPerBlk
08E7:68                      pla
08E8:8D D7 09                sta   entryLen
08EB:68                      pla
08EC:8D D6 09                sta   Thisblock
08EF:68                      pla
08F0:8D D5 09                sta   ThisBEntry
08F3:68                      pla
08F4:8D D4 09                sta   ThisEntry+1
08F7:68                      pla
08F8:8D D3 09                sta   ThisEntry
08FB:68                      pla
08FC:85 83                   sta   entPtr+1
08FE:68                      pla
08FF:85 82                   sta   entPtr

01 RECURSIVE.S     ProDOS Catalog Routine            27-AUG-88  16:20 PAGE 7

0901:                  *
0901:AD D6 09                lda   ThisBlock      ; reset last position in dir
0904:0A                      asl   a              ; = to block # times 512
0905:8D EC 09                sta   Mark+1
0908:A9 00                   lda   #0
090A:8D EB 09                sta   Mark
090D:8D ED 09                sta   Mark+2
0910:                  *
0910:20 00 BF                jsr   MLI            ; reset the file marker
0913:CE                      db    SetMCmd
0914:E9 09                   dw    SetMParms
0916:                  *
0916:20 00 BF                jsr   MLI            ; now read in the block we
0919:CA                      db    ReadCmd        ; were on last.
091A:DF 09                   dw    ReadParms
091C:                  *
091C:CE D2 09                dec   Depth
091F:60                      rts
0920:                  *
0920:                  ******************************************************
0920:                  *
0920:          0920    ExtendName equ  *
0920:                  *
0920:                  * Append the name in the current directory entry
0920:                  * to the name in the directory name buffer. This
0920:                  * will allow us to descend another level into the
0920:                  * disk hierarchy when we call ReadDir.
0920:                  *
0920:A0 00                   ldy   #0             ; get length of string to
copy
0922:B1 82                   lda   (entPtr),y
0924:29 0F                   and   #$0F
0926:8D 53 09                sta   extCnt         ; save the length here
0929:8C 54 09                sty   srcPtr         ; init src ptr to zero
092C:                  *
092C:A0 00                   ldy   #0             ; init dest ptr to end of
092E:B1 80                   lda   (dirName),y    ; the current directory name
0930:8D 55 09                sta   destPtr
0933:                  *
```

```
0933:         0933  329 extloop   equ   *
0933:EE 54 09       330           inc   srcPtr        ; bump to next char to read
0936:EE 55 09       331           inc   destPtr       ; bump to next empty location
0939:AC 54 09       332           ldy   srcPtr        ; get char of sub-dir name
093C:B1 82          333           lda   (entPtr),y
093E:AC 55 09       334           ldy   destPtr       ; tack on to end of cur. dir.
0941:91 80          335           sta   (dirName),y
0943:CE 53 09       336           dec   extCnt        ; done all chars?
0946:D0 EB    0933  337           bne   extloop       ; no - so do more
0948:               338 *
0948:C8             339           iny
0949:A9 2F          340           lda   #'/'          ; tack "/" on to the end
094B:91 80          341           sta   (dirName),y
094D:               342 *
094D:98             343           tya                 ; fix length of filename to
open
094E:A0 00          344           ldy   #0
0950:91 80          345           sta   (dirName),y
0952:               346 *
0952:60             347           rts
0953:               348 *

01 RECURSIVE.S      ProDOS Catalog Routine             27-AUG-88  16:20 PAGE 8

0953:         0001  349 extCnt    ds    1
0954:         0001  350 srcPtr    ds    1
0955:         0001  351 destPtr   ds    1
0956:               352 *
0956:               353 *
0956:               354 *******************************************************
0956:               355 *
0956:         0956  356 ChopName  equ   *
0956:               357 *
0956:               358 * Scans the current directory name, and chops
0956:               359 * off characters until it gets to a /.
0956:               360 *
0956:A0 00          361           ldy   #0            ; get len of current dir.
0958:B1 80          362           lda   (dirName),y
095A:A8             363           tay
095B:         095B  364 ChopLoop  equ   *
095B:88             365           dey                 ; bump to previous char
095C:B1 80          366           lda   (dirName),y
095E:C9 2F          367           cmp   #'/'
0960:D0 F9    095B  368           bne   ChopLoop
0962:98             369           tya
0963:A0 00          370           ldy   #0
0965:91 80          371           sta   (dirName),y
0967:60             372           rts
0968:               373 *
0968:               374 *******************************************************
0968:               375 *
0968:         0968  376 GetNext   equ   *
0968:               377 *
0968:               378 * This routine is responsible for making a pointer
0968:               379 * to the next entry in the directory. If there are
0968:               380 * still entries to be processed in this block, then
0968:               381 * we simply bump the pointer by the size of the
0968:               382 * directory entry. If we have finished with this
```

```
0968:                383 * block, then we read in the next block, point to
0968:                384 * the first entry, and increment our block counter.
0968:                385 *
0968:AD D3 09        386            lda   ThisEntry     ; dec total entries
096B:D0 05   0972    387            bne   skip1
096D:CE D4 09        388            dec   ThisEntry+1
0970:30 33   09A5    389            bmi   DirDone       ; done with this directory
0972:CE D3 09        390 skip1      dec   ThisEntry
0975:                391 *
0975:CE D5 09        392            dec   ThisBEntry    ; dec count for this block
0978:F0 10   098A    393            beq   ReadNext      ; done w/this block, get next
one
097A:                394 *
097A:18              395            clc                 ; else bump up index
097B:A5 82           396            lda   entPtr
097D:6D D7 09        397            adc   entryLen
0980:85 82           398            sta   entPtr
0982:A5 83           399            lda   entPtr+1
0984:69 00           400            adc   #0
0986:85 83           401            sta   entPtr+1
0988:18              402            clc                 ; say that the buffer's good
0989:60              403            rts
098A:                404 *
098A:        098A    405 ReadNext   equ   *
098A:20 00 BF        406            jsr   MLI           ; get the next block

01 RECURSIVE.S       ProDOS Catalog Routine              27-AUG-88  16:20 PAGE 9

098D:CA              407            db    ReadCmd
098E:DF 09           408            dw    ReadParms
0990:B0 13   09A5    409            bcs   DirDone
0992:                410 *
0992:A9 F5           411            lda   #buffer+4     ; set entry pointer to
beginning
0994:85 82           412            sta   entPtr
0996:A9 09           413            lda   #<buffer+4
0998:85 83           414            sta   entPtr+1
099A:                415 *
099A:AD D8 09        416            lda   entPerBlk     ; re-init 'entries in this
block'
099D:8D D5 09        417            sta   ThisBEntry
09A0:CE D5 09        418            dec   ThisBEntry
09A3:18              419            clc                 ; return 'No error'
09A4:60              420            rts
09A5:                421 *
09A5:        09A5    422 DirDone    equ   *             ; return 'All Done!'
09A5:38              423            sec                 ; return 'an error occurred'
09A6:60              424            rts
09A7:                425 *
09A7:                426 *******************************************************
09A7:                427 *
09A7:        09A7    428 PrintEntry equ   *
09A7:                429 *
09A7:                430 * Using the pointer to the current entry, this
09A7:                431 * routine prints the entry name. It also pays
09A7:                432 * attention to the recursion depth, and indents
09A7:                433 * by 1 space for every level.
09A7:                434 *
```

```
09A7:AD D2 09      435           lda   Depth          ; init counter for indenting
09AA:8D D1 09      436           sta   PrntCnt
09AD:4C B5 09      437           jmp   spcDec
09B0:A9 A0         438 spcloop   lda   #$A0           ; print a space for indenting
09B2:20 ED FD      439           jsr   cout
09B5:         09B5 440 spcDec    equ   *
09B5:CE D1 09      441           dec   PrntCnt        ; any more indenting?
09B8:10 F6    09B0 442           bpl   spcloop        ; yes - keep going
09BA:              443 *
09BA:A0 00         444           ldy   #0             ; get byte that has the
length byte
09BC:B1 82         445           lda   (entPtr),y
09BE:29 0F         446           and   #$0F           ; get just the length
09C0:8D D1 09      447           sta   PrntCnt        ; put it into our counter
09C3:         09C3 448 PrntLoop  equ   *
09C3:C8            449           iny                  ; bump to the next char.
09C4:B1 82         450           lda   (entPtr),y     ; get next char
09C6:09 80         451           ora   #$80           ; COUT likes high bit set
09C8:20 ED FD      452           jsr   cout           ; print it
09CB:CE D1 09      453           dec   PrntCnt        ; printed all chars?
09CE:D0 F3    09C3 454           bne   PrntLoop       ; no - keep going
09D0:60            455           rts
09D1:              456 *
09D1:         0001 457 PrntCnt   ds    1              ; counter for printing
09D2:              458 *
09D2:              459 *****************************************************
09D2:              460 *
09D2:              461 * Some global variables
09D2:              462 *
09D2:         0001 463 Depth     ds    1              ; amount of recursion
09D3:         0002 464 ThisEntry ds    2              ; abs entry number

01 RECURSIVE.S     ProDOS Catalog Routine             27-AUG-88  16:20 PAGE 10

09D5:         0001 465 ThisBEntry ds   1              ; entry in this block
09D6:         0001 466 ThisBlock ds    1              ; block with dir
09D7:         0001 467 entryLen  ds    1              ; length of each directory
entry
09D8:         0001 468 entPerBlk ds    1              ; entries per block
09D9:              469 *
09D9:              470 *****************************************************
09D9:              471 *
09D9:              472 * ProDOS command parameter blocks
09D9:              473 *
09D9:         09D9 474 OpenParms equ   *
09D9:03            475           db    3              ; number of parms
09DA:         0002 476 OpenName  ds    2              ; pointer to filename
09DC:00 00         477 ioBuf     dw    $0000          ; I/O buffer
09DE:         0001 478 oRefNum   ds    1              ; returned refnum
09DF:              479 *
09DF:         09DF 480 ReadParms equ   *
09DF:04            481           db    4              ; number of parms
09E0:         0001 482 rRefNum   ds    1              ; refnum from Open
09E1:F1 09         483           dw    buffer         ; pointer to buffer
09E3:00 02         484 reqAmt    dw    512            ; amount to read
09E5:         0002 485 retAmt    ds    2              ; amount actually read
09E7:              486 *
09E7:         09E7 487 CloseParms equ  *
```

```
09E7:01             488            db    1              ; number of parms
09E8:         0001  489 cRefNum    ds    1              ; refnum from Open
09E9:               490 *
09E9:         09E9  491 SetMParms  equ   *
09E9:02             492            db    2              ; number of parms
09EA:         0001  493 sRefNum    ds    1              ; refnum from Open
09EB:         0003  494 Mark       ds    3              ; file position
09EE:               495 *
09EE:         09EE  496 GetPParms  equ   *
09EE:01             497            db    1              ; number of parms
09EF:F1 0B          498            dw    nameBuffer     ; pointer to buffer
09F1:               499 *
09F1:         0200  500 buffer     ds    512            ; enough for whole block
0BF1:               501 *
0BF1:         0040  502 nameBuffer ds    64             ; space for directory name

01 SYMBOL TABLE    SORTED BY SYMBOL                       27-AUG-88  16:20 PAGE 11

 09F1 BUFFER         095B CHOPLOOP       0956 CHOPNAME         CC CLOSECMD
 09E7 CLOSEPARMS     FDED COUT           09E8 CREFNUM        FD8E CROUT
 09D2 DEPTH          0955 DESTPTR        09A5 DIRDONE          80 DIRNAME
 084D DONE           09D8 ENTPERBLK        82 ENTPTR         09D7 ENTRYLEN
 081E EXIT           0953 EXTCNT         0920 EXTENDNAME     0933 EXTLOOP
 BEF5 GETBUFR        0968 GETNEXT          C7 GETPCMD        09EE GETPPARMS
 09DC IOBUF          0845 ITSADIR        0831 LOOP           09EB MARK
 BF00 MLI            0BF1 NAMEBUFFER     0848 NEXTENTRY        24 OENTBLK
   25 OENTDIR          23 OENTLEN          C8 OPENCMD        0854 OPENDIR
 0899 OPENDONE       09DA OPENNAME       09D9 OPENPARMS      09DE OREFNUM
   00 OTYPE          09A7 PRINTENTRY     09D1 PRNTCNT        09C3 PRNTLOOP
   CA READCMD        0829 READDIR1       081F READDIR        098A READNEXT
 09DF READPARMS      08B0 RECURSDIR     ?09E3 REQAMT        ?09E5 RETAMT
 09E0 RREFNUM          CE SETMCMD        09E9 SETMPARMS      0972 SKIP1
 09B5 SPCDEC         09B0 SPCLOOP        0954 SRCPTR         09EA SREFNUM
?0800 START          09D5 THISBENTRY     09D6 THISBLOCK      09D3 THISENTRY
 08A1 VISITDIR       089A VISITFILE
** SUCCESSFUL ASSEMBLY := NO ERRORS
** ASSEMBLER CREATED ON 15-JAN-84 21:28
** TOTAL LINES ASSEMBLED   502
** FREE SPACE PAGE COUNT   81
```

Further Reference

o   ProDOS 8 Technical Reference Manual

END OF FILE TN.PDOS.017

##
FILE: TN.PDOS.018
##

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#18: /RAM Memory Map

Revised by: Matt Deatherage November 1988
Written by: Pete McDonald December 1986

This Technical Note describes the block to actual memory location mapping of /RAM.

Blocks	Address Range	
$70-$7F	$E000-$EFFF	
$68-$6F	$D000-$DFFF	(Bank 2)
$60-$67	$D000-$DFFF	(Bank 1)
$4E-$5C	$A200-$BFFF	
$3D-$4C	$8200-$A1FF	
$2C-$3B	$6200-$81FF	
$1B-$2A	$4200-$61FF	
$0A-$19	$2200-$41FF	

$5D-$5F	$1A00-$1FFF
$4D	$1800-$19FF
$3C	$1600-$17FF
$2B	$1400-$15FF
$1A	$1200-$13FF
$09	$1000-$11FF
$08	$2000-$21FF
$02	$0E00-$0FFF

$03	Bitmap*

Notes:
*   Synthesized.
1.   Blocks 0, 1, 4, 5, 6, and 7 do not exist.
2.   Block $7F contains the Reset, IRQ, and NMI vectors and is normally marked as used.
3.   The memory from $0C00 - $0DFF is a general purpose buffer used by the /RAM driver.

END OF FILE TN.PDOS.018

```
######################################################################
### FILE: TN.PDOS.019
######################################################################
```

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#19: File Auxiliary Type Assignment

Revised by: Matt Deatherage November 1988
Written by: Matt Deatherage May 1988

This Technical Note describes file auxiliary type assignments.

The information in a ProDOS file auxiliary type field depends upon its primary file type. For example, the auxiliary type field for a text file (TXT, $04) is defined as the record length of the file if it is a random-access file, or zero if it is a sequential file. The auxiliary type field for an AppleWorks(TM) file contains information about the case of letters in the filename (see Apple II File Type Notes, File Types $19, $1A, and $1B). The auxiliary type field for a binary file (BIN, $06) contains the loading address of the file, if one exists.

Auxiliary types are now used to extend the limit of 256 file types in ProDOS. Specific auxiliary types can be assigned to generic application file types. For example, if you need a file type for your word-processing program, Apple might assign you an auxiliary type for the generic file type of Apple II word processor file, if it is appropriate.

An application can determine if a given file belongs to it by checking the file type and the auxiliary type in the directory entry. Other programming considerations include the following:

1. If your program displays auxiliary type information, it should include all auxiliary types, not just selected ones. Try to display the auxiliary type information stored in the directory entry, just as you would display hex codes for file types for which you do not have a more descriptive message to display.
2. Programs should not store information in an undefined auxiliary type field. Storing the record length in a text file is fine, and it is even encouraged, but storing the number of words in a text file in that text file's auxiliary type field might cause problems for those programs which expect to find a record length there. Similarly, storing data in the auxiliary type field will cause problems if your data matches an auxiliary type which is assigned. To avoid these problems, only store defined items in a file's auxiliary type field. If you do not know of a definition for a particular file type's associated auxiliary type, do not store anything in its field.

To request a file type and auxiliary type, please send Apple II Developer Technical Support a description of your proposed file format, along with a justification for not using existing file and auxiliary types. We will publish this information publicly, unless you specifically prohibit it, since we feel doing so enables the exchange of data for those applications who choose to support other file formats.

Further Reference

- ProDOS 8 Technical Reference Manual
- ProDOS 16 Technical Reference

END OF FILE TN.PDOS.019

##
FILE: TN.PDOS.020
##

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#20: Mirrored Devices and SmartPort

Revised by: Matt Deatherage — November 1988
Written by: Matt Deatherage — May 1988

This Technical Note describes how ProDOS 8 reacts when more than two SmartPort devices are connected, how applications using direct device access should behave, and other related issues. This Note supersedes Section 6.3.1 of the ProDOS 8 Technical Reference Manual.

Although SmartPort theoretically can handle up to 127 devices connected to a single interface (in practice, electrical considerations curtail this considerably), ProDOS 8 can handle only two devices per slot. This is because ProDOS uses bit 7 of its unit_number is used to distinguish drives from each other, and a single bit cannot distinguish more than two devices.

When it boots, ProDOS checks each interface card (or firmware equivalent in the IIc or IIGS) for the ProDOS block-device signature bytes ($Cn01 = $20, $Cn03 = $00, and $Cn05 = $03), so it can install the appropriate device-driver address in the system global page. If the signature bytes match, ProDOS then checks the SmartPort signature byte ($Cn07 = $00), and if that byte matches and the interface is in slot 5 (or located at $C500 in the IIc or IIGS), ProDOS does a SmartPort STATUS call to determine how many devices are connected to the interface. If only one or two drives are connected to the interface, ProDOS installs its block-device entry point (the contents of $CnFF added to $Cn00) in the device-driver vector table, which starts at $BF10. In this particular instance, ProDOS would put the vector at $BF1A for slot 5, drive 1, and if two drives were found, at $BF2A for slot 5, drive 2 .

If the interface is in slot 5 and more than two devices are connected, ProDOS copies the same block-device entry point that it uses for slot 5, drives 1 and 2 in the device driver table entry for slot 2, drive 1, and if four drives are connected, for slot 2, drive 2. Further in the boot process, if ProDOS finds the interface of a block device in slot 2 (not possible on a IIc), it replaces the vectors copied from slot 5 with the proper device-driver vectors for slot 2; this is the reason mirroring is disabled if there is a ProDOS device in slot 2. Note that non-ProDOS devices (i.e, serial cards and ports, etc.) do not have vectors installed in the ProDOS device-driver table, so they do not interfere with mirroring.

When ProDOS makes an MLI call with the unit_number of a mirrored device, it sets up the call to the device driver then goes through the vector in the device-driver table starting at $BF00. When the block device driver (located on the interface card or in the firmware) gets this MLI call, it checks the unit number which is stored at $43 and verifies if the slot number (bits four, five, and six) is the same as that of the interface. If it is not, the ProDOS block device driver of the interface realizes it is dealing with a mirrored device, internally adds three to the slot number and two to the drive number, then processes it, returning the desired information or data to ProDOS.

If an application must make direct device-driver calls (something which is not encouraged), it should first check devlst (starting at $BF32) to verify that the unit_number is from an active device. In addition, the application should mask off or ignore the low nibble of entries in devlst and know that one less than the number of devices in the list is stored at $BF31 (devcnt). The application then should use the unit_number to get the proper device-driver vector from the ProDOS global page; the application should not construct the vector itself, because this vector would be invalid for a mirrored device.

The following code fragment correctly illustrates this technique. It is written in 6502 assembly language and assumes the unit_number is in the accumulator.

```
devcnt    equ    $BF31
devlst    equ    $BF32
devadr    equ    $BF10
devget    sta    unitno         ; store for later compare instruction
          ldx    devcnt         ; get count-1 from $BF31
devloop   lda    devlst,x       ; get entry in list
          and    #$F0           ; mask off low byte
devcomp   cmp    unitno         ; compare to the unit_number we filled in
          beq    goodnum        ;
          dex
          bpl    devloop        ; loop again if still less than $80
          bmi    badunitno      ; error: bad unit number
goodnum   lda    unitno         ; get good copy of unit_number
          lsr    a              ; divide it by 8
          lsr    a              ; (not sixteen because devadr entries are
          lsr    a              ; two bytes wide)
          tax
          lda    devadr,x       ; low byte of device driver address
          sta    addr
          lda    devadr+1,x     ; high byte of device driver address
          sta    addr+1
          rts
addr      dw     0              ; address will be filled in here by goodnum
unitno    dfb    0              ; unit number storage
```

Similarly, applications which construct firmware entry points from user input to "slot and drive" questions will not work with mirrored devices. If an application wishes to issue firmware-specific calls to a device, it should look at the high byte of the device-driver table entry for that device to obtain the proper place to check firmware ID bytes. In the sample code above, the high byte would be returned in addr+1. For devices mirrored to slot 2 from slot 5, this technique will return $C5, and ID bytes would then be checked (since they should always be checked before making device-specific calls) in the $C500 space. Applications ignoring this technique will incorrectly check the $C200 space.

Further Reference
o   ProDOS 8 Technical Reference Manual

o   ProDOS 8 Technical Note #21, Identifying ProDOS Devices

END OF FILE TN.PDOS.020

###
FILE: TN.PDOS.021
###

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#21: Identifying ProDOS Devices

Revised by: Dave Lyons & Matt Deatherage July 1989
Written by: Matt Deatherage & Dan Strnad November 1988

This Technical Note describes how to identify ProDOS devices and their characteristics given the ProDOS unit number. This scheme should only be used under ProDOS 8.
Changes since November 1988: Added a section on things to avoid.

There are various reasons why an application would want to identify ProDOS devices. Although ProDOS itself takes great pains to treat all devices equally, it has internal drivers for two types of devices: Disk II drives and the /RAM drive provided on 128K or greater machines. Because all devices really are not equal (i.e., some cannot format while others are read-only, etc.), a developer may need to know how to identify a ProDOS device.

Although the question of how much identification is subjective for each developer, ProDOS 8 offers a fair level of identification; the only devices which cannot be conclusively identified are those devices with RAM-based drivers, and they could be anything. The vast majority of ProDOS devices can be identified, however, so you could prompt the user to insert a disk in UniDisk 3.5 #2, instead of Slot 2, Drive 2, which could be confusing if the user has a IIc or IIGS.

Note that for the majority of applications, this level of identification is unnecessary. Most applications simply prompt the user to insert a disk by its name, and the user can place it in any drive which is capable of working with the media of the disk. You should avoid requiring a certain disk to be in a specific drive since doing so defeats much of the device-independence which gives ProDOS 8 its strength.

When you do need to identify a device (i.e., if you need to format media in a Disk II or /RAM device), however, the process is fairly straightforward. This process consists of a series of tests, any one of which could end with a conclusive device identification. It is not possible to look at a single ID byte to determine a particular device type. You may determine rather quickly that a device is a SmartPort device, or you may go all the way through the procedure to identify a third-party network device. For those developers who absolutely must identify devices, we present the following discussion.

Isn't There Some Kind of "ID Nibble?"

ProDOS 8 does not support an "ID nibble." Section 5.2.4 of the ProDOS 8

Technical Reference Manual states that the low nibble of each unit number in the device list "is a device identification: 0 = Disk II, 4 = Profile, $F = /RAM."

When ProDOS 8 finds a "smart" ProDOS block device while doing its search of the slots and ports, it copies the high nibble of $CnFE (where n is the slot number) into the low nibble of the unit number in the global page. The low nibble then has the following definition:

```
    Bit 3:      Medium is removable
    Bit 2:      Device is interruptible
    Bit 1-0:    Number of volumes on the device (minus one)
```

As you can see, it is quite easy for the second definition to produce one of the original values (e.g., 0, 4, or $F) in the same nibble for completely different reasons. You should ignore the low nibble in the unit number in the global page when identifying devices since the first definition is insufficient to uniquely identify devices and the second definition contains no information to specifically identify devices. Once you do identify a ProDOS block device, however, you may look at $CnFE to obtain the information in the second definition above, as well as information on reading, writing, formatting, and status availability.

When identifying ProDOS devices, we start with a list of unit numbers for all currently installed disk devices. As we progress through the identification process, we will identify some devices while we will not know about others until the end of the process.

Starting with the Unit Number

ProDOS unit numbers (unit_number) are bytes where the bits are arranged in the pattern DSSS0000, where D = 0 for drive one and D = 1 for drive two, SSS is a three-bit integer with values from one through seven indicating the device slot number (zero is not a valid slot number), and the low nibble is ignored.

To obtain a list of the unit numbers for all currently installed ProDOS disk devices, you can perform a ProDOS MLI ON_LINE call with a unit number of $00. This call returns a unit number and a volume name for every device in the device list. ProDOS stores the length of the volume name in the low nibble of the unit number which ON_LINE returns; if an error occurs, the low nibble will contain $0 and the byte immediately following the unit number will contain an error code. For more information on the ON_LINE call, see section 4.4.6 of the ProDOS 8 Technical Reference Manual. We will discuss the error codes in more detail later in this Note.

To identify the devices in the device list, we need to know in which physical slot the hardware resides, so we can look at the slot I/O ROM space and check the device's identification bytes. Note that the slot-number portion of the unit number does not always represent the physical slot of the device, rather, it sometimes represents the logical slot where you can find the address of the device's driver entry point in the ProDOS global page. For example, if a SmartPort device interface in slot 5 has more than two connected devices, the third and fourth devices will be mapped to slot 2; this mapping gives these two devices unit numbers of $20 and $A0 respectively, but the device's driver entry point will still be in the $C5xx address space.

ProDOS 8 Technical Note #20, Mirrored Devices and SmartPort, discusses this kind of mapping in detail. It also presents a code example which gives you the correct device-driver entry point (from the global page) given the unit number as input. We repeat this code example below for your benefit. It assumes the unit_number is in the accumulator.

```
devcnt    equ    $BF31
devlst    equ    $BF32
devadr    equ    $BF10
devget    sta    unitno         ; store for later compare instruction
          ldx    devcnt         ; get count-1 from $BF31
devloop   lda    devlst,x       ; get entry in list
          and    #$F0           ; mask off low nibble
devcomp   cmp    unitno         ; compare to the unit_number we filled in
          beq    goodnum        ;
          dex
          bpl    devloop        ; loop again if still less than $80
          bmi    badunitno      ; error: bad unit number
goodnum   lda    unitno         ; get good copy of unit_number
          lsr    a              ; divide it by 8
          lsr    a              ; (not sixteen because devadr entries are
          lsr    a              ; two bytes wide)
          tax
          lda    devadr,x       ; low byte of device driver address
          sta    addr
          lda    devadr+1,x     ; high byte of device driver address
          sta    addr+1
          rts
addr      dw     0              ; address will be filled in here by goodnum
unitno    dfb    0              ; unit number storage
```

Warning: Attempting to construct the device-driver entry point from the unit number is very dangerous. Always use the technique presented above.

Network Volumes

AppleTalk volumes present a special problem to some developers since they appear as "phantom devices," or devices which do not always have a device driver installed in the ProDOS global page. Fortunately, the ProDOS Filing Interface (PFI) to AppleTalk provides a way to identify network volumes through an MLI call. The ProDOS Filing Interface call FIListSessions is used to retrieve a list of the current sessions being maintained through PFI and any volumes mounted for those sessions. The following presents an example:

```
Network   JSR    $BF00          ;ProDOS MLI
          DFB    $42            ;AppleTalk command number
          DW     ParamAddr      ;Address of Parameter Table
          BCS    ERROR          ;error occurred

ParamAddr DFB    $00            ;Async Flag (0 means synchronous only)
                                ;note there is no parameter count
          DFB    $2F            ;command for FIListSessions
          DW     $0000          ;AppleTalk Result Code returned here
          DW     BufLength      ;length of the buffer supplied
          DW     BufPointer     ;low word of pointer to buffer
          DW     $0000          ;high word of pointer to buffer
```

```
                                ;(THIS WILL NOT BE ZERO IF THE BUFFER IS
                                ;NOT IN BANK ZERO!!!)
          DFB    $00            ;Number of entries returned here
```

If the FIListSessions call fails with a bad command error ($01), then AppleShare is not installed; therefore, there are no networks volumes mounted. If there is a network error, the accumulator will contain $88 (Network Error), and the result code in the parameter block will contain the specific error code. The list of current sessions is placed into the buffer (at the address BufPointer in the example above), but if the buffer is not large enough to hold the list, it will retain the maximum number of current sessions possible and return an error with a result code of $0A0B (Buffer Too Small). The buffer format is as follows:

```
SesnRef   DFB    $00            ;Sessions Reference number (result)
UnitNum   DFB    $00            ;Unit Number (result)
VolName   DS     28             ;28 byte space for Volume Name
                                ;(starts with a length byte)
VolumeID  DW     $0000          ;Volume ID (result)
```

This list is repeated for every volume mounted for each session (the number is placed into the last byte of the parameter list you passed to the ProDOS MLI). For example, if there are two volumes mounted for session one, then session one will be listed two times. The UnitNum field contains the slot and drive number in unit-number format, and note that bit zero of this byte is set if the volume is a user volume (i.e., it contains a special "users" folder). This distinction is unimportant for identifying a ProDOS device as a network pseudo-device, but it is necessary for applications which need to know the location of the user volume. Note that if you mount two servers or more with each having its own user volume, the user volume found first in the list (scanned top to bottom) returned by FIListSessions specifies the user volume that an application should use. See the AppleShare Programmer's Guide for the Apple IIGS (available from the Apple Programmer's and Developer's Association (APDA)) for more information on programming for network volumes.

If you keep a list of all unit numbers returned by the ON_LINE call and mark each one "identified" as you identify it, keep in mind that the unit numbers returned by FIListSessions and ON_LINE have different low nibbles which should be masked off before you make any comparisons.

Note: You should mark the network volumes as identified and not try to identify them further with the following methods.

What Slot is it Really In?

Once you have the address of the device driver's entry point and know that the device is not a network pseudo-device, you can determine in what physical slot the device resides. If the high byte of the device driver's entry point is of the form $Cn, then n is the physical slot number of the device. A SmartPort device mirrored to slot 2 will have a device driver address of $C5xx, giving 5 as the physical slot number.

If the high byte of the device driver entry point is not of the form $Cn, then there are three other possibilities:

- o The device is a Disk II with driver code inside ProDOS.
- o The device is either /RAM with driver code inside ProDOS or a

third-party auxiliary-slot RAM disk device with driver code installed somewhere in memory.

- o The device is not a RAM disk but has a RAM-based device driver, like a third-party network device.

Auxiliary-slot RAM disks are identified by convention. Any device in slot 3, drive 2 (unit number $B0) is assumed to be an auxiliary-slot RAM disk since ProDOS 8 will not recognize any card which is not an 80-column card in slot 3 (see ProDOS 8 Technical Note #15). There is a chance that some other kind of device could be installed with unit number $B0, but it is not likely.

To identify various kinds of auxiliary-slot RAM disks, you must obtain the unit number from the ProDOS global page. The list of unit numbers starts at $BF32 (DEVLST) and is preceded by the number of unit numbers minus one (DEVCNT, at $BF31). You should search through this list until you find a unit number in the form $Bx; if the unit number is $B3, $B7, $BB, or $BF, you can assume the device to be an auxiliary-slot RAM disk which uses the auxiliary 64K bank of memory present in a 128K Apple IIe or IIc, or a IIGS. If the unit number is one of the four listed above, you must remove this device to safely access memory in the auxiliary 64K bank, but if the unit number is not one of the four listed above, you can assume the device to be an auxiliary-slot RAM disk which does not use the normal bank of auxiliary memory. (Some third-party auxiliary-slot cards contain more than one 64K auxiliary bank; the normal use of this memory is as a RAM disk. If the RAM-based driver for this kind of card does not use the normal auxiliary 64K bank for storage, it should have a unit number other than one of the four listed above.) If the unit number is not one of the four listed above, you may safely access the auxiliary bank of memory without first removing this device.

Section 5.2.2.3 of the ProDOS 8 Technical Reference Manual contains a routine which disconnects the appropriate RAM disk devices in slot 3, drive 2, without removing those drivers which do not use that bank, to allow use of the auxiliary 64K bank.

Note: Previous information from Apple indicated that /RAM could be distinguished from third-party RAM disks by a driver address of $FF00. Although the address has not changed, some third-party drivers may have addresses of $FF00 as well, although this is not supported. /RAM always has a driver address of $FF00 and unit number $BF, although any third-party RAM disk could install itself with similar attributes.

For Disk II devices, the three-bit slot number portion of the unit_number will always be the physical slot number. Disk II devices can never be mirrored to another slot (the Disk II driver does not support it); therefore, it will be in the physical slot represented in the unit number which ProDOS assigns when it boots.

If the high byte of the device driver's entry point is not of the form $Cn, then you should assume that the slot number is the value SSS in the unit number (this is equivalent to assuming the device is a Disk II) for the next step, which is checking the I/O space for identification bytes.

What to Do With the Slot Number

Once you have the slot number, you can look at the slot I/O ROM space to determine the kind of device it is. As described in the ProDOS 8 Technical

Reference Manual, ProDOS looks for the following ID bytes in ROM to determine if a ProDOS device is in a slot:

```
$Cn01 = $20
$Cn03 = $00
$Cn05 = $03
```

If you use the slot number, n, you obtained above, and the three values listed above are not present, then the device has a RAM-based driver and cannot further be identified.

If the three values previously discussed are present, then examination of $CnFF will give more information. If $CnFF = $00, the device is a Disk II. If $CnFF is any value other than $00 or $FF ($FF signifies a 13-sector Disk II, which ProDOS does not support), the device is a ProDOS block device.

For ProDOS block devices, the byte at $CnFE contains several flags which further identify the device; these flags are discussed in section 6.3.1 of the ProDOS 8 Technical Reference Manual.

SmartPort Devices

Many of Apple's ProDOS block devices follow the SmartPort firmware interface. Through SmartPort, you can further identify devices. Existing SmartPort devices include SCSI hard disks, 3.5" disk drives and CD-ROM drives, with many more possible device types.

If $Cn07 = $00, then the device is a SmartPort device, and you can then make a SmartPort call to get more information about the device, including a device type and subtype. The SmartPort entry point is three bytes beyond the ProDOS block device entry point, which you already determined above. The method for making SmartPort calls is outlined in the Apple IIc Technical Reference Manual and the Apple IIGS Firmware Reference.

The most useful SmartPort call to make for device identification is the STATUS call with statcode = 3 for Return Device Information Block (DIB). This call returns the ASCII name of the device, a device type and subtype, as well as the size of the device. Some SmartPort device types and subtypes are listed in the referenced manuals, with a more complete list located in the Apple IIGS Firmware Reference. A list containing SmartPort device types only is provided in SmartPort Technical Note #4, SmartPort Device Types.

RAM-Based Drivers

One fork of the identification tree comes to an end at this point. If the high byte of the device driver entry point was not $Cn and the device was not /RAM, we assumed it was a Disk II and used the slot number portion of the unit number to examine the slot ROM space. If the ROM space for that slot number does not match the three ProDOS block device ID bytes, it cannot be a Disk II. Having ruled out other possibilities, it must be a device installed after ProDOS finished building its device table. Perhaps it is a third-party RAM disk driver or maybe a driver for an older card which does not match the ProDOS block device ID bytes.

Whatever the function of the driver, you can identify it no further. It quite literally could be any kind of device at all, and with neither slot ROM space to identify nor a standard location to compare the device driver entry point against, the best you can do is consider it a "generic device" and go on.

But Is It Connected, and Can I Read From It?

Just because a ProDOS device is in the table does not mean it is ready to be used. There is always the possibility that the drive has no media in it. Back in the beginning, we made an ON_LINE call with a unit number of $00. If the volume name of a disk in that device could not be read, or another error occurred, ProDOS 8 would return the error code to us in the ON_LINE buffer immediately following the unit number. Those errors possible include:

$27	I/O error
$28	No Device Connected
$2B	Write Protected
$2F	Device off-line
$45	Volume directory not found
$52	Not a ProDOS disk
$55	Volume Control Block full
$56	Bad buffer address
$57	Duplicate volume on-line

Note that error $2F is not listed in the ProDOS 8 Technical Reference Manual.

By convention, we interpret I/O error to mean the disk in the drive is either damaged or blank (not formatted). We interpret Device off-line to mean that there is no disk in the drive. We interpret No Device Connected to mean the drive really does not exist (for example, asking for status on a second Disk II when only one is connected).

If no error occurred for a unit number in the ON_LINE call (the low nibble of the unit number is not zero), the volume name of the disk in the drive follows the unit number.

Things To Avoid

The ProDOS device-level STATUS call generally returns the number of blocks on a device. Applications should not try to identify 3.5" drives by doing a ProDOS or SmartPort STATUS call and comparing the number of blocks to 800 or 1,600. The correct way to identify a 3.5" drive is by the Type field in a SmartPort STATUS call.

Don't assume the characteristics of a device just because it is in a certain slot. For example, be prepared to deal with 5.25" disk drives in slots other than 6. Don't assume that slot 6 is associated with block devices at all--there could be a printer card installed.

Avoid reinstalling /RAM when your application finds it removed. If you remove /RAM, you should reinstall it when you're done with the extra memory; however, if your application finds /RAM already gone, you do not have the right to just reinstall it. A driver of some kind may be installed in auxiliary memory, and arbitrary reinstallation of /RAM could bring the system down.

Further Reference

- ProDOS 8 Technical Reference Manual
- AppleShare Programmer's Guide for the Apple IIGS
- ProDOS 8 Technical Note #15, How ProDOS 8 Treats Slot 3
- ProDOS 8 Technical Note #20, Mirrored Devices and SmartPort

END OF FILE TN.PDOS.021

###
FILE: TN.PDOS.022
###

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#22: Don't Put Parameter Blocks on Zero Page

Written by: Dave Lyons July 1989

Putting ProDOS 8 parameter blocks on zero page ($00-$FF) is not recommended.

It is not a good idea to put the parameter blocks for ProDOS 8 MLI calls on zero page. This is not forbidden by the ProDOS 8 Technical Reference Manual, but then again, it also doesn't tell you not to put parameter blocks in ROM, in the $C0xx soft switch area, or just below the active part of the stack.

If you do put MLI parameter blocks on zero page, your application may break in the future.

If your parameter block comes between $80 and $FF, it won't work with AppleShare installed.

Further Reference

---

o ProDOS 8 Technical Reference Manual

END OF FILE TN.PDOS.022

```
#####################################################################
### FILE: TN.PDOS.023
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#23:   ProDOS 8 Changes and Minutia

Written by: Matt Deatherage   July 1989

This Technical Note documents the change history of ProDOS 8 through V1.8, and it supersedes the information on this topic in the ProDOS 8 Technical Reference Manual and the ProDOS 8 Update.

Changes? You're kidding.

No. One of the side effects of evolving technology is that eventually little things (like the disk operating system) have to change to support the new technologies. Every time Apple changes ProDOS 8, the manuals can't be reprinted. For one thing, it takes a long time to turn out a manual, by which time there's often a new version done which the new manual doesn't cover. For another thing, programmers and developers don't tend to purchase revised manuals (our informal research shows that more people have up-to-date Apple /// RPS documentation than have up-to-date Apple IIc documentation--and this was done before the Apple IIc Plus was released...).

So this Note explains what has changed between ProDOS 8 V1.0 and the current release, V1.8, which began shipping with System Software 5.0. Table 1 shows what versions of ProDOS 8 existing documentation covers.

Document	Version Number
ProDOS 8 Technical Reference Manual	1.1.1
ProDOS 8 Update	1.4
AppleShare Programmer's Guide to the Apple IIGS	1.5

ProDOS 1.0

This was the first release of ProDOS, which was so unique it didn't even have to be called ProDOS 8 to distinguish it from ProDOS 16, which we're not talking about. If you have documentation that predates ProDOS 1.0, you should seek professional help from APDA at the address listed in Technical Note #0.

ProDOS 1.0.1

- Fixed a bug in the STATUS call which affected testing for the write-protected condition.

ProDOS 1.0.2

- o Changed instructions used in interrupt entry routines on the global page so the accumulator would not be destroyed.
- o Fixed a bug in the Disk II core routines so the motor would shut off after recalibration on an error.

ProDOS 1.1

- o Changed the internal MLI layout for future expandability and maintenance.
- o Modified machine ID routines to identify IIc and enhanced IIe ROMs.
- o Removed code that allowed ProDOS to boot on 48K machines.
- o Removed the check for the ProDOS version number from the OPEN routine.
- o Incremented KVERSION (the ProDOS Kernel version) on the global page.
- o Modified the loader routines to reflect the presence of any 80-column card following the established protocol (see ProDOS 8 Technical Note #15, How ProDOS 8 Treats Slot 3). Also, at this time, added code to allow slot 3 to be enabled on a IIe if an 80-column card following the protocol was found.
- o Added code to turn off all disk motor phases prior to seeking a track in the Disk II driver.
- o Fixed a bug to prevent accesses to /RAM after it had been removed from the device list.
- o Reduced the size of the /RAM device by one block to protect interrupt vectors in the auxiliary language card. The correct vectors are installed at boot time.

ProDOS 1.1.1

- o Fixed a Disk II driver bug for mapping into drive 1.
- o Modified machine ID routines to give precedence to identifiable 80-column cards in slot 3.

ProDOS 8 1.2

- o Changed the name from ProDOS to ProDOS 8 to avoid confusion with ProDOS 16, which, again, this Note does not discuss.
- o Introduced the clock driver for the Apple IIGS. The machine identification code was changed to indicate the presence of the clock on the IIGS.
- o Added preliminary network support by adding the network call and preliminary network driver space.
- o Fixed a bug in returning errors from calls to the RAM disk. Changed the RAM disk driver to return values of zero on reads and ignore writes to blocks zero, one, four, five, six, and seven, which are not accessible as storage in the driver's design.
- o Added a new system error ($C) for errors when deallocating blocks from a tree file.
- o Fixed a bug in zeroing a Volume Control Block (VCB) when trying to reallocate a previously used VCB.
- o Modified the ProDOS 8 loader code to automatically install up to four drives in slot 5 if a SmartPort device is found. Removed the code to always leave interrupts disabled, which leaves the state of the interrupt flag at boot time unchanged while ProDOS 8 loads.

- o Changed the MLI entry to disable interrupts until after the MLIACTV flag is set and other ProDOS parameters are initialized.
- o Modified the QUIT code to allow the Delete key to function the same as the left arrow key. Also fixed a bug so screen holes would not be trashed in 80-column mode. Crunched code to allow soft switch accesses to force 40-column text mode. Fixed a bug so the dispatcher would not trash the screen when executed with a NIL prefix.
- o Modified the ONLINE call so that it could be made to a device that had just been removed from the device list by the standard protocol. Previous to this change, a VCB for the removed device was left, reducing the number of online volumes by one for each such device. From this point on, removing a device should be followed by an ONLINE call to the device just removed. The call returns error $28 (No Device Connected), but deallocates the VCB.
- o Added a spurious interrupt handler to allow up to 255 unclaimed interrupts before system death.
- o Removed the code which invoked low-resolution graphics on system death--it had not worked well and the space was needed. The system had previously had the ability to display "INSERT SYSTEM DISK AND RESTART" without also displaying "-ERR xx", which was removed at this point for space reasons since the system wasn't using it (and hopefully you weren't, either, since it wasn't documented).
- o Changed MLIACTV to use an ASL instead of an LSR to turn "off" the flag.
- o Changed the OPEN call to correctly return error $4B (Unsupported Storage Type) instead of error $4A (incompatible file format for this version) when attempting to open a file with an unrecognized storage type.
- o Fixed an obscure bug involving READ in Newline mode. If the requested number of bytes was greater than $FF, and the number of bytes in the file after the newline character was read was a multiple of $100, then the number of bytes reported transferred by ProDOS was equal to the correct number of transferred bytes plus $100.
- o Starting with V1.2 on an Apple IIGS, stopped switching slot 3 ROM space and left the determination of whether the slot or the port was enabled to the Control Panel; however, there was a bug in this implementation which was fixed in V1.7 and described in ProDOS 8 Technical Note #15, How ProDOS 8 Treats Slot 3.
- o Updated the slot-based clock driver's year table through 1991.
- o Added a feature which allows ProDOS 8 to search for a file named ATINIT in the boot volume's root directory, to load and execute it, then to proceed normally with the boot process by loading the first .SYSTEM file. No error occurs if the ATINIT file is not found, but any other error condition (including the file existing and not having file type $E2) causes a fatal error.
- o Changed loader code so ProDOS 8 could be loaded by ProDOS 16 without automatically executing the ATINIT and the first .SYSTEM file.
- o Changed the device search process in the ProDOS 8 loader so SmartPort devices are only installed if they actually exist, and Disk IIs are placed with lowest priority in the device list so they are scanned last.
- o Forced Super Hi-Res off on an Apple IIGS when a fatal error occurs. (Actually, this did not work, but it was fixed in V1.7.)
- o Inserted a patch to fix a bug in the first IIGS ROM that caused internal $Cn00 ROM space to be left mapped in if SmartPort failed

to boot.

ProDOS 8 1.3

Warning: This is not a stable version of ProDOS due to an illegal 65C02 instruction which was added. This version can damage disks if used with a 6502 processor.

- o Changed the code that resets phase lines for Disk IIs so phase clearing is done with a load instead of a store, since stores to even numbered locations cause bus contention, which is major uncool. Changed the routine to force access to all eight even locations, which not only clears the phases, but also forces read mode, first drive, and motor off. DOS used to do this; ProDOS had not been doing it. If L7 had been left on when the Disk II driver was called and it checked write-protect with L6 high, write mode was enabled. Forcing read mode leaves less to chance.
- o Changed deallocation of index blocks so index blocks are not zeroed, allowing the use of file recovery utilities. Instead, index blocks are "flipped" (the first 256 bytes are exchanged with the last 256 bytes).
- o Since the UniDisk 3.5 interface card for the][+ and IIe does not set up its device chain unless a ProDOS call is made to it, ProDOS STATUS calls are now made to the device before SmartPort STATUS calls.

ProDOS 8 1.4

- o Removed an illegal 65C02 instruction which was added in V1.3.
- o Modified the Disk II driver so a routine that should only clear the phase lines only clears the phase lines. Also clear Q7 to prevent inadvertent writes.

ProDOS 8 1.5

- o ProDOS 8 1.5 is the first version to include network support through the ProDOS Filing Interface (PFI) as part of ProDOS 16 or on the Apple IIe Workstation Card. Made many changes to internal routines for PFI location and compatibility at this point. Crunched and moved code for PFI booting and accessibility.
- o Changed some strings to all uppercase internally for string comparisons.
- o Removed the generic $42 AppleTalk call which was introduced in V1.2, as PFI gets called through the global page.
- o Changed the ASL to clear the MLIACTV flag back to an LSR. This doesn't make nested levels of busy states possible, but always clears the flag before calling interrupt handling routines that check MLIACTV as described in the ProDOS 8 Technical Reference Manual.
- o If an Escape key is detected in the keyboard buffer on an Apple IIc, it is removed. This is friendly to the Apple IIc Plus, the ROM of which does not remove the Escape key it uses to detect that the system should be booted at normal speed.

ProDOS 8 1.6

- o Set up a parallel pointer to correct a PFI misinterpretation of an internal MLI pointer.

ProDOS 8 1.7

- o Made a change to ensure that ProDOS 8 counts the volume's bitmap before incrementing the number of free blocks. This fixed a bug where an uninitialized location was being incremented and decremented, incorrectly reporting a Disk Full error where none should have occurred.
- o Changed the handling of slot 3 ROM space to that described in ProDOS 8 Technical Note #15, How ProDOS 8 Treats Slot 3.
- o Changed code to permit the invisible bit of the access byte (bit 2) to be set by applications.

ProDOS 8 1.8

- o Fixed a bug introduced in V1.3. If an error occurs while calling DESTROY on a file, the file is not deleted but the index blocks are not swapped back to normal position. If a subsequent DESTROY of the same file succeeds, the volume's integrity is destroyed. Now ProDOS 8 marks the file as deleted, even if an error occurs, so any other errors will not cause a subsequent MLI call to trash the volume. Note that "undelete" utilities attempting to undelete such a file (one in which an error occurred during the DESTROY) probably will trash the volume.
- o Fixed the ONLINE call to ignore the unused low nibble of the unit_num parameter when deciding how many bytes to zero in the application's buffer. This change fixes a bug which zeroed only the first 16 bytes of the caller's buffer before filling them if an ONLINE call was made with a unit_num of $0X, where X is non-zero.
- o When loading on an Apple IIGS, ProDOS 8 now sets the video mode so the 80-column firmware is not active when the ProDOS 8 application gets control.
- o Changed internal version checking between GS/OS and ProDOS 8. Note that GS/OS and ProDOS 8 are still tied to each other--versions that didn't come on the same disk can't be used together. The methods for checking versions were just altered.
- o Made the backward compatibility check when opening subdirectories inactive. The test would always fail when opening a subdirectory with lowercase characters in the name (as assigned by the ProDOS FST under GS/OS), so the check was removed. Note that using earlier versions of ProDOS 8 with such disks will cause errors when trying to access files with such directories in their pathnames.
- o Expanded the ProDOS 8 loader code to provide for more room for future compatibility.

Further Reference

- o ProDOS 8 Technical Reference Manual
- o ProDOS 8 Update
- o AppleShare Programmer's Guide to the Apple IIGS

END OF FILE TN.PDOS.023

```
#####################################################################
### FILE: TN.PDOS.024
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#24: BASIC.SYSTEM Revisions

Written by: Matt Deatherage July 1989

This Technical Note documents the change history of BASIC.SYSTEM through V1.3, which ships with System Software 5.0. V1.0, the initial release, is not documented in this Note, and V1.1 is described in BASIC Programming with ProDOS.

V1.1

- Fixed a bug in variable packing (used by CHAIN, STORE, and RESTORE).
- Changed the interpreter to use the ProDOS startup convention of a JMP instruction followed by two $EE bytes and a startup pathname buffer.
- Removed a bad buffer address in the FIELD parameter of the READ routine.
- Fixed a bug in APPEND so calls to OPEN and READ from a random-access file would not cause the next call to APPEND to any file to use the record length of the random-access file.
- Added the BYE command to allow ProDOS QUIT calls from BASIC.
- Removed the limited support for run-time capabilities which had been present.

V1.2

- Changed the CATALOG command to ignore the number of entries in a directory when listing it so AppleShare volumes could be cataloged properly (this number can change on the fly on an AppleShare volume).
- Fixed another bug in CATALOG so pressing an unexpected key when a catalog listing was paused with a Control-S would no longer abort the catalog.

V1.3

- Changed BSAVE so it now truncates the length of the saved file when the B parameter is not used. To replace the first part of a file without truncation, use the B parameter with a value of zero. This behavior with the B parameter is how V1.1 and V1.2 worked without the B parameter.
- Fixed a bug in CHAIN and STORE where they expected one branch to go two ways at the same time.
- Added the MTR command for easier access to the Monitor from BASIC.

- o Made internal changes to the assembly process for easier project management. These changes do not affect the code image.

Further Reference

- o BASIC Programming with ProDOS
- o ProDOS 8 Technical Reference Manual

END OF FILE TN.PDOS.024

###
FILE: TN.PDOS.025
###

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#25: Non-Standard Storage Types

Written by: Matt Deatherage April 1989

This Technical Note discusses storage types for ProDOS files which are not documented in the ProDOS 8 Technical Reference Manual.

Warning: The information provided in this Note is for the use of disk utility programs which occasionally must manipulate non-standard files in unusual situations. ProDOS 8 programs should not create or otherwise manipulate files with non-standard storage types.

Introduction

One of the features of the ProDOS file system is its ability to let ProDOS 8 know when someone has put a file on the disk that ProDOS 8 can't access. A file not created by ProDOS 8 can be identified by the storage_type field. ProDOS 8 creates four different storage types: seedling files ($1), sapling files ($2), tree files ($3), and directory files ($D). ProDOS 8 also stores subdirectory headers as storage type $E and volume directory headers as storage type $F. These are all described in the ProDOS 8 Technical Reference Manual.

Other files may be placed on the disk, and ProDOS 8 can catalog them, rename them, and return file information about them. However, since it does not know how the information in the files is stored on the disk, it cannot perform normal file operations on these files, and it returns the Unsupported Storage Type error instead.

Apple reserves the right to define additional storage types for the extension of the ProDOS file system in the future. To date, two additional storage types have been defined. Storage type $4 indicates a Pascal area on a ProFile hard disk, and storage type $5 indicates a GS/OS extended file (data fork and resource fork) as created by the ProDOS FST.

Storage Type $4

Storage type $4 is used for Apple II Pascal areas on Profile hard disk drives. These files are created by the Apple Pascal ProFile Manager. Other programs should not create these files, as Apple II Pascal could freak out.

The Pascal Profile Manager (PPM) creates files which are internally divided into pseudo-volumes by Apple II Pascal. The files have the name PASCAL.AREA (name length of 10), with file type $EF. The key_pointer field of the directory entry points to the first block used by the file, which is the second to last block on the disk. As ProDOS stores files non-contiguously up from the bottom, PPM creates pseudo-volumes contiguously down from the end of the ProFile. Blocks_used is 2, and header_pointer is also 2. All other fields in the directory are set to 0. PPM looks for this entry (starting with the name PASCAL.AREA) to determine if a ProFile has been initialized for Pascal use.

The file entry for the Pascal area increments the number of files in the ProDOS directory and the key_pointer for the file points to TOTAL_BLOCKS - 2, or the second to last block on the disk. When PPM expands or contracts the Pascal area, blocks_used and key_pointer are updated accordingly. With any access to this entry (such as adding or deleting pseudo-volumes within PPM), the backup bit is not set (PPM provides a utility to back up the Pascal area).

The Pascal volume directory contains two separate contiguous data structures that specify the contents of the Pascal area on the Profile. The volume directory occupies two blocks to support 31 pseudo-volumes. It is found at the physical block specified in the ProDOS volume directory as the value of key_pointer (i.e., it occupies the first block in the area pointed to by this value).

The first portion of the volume directory is the actual directory for the pseudo-volumes. It is an array with the following Apple II Pascal declaration:

```
TYPE    RTYPE = (HEADER, REGULAR)

VAR     VDIR:    ARRAY [0..31] OF
            PACKED RECORD
                CASE RTYPE OF
                    HEADER:    (PSEUDO_DEVICE_LENGTH:INTEGER;
                               CUR_NUM_VOLS:INTEGER;
                               PPM_NAME:STRING[3]);
                    REGULAR:   (START:INTEGER;
                               DEFAULT_UNIT:0.255
                               FILLER:0..127
                               WP:BOOLEAN
                               OLDDRIVERADDR:INTEGER
            END;
```

The HEADER specifies information about the Pascal area. It specifies the size in blocks in PSEUDO_DEVICE_LENGTH, the number of currently allocated volumes in CUR_NUM_VOLS, and a special validity check in PPM_NAME, which is the three-character string PPM. The header information is accessed via a reference to VDIR[0]. The REGULAR entry specifies information for each pseudo-volume. START is the starting block address for the pseudo-volume, and LENGTH is the length of the pseudo-volume in blocks. DEFAULT_UNIT specifies the default Pascal unit number that this pseudo-volume should be assigned to upon booting the system. This value is set through the Volume Manager by either the user or an application program, and it remains valid if it is not released.

If the system is shut down, the pseudo-volume remains assigned and will be active once the system is rebooted. WP is a Boolean that specifies if the pseudo-volume is write-protected. OLDDRIVERADDR holds the address of this unit's (if assigned) previous driver address. It is used when normal floppy unit numbers are assigned to pseudo-volumes, so when released, the floppies can be reactivated. Each REGULAR entry is accessed via an index from 1 to 31. This index value is thus associated with a pseudo-volume. All references to pseudo-volumes in the Volume Manager are made with these indexes.

Immediately following the VDIR array is an array of description fields for each pseudo-volume:

```
    VDESC:    ARRAY [0..31] OF STRING[15]
```

The description field is used to differentiate pseudo-volumes with the same name. It is set when the pseudo-volume is created. This array is accessed with the same index as VDIR.

The volume directory does not maintain the names of the pseudo-volumes. These are found in the directories in each pseudo-volume. When the Volume Manager is activated, it reads each pseudo-volume directory to construct an array of the pseudo-volume names:

```
    VNAMES:   ARRAY [0..31] OF STRING[7]
```

Each pseudo-volume name is stored here so the Volume Manager can use it in its display of pseudo-volumes. The name is set when the pseudo-volume is created and can be changed by the Pascal Filer. The names in this array are accessed via the same index as VDIR. This array is set up when the Volume Manager is initialized and after there is a delete of a pseudo-volume. Creating a pseudo-volume will add to the array at the end.

Pascal Pseudo-Volume Format

Each Pascal pseudo-volume is a standard UCSD formatted volume. Blocks 0 and 1 are reserved for bootstrap loaders (which are irrelevant for pseudo-volumes). The directory for the volume is in blocks 2 through 5 of the pseudo-volume. When a pseudo-volume is created, the directory for that pseudo-volume is initialized with the following values:

```
dfirstblock = 0            first logical block of the volume
dlastblock = 6             first available block after the directory
dvid    = name of the volume used in create
deovblk = size of volume specified in create
dnumfiles = 0              no files yet
dloadtime = set to current system date
dlastboot = 0
```

The Apple II Pascal 1.3 Manual contains the format for the UCSD directory. Files within this subdirectory are allocated via the standard Pascal I/O routines in a contiguous manner.

Storage Type $5

Storage type $5 is used by the ProDOS FST in GS/OS to store extended files. The key block of the file points to an extended key block entry. The extended key block entry contains mini-directory entries for both the data fork and resource fork of the file. The mini-entry for the data fork is at offset +000 of the extended key block, and the mini-entry for the resource fork is at offset +$100 (+256 decimal).

The format for mini-entries is as follows:

```
storage_type    (+000)    Byte    The standard ProDOS storage
                                  type for this fork of the file.
                                  Note that for regular directory
                                  entries, the storage type is the
                                  high nibble of a byte that contains
                                  the length of the filename as the
                                  low nibble.  In mini-entries, the
                                  high nibble is reserved and must be
                                  zero, and the storage type is
                                  contained in the low nibble.
key_block       (+001)    Word    The block number of the key
                                  block of this fork.  This value and
                                  the value of storage_type combine to
                                  determine how to find the data in
                                  the file, as documented in the
                                  ProDOS 8 Technical Reference Manual.
blocks_used     (+003)    Word    The number of blocks used by
                                  this fork of the file.
EOF             (+005)    3 Bytes Three-byte value (least
                                  significant byte stored first)
                                  representing the end-of-file value
                                  for this fork of the file.
```

All remaining bytes in the extended key block are reserved and must be zero.

Further Reference

- o Apple II Pascal ProFile Manager Manual
- o GS/OS Reference
- o ProDOS 8 Technical Reference Manual

END OF FILE TN.PDOS.025

##
FILE: TN.SmPt.001
##

Apple II
Technical Notes

Developer Technical Support

SmartPort
#1: SmartPort Introduction

Revised by: Matt Deatherage November 1988
Written by: Mike Askins November 1985

This Technical Note formerly introduced the SmartPort firmware interface.

This Note formerly contained a general introduction to the SmartPort firmware interface. Information on SmartPort as found in the Apple IIe and IIc is now found in the Apple IIc Technical Reference Manual.

For a more complete reference on SmartPort, including information on Extended SmartPort (for peripherals which can address more than one 64K bank of memory) and its parameters, please see chapter 7 of the Apple IIGS Firmware Reference.

Further Reference
- Apple IIGS Firmware Reference
- Apple IIc Technical Reference Manual

END OF FILE TN.SmPt.001

```
######################################################################
### FILE: TN.SmPt.002
######################################################################
```

Apple II
Technical Notes

Developer Technical Support

SmartPort
#2: SmartPort Calls Updated

Revised by: Llew Roberts — September 1989
Written by: Mike Askins — May 1985

This Technical Note documents SmartPort call information which is not found in the descriptions of SmartPort in the Apple IIGS Firmware Reference and the Apple IIc Technical Reference Manual. The device-specific information which had been included in this Note is now found in these manuals.
Changes since November 1988: Added diagram and information on vendor ID numbers.

STATUS Calls

A STATUS call with unit number = $00 and status code = $00 is a request to return the status of the SmartPort host, as opposed to unit numbers greater than zero which return the status of individual devices. The number of devices as well as the current interrupt status is returned. The format of the status list returned is illustrated in Figure 1.

```
           +-------------------+
    Byte 0 |   Device Count    |
           +-------------------+
    Byte 1 | Interrupt Status  |
           +-------------------+
    Byte 2 |      Vendor       |   $0000         Vendor unknown
           +                   +---$0001         Apple Computer, Inc.
    Byte 3 |        ID         |   $0002-$FFFF   Third-Party Vendor
           +-------------------+
    Byte 4 |     Interface     |   _____|___________________|_____
           +                   +--|F|E|D|C|B|A|9|8|7|6|5|4|3|2|1|0|
    Byte 5 |      Version      |  |_______|_______________|_______|
           +-------------------+  |       |               |
    Byte 6 |     Reserved      |  |Major  |    Minor      |$A=Alpha
           +-------------------+  |Release|    Release    |$B=Beta
    Byte 7 |     Reserved      |                          |$E=Experimental
           +-------------------+                          |$0=Final
```

Figure 1-Host General Status Return Information

```
Stat_list    byte 0       Number of devices
             byte 1       Interrupt Status (If bit 6 is set, then no interrupt)
             bytes 2-3    Driver manufacturer (were Reserved prior to May
                          1988):
```

```
                         $0000          Undetermined
                         $0001          Apple
                         $0002-$FFFF    Third-party driver
             bytes 4-5   Interface Version
             bytes 6-7   Reserved (must be $0000)
```

The Number of devices byte tells the caller the total number of devices hooked to this slot or port.

The Interrupt Status byte is used by programs which try to determine if the SmartPort was the source of an interrupt. If bit 6 of this byte is clear, there is a device (or devices) in the chain that require interrupt service. You cannot use this value to determine which device in the chain is actually interrupting. Your interrupt handler, having determined that a SmartPort interrupt has occurred, must poll each device on the chain to find out which device requires service. The UniDisk 3.5 and Memory Expansion Card do not generate interrupts, so in these cases, this byte has bit 6 set.

The vendor ID number may be used to determine the manufacturer of a specific SmartPort peripheral interface card, a useful piece of information when dealing with device-specific calls. Contact Apple Developer Technical Support if you require a specific vendor ID number. The version word follows the SmartPort Interface Version definition described later in this Note.

CONTROL Codes

Before May 1988, control code $04 was defined as device-specific. It is now defined as EJECT, and all SmartPort devices which support removable media must support this call. If a device does not support removable media, it should simply return from this call without an error.

Note that the Apple II SCSI card firmware was revised in early 1988 to support this change.

INIT

An application should never make an INIT call (SmartPort code $05), since doing so is likely to destroy operating system integrity and may cause media damage as well.

If you are writing your own operating system (not encouraged) and need to reset all SmartPort devices, the INIT call with unit number = $00 will do just that. Note that SmartPort devices cannot be selectively reset, and INIT must never be made at all with any unit number other than $00.

SmartPort Interface Version Definition

The SmartPort Interface Version definition uses the most significant nibble of the word as the major version number, the next two most significant nibbles as the minor version number, and the least significant nibble as a release indicator:

```
    $0 = Final    $A = Alpha    $B = Beta    $E = Experimental
```

Therefore, the interface version word for an experimental SmartPort interface

1.15 would be $115E while the interface version word for SmartPort interface 2.0 would be $2000. GS/OS driver version numbers also follow this definition.

Further Reference

- Apple IIGS Firmware Reference
- Apple IIc Technical Reference Manual
- Apple IIGS Technical Note #25, Apple IIGS Firmware Reference Updates

END OF FILE TN.SmPt.002

##
FILE: TN.SmPt.003
##

Apple II
Technical Notes

Developer Technical Support

SmartPort
#3: SmartPort Bus Architecture

Revised by: Matt Deatherage November 1988
Written by: Mike Askins March 1985

This Technical Note formerly described the SmartPort Bus architecture, but this information is now documented in the Apple IIGS Firmware Reference.

Do not be confused by the name "SmartPort Bus" architecture. The information in the Apple IIGS Firmware Reference describes the mechanics of how devices interface with the disk port on a IIGS or IIc and with the UniDisk 3.5 Interface card on a][+ or IIe. It is not necessary to understand this information to use SmartPort firmware calls, nor do all devices which have SmartPort firmware necessarily have to connect mechanically through the disk port or UniDisk 3.5 Interface card.

The physical or electrical side of the hardware is called the "SmartPort Bus," while the firmware protocols are called the "SmartPort Interface." Although the term "SmartPort" can refer to either or both parts, it is most often used to refer to the SmartPort Interface. Only those developers who are designing products which will attach to either the IIGS or IIc disk port or to the UniDisk 3.5 Interface card need be concerned with the SmartPort Bus architecture. Software developers need not learn about the SmartPort Bus architecture to use the SmartPort Interface firmware.

Further Reference
o Apple IIGS Firmware Reference

END OF FILE TN.SmPt.003

##
FILE: TN.SmPt.004
##

Apple II
Technical Notes

Developer Technical Support

SmartPort
#4: SmartPort Device Types

Revised by: Matt Deatherage — November 1988
Written by: Rilla Reynolds — June 1987

This Technical Note documents additional device types which the SmartPort firmware recognizes, but which may not be currently documented in the technical reference manuals which cover SmartPort.

The following is an updated list of possible SmartPort device types, extended to support an increasing variety of third-party peripheral products. A device type byte is returned as part of the Device Information Block (DIB) from a SmartPort STATUS call ($03).

Type	Device
$00	Memory Expansion Card (RAM disk)
$01	3.5" disk
$02	ProFile-type hard disk
$03	Generic SCSI
$04	ROM disk
$05	SCSI CD-ROM
$06	SCSI tape or other SCSI sequential device
$07	SCSI hard disk
$08	Reserved
$09	SCSI printer
$0A	5-1/4" disk
$0B	Reserved
$0C	Reserved
$0D	Printer
$0E	Clock
$0F	Modem

It is likely that the SmartPort device type list will expand in the future. If you are developing a SmartPort device and do not see a suitable device type in the list, contact Apple II Developer Technical Support at the address listed in Technical Note #0.

Further Reference
- o Apple IIGS Firmware Reference
- o Apple IIc Technical Reference Manual

END OF FILE TN.SmPt.004

```
#####################################################################
### FILE: TN.SmPt.005
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

SmartPort
#5: SCSI SmartPort Call Changes

Revised by: Llew Roberts — January 1989
Written by: Rilla Reynolds & Matt Deatherage — May 1988

This Technical Note describes two CONTROL codes which have changed in revision C of the Apple II SCSI card firmware.
Changes since November 1988: Added compatibility guidelines for future SCSI products.

Revision C of the Apple II SCSI card firmware includes two CONTROL code changes.

CONTROL code $04, previously defined as FORMAT, is now defined as EJECT. This change reflects the revised SmartPort requirement that all devices maintain CONTROL code $04 as EJECT. See SmartPort Technical Note #2, SmartPort Calls Updated, for more information.

CONTROL code $15 is now defined as FORMAT instead of RESERVED. Note that there are two EJECT calls in this version, as CONTROL code $26 is still defined as EJECT.

To determine which version of the SCSI ROM is on any particular Apple II SCSI Interface Card, issue a $03 SmartPort STATUS call. The revision C SCSI ROM will return the word $0200. This does not follow the SmartPort Interface Version scheme described in SmartPort Technical Note #2. However, future revisions of the Apple II SCSI card will follow this scheme. Therefore, applications should expect any SmartPort SCSI firmware to behave as described in this Note if the version number is $0200 or if it is greater than or equal to $2000.

To maintain compatibility with future Apple II SCSI products, you should use the following guidelines:

- Avoid access to the hardware or any RAM locations on the SCSI card.
- Do not use the Patch1Call, SetNewSDAT, or SetBlockSize control calls.
- For devices with a block size other than 512 bytes, use the SmartPort Read and Write calls. Do not use ReadBlock and WriteBlock calls for these devices, since they only read or write the first 512 bytes of a block. The Read and Write calls may also be used for devices with a 512-byte block size.
- Never Reset the SCSI bus.

The Apple II SCSI Card firmware was designed to operate with SCSI CD-ROM and disk drives only.

Further Reference

- o Apple II SCSI Card Technical Reference
- o SmartPort Technical Note #2, SmartPort Calls Updated

END OF FILE TN.SmPt.005

##
FILE: TN.SmPt.006
##

Apple II
Technical Notes

Developer Technical Support

SmartPort
#6: Apple IIGS SmartPort Errata

Written by: Matt Deatherage November 1988

This Technical Note documents two bugs in the Apple IIGS SmartPort firmware.

Developers should be aware of the following two bugs in the Apple IIGS SmartPort firmware:

1. SmartPort accidentally uses locations $57 through $5A on the zero page without saving and restoring them first. There is some confusion as to whether these bytes are used on the absolute zero page or on the caller's direct page. This is a moot point--SmartPort calls are required to be made from full-emulation mode. This requirement means the emulation bit must be set and the data bank and direct page registers must both be set to zero. The bytes are used on the absolute zero page, as that should be the direct page when SmartPort is called.
2. If an extended SmartPort CONTROL call is made, the CONTROL list must not start at $FFFE or $FFFF of any bank. The IIGS SmartPort interface does not increment the bank pointer when moving past the two-byte CONTROL list length. If a CONTROL list starts one or two bytes before a bank boundary, SmartPort will incorrectly read the list from the beginning of that bank, instead of the beginning of the next bank.

These bugs will be fixed in any future release of the Apple IIGS firmware.

Further Reference
o Apple IIGS Firmware Reference

END OF FILE TN.SmPt.006

```
#####################################################################
### FILE: TN.SmPt.007
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

SmartPort
#7: SmartPort Subtype Codes

Written by: Matt Deatherage November 1988

This Technical Note clarifies information about SmartPort subtype codes.

Following is a definition of the SmartPort subtype code as given in the Apple IIGS Firmware Reference:

```
 _________________________________________________
|     |     |     |     |     |     |     |     |     |
|  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
|     |     |     |     |     |     |     |     |     |
 _________________________________________________
   |     |     |     |     |     |     |     |
   |     |     |     |_____|_____|_____|_____|___ Reserved
   |     |     |___________________________________ 0 = Removable Media
   |     |_________________________________________ 1 = Supports disk-switched
   |                                                    errors
   |_______________________________________________ 1 = Supports Extended
                                                        SmartPort
```

Figure 1-SmartPort Subtype Byte

Note that the value for subtype is defined for certain characteristics of the device; it is not assigned to the device as with Smartport device types (see SmartPort Technical Note #4, SmartPort Device Types for a complete list).

Attempting to distinguish different kinds of the same device by the subtype field can be confusing. For example, the Apple IIc Plus has an internal 3.5" disk drive. This drive does not support disk-switched errors nor does it support Extended SmartPort, and it has removable media. This combination of features gives it a subtype definition of $00. However, this is the same subtype returned for a UniDisk 3.5. Any program which finds type $01 (3.5" Disk) and subtype $00 and assumes the drive is a UniDisk 3.5 will be misled by any other 3.5" drive matching the characteristics of the UniDisk 3.5.

Some Apple technical manuals state that the subtype byte may be used for identification purposes, but this cannot be supported if more than one variety of a specific device has the same characteristics and subtype.

To determine if a particular device type is the subtype you want, you may examine the name returned in the Device Information Block (DIB) from a STATUS call with statcode = 3. For 3.5" drives, however, this is not too helpful (both a UniDisk 3.5 and an Apple 3.5 Drive return DISK 3.5).

Because the subtype can not conclusively identify different flavors of 3.5" drives (and perhaps other individual device types), applications must look for errors on device specific calls and respond appropriately. Typical errors returned from making a device-specific call to the wrong device are $21 (BADCTL) and $22 (BADCTLPARM), although these are not the only ones. Also note that error codes in the range $20 - $2F are duplicated as $60 - $6F, the difference being that codes in the latter range are returned if the error was a soft error--a non-fatal error returned when the operation is completed successfully but an abnormal condition is detected.

The Reserved fields in the SmartPort subtype byte are reserved for future expansion. Present peripherals must have them set to zero so that they will not appear to support future features which are not presently defined. For this reason, programs checking the status of bits in the subtype byte should do so on a bit-by-bit basis only. For example, if you need to know if a device supports Extended Smartport, mask off all bits except bit 7 in the subtype byte before doing any comparisons. Blindly comparing to existing common subtype values (like $00 and $C0) will cause comparisons to fail when future bits in the subtype byte are defined.

Further Reference
- SmartPort Technical Note #4, SmartPort Device Types

END OF FILE TN.SmPt.007

```
########################################################################
### FILE: TN.SmPt.008
########################################################################
```

Apple II
Technical Notes

Developer Technical Support

SmartPort
#8: SmartPort Packets

Written by: Llew Roberts May 1989

This Technical Note describes the structure and timing of a sample SmartPort packet.

SmartPort devices communicate using SmartPort packets. The following packet shows the timing and content of a SmartPort READBLOCK call. For further explanation of the structure, please see the Apple IIGS Hardware Reference and the Apple IIGS Firmware Reference.

Note: The CPU will recognize and act on any packet put on the bus by a SmartPort Device.

DATA (SmartPort Bus)	MNEMONIC	DESCRIPTION	TIME (Relative)
FF	SYNC	SELF SYNCHRONIZING BYTES	0
3F	:	:	32 micro Sec.
CF	:	:	32 micro Sec.
F3	:	:	32 micro Sec.
FC	:	:	32 micro Sec.
FF	:	:	32 micro Sec.
C3	PBEGIN	MARKS BEGINNING OF PACKET	32 micro Sec.
81	DEST	DESTINATION UNIT NUMBER	32 micro Sec.
80	SRC	SOURCE UNIT NUMBER	32 micro Sec.
80	TYPE	PACKET TYPE FIELD	32 micro Sec.
80	AUX	PACKET AUXILLIARY TYPE FIELD	32 micro Sec.
80	STAT	DATA STATUS FIELD	32 micro Sec.
82	ODDCNT	ODD BYTES COUNT	32 micro Sec.
81	GRP7CNT	GROUP OF 7 BYTES COUNT	32 micro Sec.
80	ODDMSB	ODD BYTES MSB's	32 micro Sec.
81	COMMAND	1ST ODD BYTE = Command Byte	32 micro Sec.
83	PARMCNT	2ND ODD BYTE = Parameter Count	32 micro Sec.
80	GRP7MSB	MSB's FOR 1ST GROUP OF 7	32 micro Sec.
80	G7BYTE1	BYTE 1 FOR 1ST GROUP OF 7	32 micro Sec.
98	G7BYTE2	BYTE 2 FOR 1ST GROUP OF 7	32 micro Sec.
82	G7BYTE3	BYTE 3 FOR 1ST GROUP OF 7	32 micro Sec.
80	G7BYTE4	BYTE 4 FOR 1ST GROUP OF 7	32 micro Sec.
80	G7BYTE5	BYTE 5 FOR 1ST GROUP OF 7	32 micro Sec.
80	G7BYTE5	BYTE 6 FOR 1ST GROUP OF 7	32 micro Sec.
80	G7BYTE6	BYTE 7 FOR 1ST GROUP OF 7	32 micro Sec.
BB	CHKSUM1	1ST BYTE OF CHECKSUM	32 micro Sec.
EE	CHKSUM2	2ND BYTE OF CHECKSUM	32 micro Sec.

C8	PEND	PACKET END BYTE	32 micro Sec.
00	FALSE	FALSE IWM WRITE TO CLEAR REGISTER	32 micro Sec.

Further Reference

- o Apple IIGS Hardware Reference
- o Apple IIGS Firmware Reference

END OF FILE TN.SmPt.008

##
FILE: TN.UDsk.001
##

Apple II
Technical Notes

Developer Technical Support

UniDisk 3.5
#1: UniDisk 3.5 Internals

Revised by: Matt Deatherage November 1988
Written by: Mike Askins May 1985

This Technical Note formerly described the internals of the UniDisk 3.5, and this information is now documented in the Apple IIGS Firmware Reference.

This Note formerly documented the internal structure of the UniDisk 3.5, primarily for those interested in providing copy protection. Apple Computer no longer supports copy protection schemes, and we strongly urge developers to make use of alternate methods to limit unauthorized duplication.

The internals of the UniDisk 3.5 are now documented in the Apple IIGS Firmware Reference.

Further Reference
o Apple IIGS Firmware Reference

END OF FILE TN.UDsk.001

```
#####################################################################
### FILE: TN.UDsk.002
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

UniDisk 3.5
#2: UniDisk 3.5 ID Bytes

Revised by: Matt Deatherage November 1988
Written by: Mike Askins May 1985

This Technical Note describes the signature bytes of the UniDisk 3.5.

The signature bytes for the UniDisk 3.5 are the same as those for any SmartPort device:

```
    $Cn01 = $20
    $Cn03 = $00        ProDOS Block Device
    $Cn05 = $03

    $Cn07 = $00        SmartPort Interface
```

where n is the slot number of the device.

When searching the slots for a UniDisk 3.5 it is very important to check all the signature bytes, since there are other peripherals with similar ID bytes. Once you find a SmartPort card (or port), you should do a SmartPort STATUS call to determine which devices are connected to it. Any number of different devices could match the SmartPort ID bytes, so trying to identify a device without making a SmartPort STATUS call is very likely to produce inaccurate results.

Why the UniDisk 3.5 Does Not Auto-Boot on Older Machines

If you look carefully, you will notice that the older (][,][+ and unenhanced IIe) Autostart Monitor will not boot any SmartPort device because the ID byte at $Cn07 = $00 instead of $3C (like the old Disk II). If Apple had left the ID bytes the same as the Disk II, then older versions of Apple II Pascal (1.2 and earlier) would assume that the drive was a Disk II.

Where This Leaves You

The enhanced IIe ROMs, as well as the UniDisk 3.5 IIc ROMs and later (which you have if you are using a UniDisk 3.5 on a IIc) check only the first three ID bytes. This check means that they will not only auto-boot the UniDisk 3.5, but any SmartPort or ProDOS block device. On an older machine, you can boot one of these devices by typing PR#n from AppleSoft or Cn00G from the Monitor.

Further Reference
o   Apple IIGS Firmware Reference

END OF FILE TN.UDsk.002

###
FILE: TN.UDsk.003
###

Apple II
Technical Notes

Developer Technical Support

UniDisk 3.5
#3: STATUS Call Bug

Revised by: Matt Deatherage November 1988
Written by: Mike Askins & Cameron Birse September 1984

This Technical Note documents a bug in the ProDOS STATUS call when used with a UniDisk 3.5.

The Bug

We have found that SmartPort does not return the WRITE PROTECT error on the STATUS call. (The WRITE call does return the WRITE PROTECT error as required.)

The bug manifests itself under ProDOS (and not under Pascal, since Pascal does not require the write protect error to be returned on the STATUS call). Specifically, if a write-protected disk is present in the UniDisk 3.5, and the application tries to write less than 512 bytes of data to a file that already exists on the media, it becomes impossible to finish the write or to close the file. Many applications ignore errors on close calls and try to reuse the buffer area which was presumably freed by the close call. This reuse results in further errors, even if the UniDisk 3.5 is later write-enabled, since ProDOS still thinks the file is open. This bug also decreases the maximum number of open files allowed, as the file left open is included in that number.

The bug also seems to cause the ProDOS CREATE call to fail. When a new file is created, opened and written to, and the write fails, the file manager does not deallocate the block that it reserved in the creation attempt. (The RAM copy of the bitmap seems to get trashed--GET_FILE_INFO calls at this point report that there are zero blocks available.) If you subsequently write enable the disk and do the save (with any size file), the file is written to the disk, and the bitmap is updated. The result is that there is a block reserved on the disk that no file owns, and that block cannot be freed through normal ProDOS file calls.

The Solution

Although this problem was fixed in later IIc revisions, the UniDisk 3.5 interface for the Apple][+ and IIe has never been modified. Therefore, if your application habitually performs the actions outlined above, you may avoid it by first checking to see if the media is write-protected instead of letting the buggy ProDOS STATUS call do it for you.

One way to accomplish this would be to issue a SmartPort STATUS call using a statcode = $00. This call returns four bytes of information, the first of which is the general status byte. This byte has the following format:

Bit	Meaning
7	0 = character device; 1 = block device
6	1 = write allowed
5	1 = read allowed
4	1 = device on line or disk in drive
3	0 = format allowed
2	0 = medium write protected (block devices only)
1	1 = device currently interrupting (Apple IIc only)
0	1 = device currently open (character devices only)

As shown in the table, bit 2 of this byte tells you what the ProDOS STATUS call cannot seem to figure out--the media in the drive is currently write-protected.

END OF FILE TN.UDsk.003

```
#####################################################################
### FILE: TN.UDsk.004
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

UniDisk 3.5
#4: Accessing Macintosh Disks

Revised by: Matt Deatherage November 1988
Written by: Mike Askins May 1985

This Technical Note formerly discussed drive-specific SmartPort calls. These calls are now documented in the Apple IIGS Firmware Reference. This Note now describes how to access Macintosh disks from a UniDisk 3.5 disk drive, as this information was not documented in the manual.

Macintosh Disk Access

The disk data format used in the UniDisk 3.5 is essentially identical to that used for Macintosh disks. There are three notable differences between the two formats:

- o Macintosh blocks are 524 bytes; UniDisk 3.5 blocks are 512 bytes.
- o Macintosh MFS disks are single sided; UniDisk 3.5 disks are double sided. (Macintosh HFS disks are double sided.)
- o The Macintosh uses a 2:1 physical block interleave; the UniDisk 3.5 uses a 4:1 interleave.

Accessing Blocks on a Macintosh Disk

Reading from a Macintosh disk is accomplished with the use of the READ command (as opposed to the READBLOCK command, which enforces 512 byte data.) A call to load block zero from the Macintosh disk in Unit #1 into memory at $2000 would look like this:

```
MacRead     JSR    Dispatch                 ;Normal SmartPort Entry point
            DFB    $08                      ;Character READ command code
            DW     Cmd_List                 ;The parameter list
            BCS    Error                    ;Optional error handling...
            ...
Cmd_List    DFB    $04                      ;CharRead has four parameters
            DFB    $01                      ;Unit number
            DW     $2000                    ;Buffer address
            DW     524                      ;Always transfer 524 bytes
            DFB    $00                      ;Block (lo)
            DFB    $00                      ;Block (med)
            DFB    $00                      ;Block (hi)
```

Writing to a Macintosh disk is accomplished with the use of the WRITE command. A call to write block zero to the Macintosh disk in Unit #1 with data at memory location $2000 would look like this:

```
MacWrite   JSR    Dispatch               ;Normal SmartPort Entry point
           DFB    $09                    ;Character WRITE command code
           DW     Cmd_List               ;The parameter list
           BCS    Error                  ;Optional error handling...
```

The Cmd_List is the same as in the READ example.

Formatting Macintosh Disks

The formatting routine in the UniDisk 3.5 firmware can format single- or double-sided disks of variable physical block interleave. The parameters controlling the interleave and the number of disk sides are located in the controller's zero page and are set to defaults whenever the INIT call is issued to SmartPort. These parameters can be altered by using the SET_DOWN_ADR and DOWNLOAD subcalls of the CONTROL call. Once altered, the FORMAT call uses these values in the formatting process. These zero page locations and their values are detailed below:

Parameter	Location	Values
Interleave	$0062	$02 = Mac, $04 = UniDisk 3.5
DoubleSided	$0063	$00 = Single, $80 = Double-sided

The following code example formats the media in Unit #1 as a Macintosh disk:

```
MacFormat  JSR    Dispatch               ;Set address to patch interleave
           DFB    $04                    ;Control call (Set_Down_Adr)
           DW     Cmd_ListA              ;Parameter List
           BCS    Error
;
           JSR    Dispatch               ;Now patch the interleave byte
           DFB    $04                    ;Control call (DOWNLOAD)
           DW     Cmd_ListB              ;Parameter List
           BCS    Error
;
           JSR    Dispatch               ;Set address to patch single sided
           DFB    $04                    ;Control call (Set_Down_Adr)
           DW     Cmd_ListC              ;Parameter List
           BCS    Error
;
           JSR    Dispatch               ;Now patch the single sided byte
           DFB    $04                    ;Control call (DOWNLOAD)
           DW     Cmd_ListD              ;Parameter List
           BCS    Error
;
           JSR    Dispatch               ;Finally...
           DFB    $03                    ;This is the actual format call
           DW     Cmd_ListE              ;Parameter List
           BCS    Error
;
           RTS
```

The parameter lists are as follows:

```
Cmd_ListA  DFB    $03                        ;All control calls are 3 parms long
           DFB    $01                        ;Unit #1
           DW     Ctrl_ListA                 ;This has the interleave address
           DFB    $06                        ;Set_Down_Adr control code

Ctrl_ListA DW    $02                         ;Two bytes for download address
           DW    $0062                       ;Interleave address

Cmd_ListB  DFB    $03                        ;All control calls are 3 parms long
           DFB    $01                        ;Unit #1
           DW     Ctrl_ListB                 ;This has the interleave value
           DFB    $07                        ;Download control code

Ctrl_ListB DW     $01                        ;Two bytes for download address
           DFB    $02                        ;Mac Disk Interleave value

Cmd_ListC  DFB    $03                        ;All control calls are 3 parms long
           DFB    $01                        ;Unit #1
           DW     Ctrl_ListC                 ;This has the sides byte address
           DFB    $06                        ;Set_Down_Adr control code

Ctrl_ListC DW    $02                         ;Two bytes for download address
           DW    $0062                       ;Interleave address

Cmd_ListD  DFB    $03                        ;All control calls are 3 parms long
           DFB    $01                        ;Unit #1
           DW     Ctrl_ListD                 ;This has the sides value
           DFB    $07                        ;Download control code

Ctrl_ListD DW     $01                        ;Two bytes for download address
           DFB    $00                        ;Value for single sided disk

Ctrl_ListE DFB    $01                        ;Format call has just one parameter
           DFB    $01                        ;Unit number
```

Note: You may encounter difficulties when switching 400K single-sided disks and 800K double-sided disks in the same drive. STATUS requests for the number of blocks on the disk in the drive are valid for the disk last accessed. Thus, when you READ from an 800K disk, eject it, and insert a 400K disk, a STATUS call will reveal a size of 800K until a READ or WRITE command is issued. Applications which intend to handle both 800K and 400K disks should do a READ before each STATUS call.

Further Reference

- o Apple IIGS Firmware Reference
- o Apple IIc Technical Reference Manual

END OF FILE TN.UDsk.004

```
#####################################################################
### FILE: TN.UDsk.005
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

UniDisk 3.5
#5: Architectural Differences Between 3.5" Drives

Revised by:	Matt Deatherage	November 1988
Written by:	Cameron Birse & Mike Askins	October 1986

This Technical Note provides information of interest to those developers writing low-level software for the UniDisk 3.5 and Apple 3.5 disk drives.

Definition of Drives

It is important to understand the differences between Apple's 3.5" drives if you are considering writing low-level software for use on the Apple II family drives.

UniDisk 3.5 is an intelligent drive, meaning that it has a microprocessor-based controller inside the drive enclosure that communicates with the host computer in an intelligent fashion through the IWM port. The host sends commands to the intelligent controller in the drive and the controller manipulates the drive hardware to read or write, and sends the data back to the host in a "packet" format.

Apple 3.5 Drive is an unintelligent drive that depends on the host computer to manipulate the drive hardware to read and write data to and from the drive. Apple IIGS low-level routines for this drive will be essentially the same as those downloaded to the UniDisk 3.5 controller RAM, except they will reside in the host computer's memory. New device-specific control calls must be used for the Apple 3.5 Drive.

Tips for Low-Level Drive Access

The following calls are not guaranteed to be compatible in the future; for the highest level of compatibility, avoid disk access at this level.

- Identifying the drives: The drives can be identified by first searching for a device that has the SmartPort firmware. After determining that there is a SmartPort device in the machine, perform a STATUS call with the statcode = $03 (return Device Information Block (DIB)). In the DIB there is a type byte and a subtype byte. The UniDisk 3.5 has a value of $01 for the type byte and $00 for the subtype byte. The Apple 3.5 Drive also has a value of $01 for the type byte, but its subtype byte value is $C0. Be sure to make device-specific calls to ensure drive identification. See SmartPort Technical Note #7, SmartPort Subtype Codes for more details.

- Special routines: In the UniDisk 3.5, there is extra RAM space in the controller's memory map for custom read, write and ID routines. These routines can be downloaded to the controller from the host and executed via the SmartPort. With the Apple 3.5 Drive, these special routines reside in the host memory. Equivalent mark and hook tables for the Apple 3.5 Drive, set by control calls through the SmartPort, are supported on the Apple IIGS , but are not guaranteed for all drives and CPUs.
- IWM hardware differences: On the UniDisk 3.5, the IWM registers are located in the drive's controller memory starting at $0A00. On the Apple 3.5 Drive, the IWM registers are located in host memory starting at $C0E0 (slot 6 I/O space).
- Speed differences: Downloaded code in the UniDisk 3.5 controller runs at slightly under 2 MHz, and the cycle times are regular. The Apple IIGS running at 1 MHz also has regular cycles, however, when running at 2.8 MHz, the timing is complicated by RAM refresh and I/O synchronization times. It is best to avoid timing critical solutions, or be sure to run at 1 MHz for the Apple 3.5 Drive.

As always, in order to promote compatibility between your software and future Apple II systems and to avoid writing utilities which will only work on one kind of drive, you should avoid low-level calls that are specific to a particular device or CPU.

Further Reference

- Apple IIGS Firmware Reference

END OF FILE TN.UDsk.005

F I N I S