Apple][Computer Family Technical Documentation

Technical Notes

Apple Computer -- Developer CD -- March 1993

```
######################################################################
### FILE: ATN.ABOUT.92.06
######################################################################
```

Apple II
Technical Notes

Developer Technical Support

#0: About Apple II Technical Notes

June 1992

Technical Note #0 (this document) accompanies each release of Apple II Technical Notes. This release includes new Notes for the Apple IIgs #105, 106, 107 & 108, new ProDOS Note #30, new GS/OS Note #14, revised Notes for the IIgs #12, 14, 24, 25, 51, 52, 53, 66, 67, 71, 72, 74, 75, 76, 78, 83, 91, 93, 94, 98, 99, 100, 101, 102 and 103, ProDOS #23 and 24, Miscellaneous #14 and 15, and GS/OS #1, 9, 10 and 13 as well as an index to all released Apple II Technical Notes, File Type Notes, and Sample Code. 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: DEVSUPPORT
Internet: DEVSUPPORT@AppleLink.Apple.com

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 130.43.2.3). You can also order them through Resource Central. As a Resource Central customer, you have access to the tools and documentation necessary to develop Apple II-compatible products. For more information about Resource Central, contact:

Resource Central, Inc.
P.O. Box 11250
Overland Park, KS 66207
(913) 469-6502
Fax: (913) 469-6507
AppleLink: A2.CENTRAL
Internet: A2.CENTRAL@AppleLink.Apple.com
GEnie: RC.ELLEN

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.

This Technical Note batch was originally released in May 1992. Since that time, many of the contact addresses have changed and some typographical errors have been fixed. To note these changes, this document now bears the date June

```
1992.  No content of any Notes has changed since May 1992.
Released Apple II Technical Notes                          June 1992

 New ***
 Revised *R*
Apple IIc

     1     Mouse Differences On IIe and IIc                      11/88
     2     40-Column and Double High-Resolution Graphics         11/88
     3     Foreign Language Keyboard Layouts                     11/88
     4     Dvorak Keyboard Layout                                11/88
     5     Memory Expansion on the Apple IIc                     11/88
     6     Buffering Blues                                       11/88
     7     Existing Versions                                     11/88
     8     Single-Sided 3.5" Media and the Apple IIc Plus         5/89
     9     Detecting VBL                                         11/90

Apple IIe

     1     Overview of the Apple IIe                             11/88
     2     Hardware Protocol for Doing DMA                       11/88
     3     Double High-Resolution Graphics                       11/88
     4     RDY line                                              11/88
     5     /INH line                                             11/88
     6     The Apple II Paddle Circuits                          11/88
     7     Interfaces--Serial, Parallel, and IEEE-488            11/88
     8     Known Anomalies of Enhanced IIe ROMs                  11/88
     9     Switch Input Changes                                  11/88
     10    The Apple IIe Card for the Macintosh LC               07/91

Apple IIgs

     1     How to Install Custom BRK and /NMI Handlers           11/88
     2     Transforming I/O Subroutines for Use in "Native" Mode 11/88
     3     Window Information Bar Use                             1/91
     4     Changing Graphics Modes in Mid-Application             1/91
     5     Window and Menu Titles                                11/90
     6     QuickDraw II Pattern Data Structure                    7/89
     7     Halt Mechanism in IIgs SANE                           11/88
     8     Elems Functions in IIgs SANE                          11/88
     9     IIgs Sound Expansion Connector:
           Analog Input/Output Impedances                        11/88
     10    InvalRgn Twist                                        11/88
     11    Ensoniq DOC Swap-Mode Anomaly                         11/88
 *R* 12    Tool Set Interdependencies                             5/92
     13    ROM 1.0 Modem Firmware Bug                            11/88
 *R* 14    Standard File Screwiness                               5/92
     15    InstallFont and Big Fonts                              7/89
     16    Notes on Background Printing                          11/88
     17    Application Memory Management and MMStartUp User ID   11/88
     18    Do-It-Yourself SCC Access                              7/90
     19    Multichanel Out. with the Apple IIgs Note Synthesizer 11/88
     20    Catalog of APW Language Numbers                        3/90
     21    DMA Compatibility for Expansion RAM                   11/88
     22    Proper Use of Dynamic Segments                         9/90
     23    Toolbox Use of DOC RAM                                11/88
 *R* 24    Apple IIgs Toolbox Reference Updates                   5/92
 *R* 25    Apple IIgs Firmware Reference Updates                  5/92
```

	26	ROM Revision Summary		9/89
	27	Graphics Image File Formats		11/88
	28	Interface Card Design Guidelines		11/88
	29	Monochrome High-Resolution Mode		11/88
	30	Apple IIgs Hardware Reference Updates		9/90
	31	Redirecting Output in APW C		11/88
	32	/INH Line Anomaly		11/88
	33	ERRORDEATH Macro		11/88
	34	Low-Level QuickDraw II Routines		1/91
	35	Printer Driver Specifications		9/90
	36	Port Driver Specifications		9/89
	37	Free-Form Synthesizer Tips		11/88
	38	List Controls in Dialog Boxes		9/90
	39	Mega II Video Counters		7/89
	40	VBL Signal		7/89
	41	Font Family Numbers		11/90
	42	Custom Windows		11/88
	43	Undocumented Feature of CalcMenuSize		11/88
	44	GetPenState and SetPenState Record Error		11/88
	45	Parameters for GetFrameColor		9/89
	46	DrawPicture Data Format		11/88
	47	What SetDataSize Does		11/88
	48	All About AlertWindow	obsolete	11/90
	49	Rebooting (Really)		1/89
	50	Extended Serial Interface Error Handling		1/89
R	51	How to Avoid Running Out of Memory		5/92
R	52	Loading and Special Memory		5/92
R	53	Desk Accessories and Tools		5/92
	54	MIDI Drivers		11/90
	55	Avoiding ClrHeartBeat		7/89
	56	Managing Dynamic Segments	obsolete	11/90
	57	The Memory Manager and Interrupts		12/91
	58	Keyboard Modifiers Register Anomaly		7/89
	59	Do Not Create Zero-Length Text Scraps	obsolete	1/91
	60	Menu Manager Memorabilia		11/90
	61	Window Title Handles		7/89
	62	No Non-Solid Window Background Patterns		7/89
	63	Master Color Values		7/89
	64	Apple IIgs Installer and Installer Scripts		9/89
	65	Control-^ is Harder Than It Looks		9/89
R	66	ExpressLoad Philosophy		5/92
R	67	LaserWriter Font Mapping		5/92
	68	Tips for I/O Expansion Slot Card Design		9/89
	69	The Ins and Outs of Slot Arbitration		5/90
	70	Fast Graphics Hints		9/89
R	71	DA Tips and Techniques		5/92
R	72	QuickDraw II Quirks		5/92
	73	Using User Tool Sets		1/91
R	74	A Faster List Manager Draw Routine		5/92
R	75	BeginUpdate Anomaly		5/92
R	76	Miscellaneous Resource Formats		5/92
	77	Print Manager & AppleTalk Configuration Files		1/90
R	78	Bank Alignment and Memory Management		5/92
	79	Integer Math Data Types		5/90
	80	QuickDraw II Clipping		3/90
	81	Extended Control Ecstasy		11/90
	82	Controlling the Control Manager	obsolete	11/90
R	83	Resource Manager Stuff		5/92

84 TaskMaster Madness 7/90
85 Moving the Mouse 7/90
86 Risking Resourceful Code 9/90
87 Patching the Tool Dispatcher 9/90
88 The Page One Stack in a 16-Bit World 9/90
89 MessageByName--Catchy Messages 9/90
90 65816 Tips and Pitfalls 11/90
R 91 The Wonderful World of Universal Access 5/92
92 Twisted Tales of TextEdit 11/90
R 93 Compatible Printing 5/92
R 94 Packing It In (and Out) 5/92
95 ROM Diagnostic Errors 9/90
96 Standard File Customization 11/90
97 Picture Comments and Printing 11/90
R 98 Aren't Windows A Pane 5/92
R 99 Supplemental Scrap Types 5/92
R 100 VersionVille 5/92
R 101 Patching the Toolbox 5/92
R 102 Various Vectors 5/92
R 103 Inline Procedure Names Format 5/92
104 Font Manager Fundamentals 12/91
*** 105 We Interrupt This CPU... 5/92
*** 106 ADB Addendum 5/92
*** 107 Tool Locator Tribulations 5/92
*** 108 Finder Funkiness 5/92

Apple II Miscellaneous

1 80-Column Screen Dump 11/88
2 Apple II Family Identification Routines 2.1 11/88
3 Super Serial Card Firmware Bug 11/88
4 AppleWorks Keys 5/89
5 AppleWorks File Formats 5/89
6 IWM Port Description 11/88
7 Apple II Family Identification 11/88
8 Pascal 1.1 Firmware Protocol ID Bytes 11/88
9 AppleSoft Real Variable Storage 11/88
10 80-Column GetChar Routine 9/89
11 Examining the $C800 Space from AppleSoft 5/89
12 Apple II Firmware WAIT Routine 11/88
13 not used
R 14 Guidelines for Telecommunication Programs 5/92
R 15 Compatibility Across Apple II Models 5/92
16 Apple II Parallel Interface Card Firmware 7/90
17 Buried Treasures of the Video Overlay Card 9/90

AppleTalk

1 Identifying AppleTalk 3/90
2 ProDOS 8 Compatibility on the IIe and IIgs 11/88
3 Avoiding Remote Printer Time-Outs 9/89
4 Printing Through the Firmware 9/90
5 SPCommand Calls and Error $0702 7/89
6 Apple IIe Workstation Card Anomalies 7/89
7 MLIACTV Flag and the IIe Workstation Card 11/89
8 Using the @ Prefix 9/90
9 The PAP Status Buffer 11/90

HyperCard IIGS

```
     1     Corrections to the Script Language Guide              3/91
 *R* 2     Known HyperCard Bugs                                  5/92
     3     Pitching Sampled Sounds                               3/91

GS/OS

 *R* 1     Contents of System Software Distribution Disks        5/92
     2     GS/OS and the 80-Column Firmware                     11/88
     3     Pointers on Caching                                  11/88
     4     A GS/OS State of Mind                                 7/89
     5     Resource Fork Formats                                 7/89
     6     Drivers and GS/OS Direct Page                        11/90
     7     Behavior of SET_DISKSW                                7/89
     8     Filenames With More Than CAPS and Numerals            7/89
 *R* 9     Interrupt Handling Anomalies                          5/92
 *R* 10    How Applications Find Their Files                     5/92
     11    About EraseDisk and Format                           11/90
     12    All About Notify Procs                                9/90
 *R* 13    GS/OS Reference Update                                5/92
 *** 14    The Console Driver Technical Note                     5/92

ImageWriter

     1     Custom Font Selection                                11/88

Memory Expansion Card

     1     Questions and Answers                                11/88

Mouse

     1     Interrupt Environment with the Mouse                 11/88
     2     Varying VBL Interrupt Rate                           11/88
     3     Mode Byte of the SetMouse Routine                    11/88
     4     Mouse Firmware Bug Affecting ServeMouse              11/88
     5     Check on Mouse Firmware Card                         11/90
     6     MouseText Characters                                  1/89
     7     Mouse Clamping                                       11/88
Pascal

     4     Pascal Declarations and the Directory Structure
           of a Blocked Volume                                  11/88
     10    Configuration and Use of the Apple II Pascal
           Run-Time Systems                                     11/88
     12    Disk Formatter Routine                               11/88
     14    Apple Pascal 1.3 TREESEARCH and IDSEARCH             11/88
     15    Apple II Pascal SHORTGRAPHICS Module                 11/88
     16    Driver to Have Two Volumes on One 3.5" Disk          11/88
     17    SYSTEM.APPLE Patch V2.0                               3/90

ProDOS 8

     1     The GETLN Buffer and a ProDOS Clock Card             11/88
     2     Porting DOS 3.3 Programs to ProDOS and BASIC.SYSTEM  11/88
     3     Device Search, Identification, and Driver Conventions 11/88
     4     I/O Redirection in DOS and ProDOS                    11/88
     5     ProDOS Block Device Formatting                       11/88
     6     Attaching External Commands to BASIC.SYSTEM          11/88
     7     Starting and Quitting Interpreter Conventions        11/88
     8     Dealing with /RAM                                    11/88
     9     Buffer Management Using BASIC.SYSTEM                 11/88
     10    Installing Clock Driver Routines                     11/88
     11    The ProDOS 8 MACHID Byte                             11/88
     12    Interrupt Handling                                   11/88
     13    Double High-Resolution Graphics Files                11/88
     14    Selector and Dispatcher Conventions                  11/88
     15    How ProDOS 8 Treats Slot 3                           11/88
     16    How to Format a ProDOS Disk Device                   11/88
     17    Recursive ProDOS Catalog Routine                     11/89
     18    /RAM Memory Map                                      11/88
     19    File Auxiliary Type Assignment                       11/88
     20    Mirrored Devices and SmartPort                       11/88
     21    Identifying ProDOS Devices                            3/90
     22    Don't Put Parameter Blocks on Zero Page               7/89
 *R* 23    ProDOS 8 Changes and Minutia                          5/92
 *R* 24    BASIC.SYSTEM Revisions                                5/92
     25    Non-Standard Storage Types                           12/91
     26    Polite Use of Auxiliary Memory                        1/90
     27    Hybrid Applications                                   1/90
     28    ProDOS Dates--2000 and Beyond                         9/90
     29    Clearing the Backup Needed Bit                        9/90
 *** 30    Sparse Station                                        5/92

SmartPort

     1     SmartPort Introduction                               11/88
     2     SmartPort Calls Updated                               9/89
     3     SmartPort Bus Architecture                           11/88
     4     SmartPort Device Types                               11/88
     5     SCSI SmartPort Call Changes                          11/90
     6     Apple IIgs SmartPort Errata                          11/90
     7     SmartPort Subtype Codes                              11/88
     8     SmartPort Packets                                     5/89
     9     Apple II SCSI Errata                                  7/90

UniDisk 3.5

     1     UniDisk 3.5 Internals                                11/88
     2     UniDisk 3.5 ID Bytes                                 11/88
     3     STATUS Call Bug                                      11/88
     4     Accessing Macintosh Disks                            11/88
     5     Architectural Differences Between 3.5" Drives        11/88
```

### END OF FILE ATN.ABOUT.92.06

```
#####################################################################
### FILE: ATN.INDEX.92.06
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Index June 1992

This index encompasses all Apple II Technical Notes, File Type Notes, and Sample Code. Technical Notes are denoted by a four-letter category (e.g., IIgs for Apple IIgs Notes), File Type Notes by the letters FTN, and Sample Code by the letters SC.

$C800 space	Misc 11
$E100CB	IIgs 57
/RAM volume	GSOS 2, PDOS 15, 18, 21, 23, 24, 26, 8
3.5 drive differences	UDsk 5
3.5 ROM IIc	IIc 7
5.0.x	IIgs 51
5.25 drives	Pasc 17, PDOS 21
40/80-column switch	IIc 7
40 columns	IIc 2
switching to	IIe 8
40COL (soft switch)	IIgs 29
48K run-time system (Pascal)	Pasc 15
50 Hertz	Mous 2
74LS244	IIe 2
74LS245	IIe 2
74LS251	IIe 9
80-column card	PDOS 11, 15
80-column firmware	GSOS 2, IIgs 25
80-column GETCHAR routine	Misc 10
80-column text page 2	IIe 3
80COL (soft switch)	IIc 2, IIe 8, IIe 10
80STORE (soft switch)	IIe 3, IIgs 68, IIe 10
320 mode	IIgs 4
640 mode	IIgs 4
3200-color picture	FTN $C0/0002, $C1/0002
65816 microprocessor	IIgs 90
@ prefix	ATLK 8, GSOS 10, SC 16
A/D input	IIgs 9
acceleration	IIe 10
access privileges	ATLK 8, GSOS 4
AccessPriv	SC 18
ActionNDA	SC 19
ADB	IIgs 25, 26, 30, 91, 106
AddResource	IIgs 83
address tables	IIgs 90
AddToQueue	IIgs 24
AddToRunQ	IIgs 24
AIFF	FTN $D8/0000

AIFF-C	FTN $D8/0001
AlertWindow	IIgs 48, 75, SC 5, 7
ALLOC_INT	GSOS 9
ALLOC_INTERRUPT	IIgs 18, 105
Allow Removal	SmPt 9
ALTCHARSET (softswitches)	IIe 10
ALTZP (soft switch)	IIgs 30, 68
AN3 (soft switch)	IIe 3
AN3OFF (soft switch)	IIgs 29
analog output impedence, sound	IIgs 9
animation	IIgs 70, SC 3
AnimDemo	SC 3
annunciators	IIe 10
APPEND (BASIC.SYSTEM)	PDOS 24
Apple /// emulation	Misc 2
Apple 3.5 drive	UDsk 5
Apple Bowl	GSOS 1
Apple Desktop Bus	see ADB
Apple II High-Speed SCSI Card	PDOS 23, SmPt 5
Apple IIc	IIc all
Apple IIe Card (Macintosh LC)	IIe 10, Misc 2, Misc 7
Apple IIe	IIe all
Apple IIe Workstation Card	ATLK 2, 4, 6, 7, PDOS 23
Apple IIgs	IIgs all
Apple IIgs Programmers Workshop	IIgs 20
Apple Preferred Format	FTN $C0/0002, IIgs 27
Apple Sampled Instrument Format	FTN $D5/0007
AppleDouble	FTN $E0/0002-3
AppleShare activity arrows	IIgs 5
AppleShare	FTN $B6, $C7, $E2/FFFF
volumes	ATLK 8, PDOS 17, 21, 22, 23, 30, SC 16, 18, 22
AppleSingle	FTN $E0/0001
AppleSoft	Misc 11
MouseText	Mous 6
real variable storage	Misc 9
AppleTalk	ATLK all, IIe 10, IIgs 105, FTN $E2/FFFF
drivers	FTN $BB, IIgs 18, 26, 77
AppleTalk Session Protocol	see ASP
AppleWorks GS word processor	FTN $50/8010
AppleWorks	Misc 4
data base	FTN $19
spreadsheet	FTN $1B
word processor	FTN $1A, 5
application directory	GSOS 10
application, GS/OS	FTN $B3
application, sample	SC 1
APW C	IIgs 30
APW	FTN $B0, $B5, IIgs 20, 33
arcRot	IIgs 6
ASIF	FTN $D5/0007, $D8/0002
ASP	ATLK 5
ATINIT file format	IIgs 77
ATINIT file	PDOS 23
ATLK ROM signature	ATLK 1, 2
Audio Compression and Expansion (ACE)	FTN $D8/0001
Audio IFF	FTN $D8/0000, $D8/0001

auto repeat IIgs 58
AutoLaunch SC 22
auxiliary ID IIgs 17
auxiliary memory ATLK 2, PDOS 26
auxiliary types PDOS 19
AUXMOVE routine IIe 3
Aware SC 16
background color IIgs 63
background patterns, window IIgs 62
background printing IIgs 16
backup bit PDOS 29
Bad blocks file FTN $01
BADCTL error SmPt 7
BADCTLPARM SmPt 7
bank aligned memory IIgs 78
bank-switched memory see language card
BASIC.SYSTEM PDOS 24
  buffers PDOS 9
battery RAM FTN $5A/0002
BeginUpdate IIgs 75
BELL (firmware routine) IIe 10
BELL1 (firmware routine) IIe 10
Binary II FTN $E0/8000, Misc 14
Binary Library Utility (BLU) Misc 14
BindInt GSOS 9, IIgs 18, 105
bit-encoded slot configuration IIgs 69
BLOAD (BASIC.SYSTEM) PDOS 24
block device formatting PDOS 5
block size UDsk 4
boot disk (minimal) GSOS 1
boot disks, Pascal run-time Pasc 10
booting UDsk 2
BootInit functions IIgs 73
border color IIgs 63
bottleneck procedures (QuickDraw II) IIgs 34, 97
boundsRect IIgs 80
BRK instruction IIgs 1, 105
BRKVECTOR IIgs 1
BSAVE (BASIC.SYSTEM) PDOS 24
BSAVE PDOS 24
BUBIT location PDOS 29
buffer
  QuickDraw II IIgs 72
  UnPackBytes IIgs 94
buffering
keyboard IIc 6
serial IIc 6, IIgs 25
buffering, keyboard IIgs 106
buffers, BASIC.SYSTEM PDOS 9
bug
  UniDisk 3.5 UDsk 3
  ProDOS 8 PDOS 23
  HyperCard IIgs HCGS 2
bus contention IIgs 68, Misc 3
bus, SmartPort SmPt 3
busy flag, system IIgs 57, 71
BusyBox SC 2
cache GSOS 7

cache priority	GSOS 3
caching	GSOS 3
windows	SC 7
CalcMenuSize	IIgs 43
callbacks, HyperCard IIGS	HCGS 1
cards	IIgs 68
DMA	IIe 2
physical dimensions	IIgs 28
Carta	IIgs 67
CASSIN (softswitch)	IIe 10
CASSO (softswitch)	IIe 10
CATALOG (BASIC.SYSTEM)	PDOS 24
recursive	PDOS 17
CD-ROM disks	SmPt 9
CDAs	FTN $B9, IIgs 71
CDev.Data file	FTN $C7
CDevs	FTN $C7, IIgs 100, 86, SC 15
CHAIN (BASIC.SYSTEM)	PDOS 24
ChangePath	GSOS 13
changing resolution	IIgs 4
channels, SCC	IIgs 18
character devices	GSOS 4
status	GSOS 13
character I/O	GSOS 14
character sets	IIe 10
characteristics, device	GSOS 13
characters, waiting	GSOS 13
checksum	FTN $E0/0005
chExtra	IIgs 72
clamping (mouse)	Mous 7
ClampMouse	Mous 1
Claris	FTN $19, $1A, $1B, $50/8010, Misc 4, 5
Classic Desk Accessories	see CDAs
CLD instruction	PDOS 12, 22
clearModes	IIgs 106
ClearMouse	Mous 3
ClearScreen	IIgs 72
CLI instruction	PDOS 12
clicks, multiple	IIgs 84
clipboard	IIgs 99
clipping buffer	IIgs 72
clipping	IIgs 80
clipRgn	IIgs 80
clock driver	PDOS 23
clock	PDOS 1, 11, IIe 10
Close	IIgs 53
CloseAllNDAs	IIgs 4, 53
CloseNDAByWinPtr	IIgs 71
CloseResourceFile	IIgs 71
CloseView	IIgs 91
ClrHeartBeat	IIgs 55
CLRVBLINT (soft switch)	IIgs 49
CMReleaseResource	IIgs 60
code resources	IIgs 86
attributes	IIgs 81
color table	FTN $C7
color table scrap	IIgs 99
color tables	IIgs 63, 97

window	IIgs 98
color values (RGB)	IIgs 63
command keys, NDAs	IIgs 71
Command-W and desk accessories	IIgs 71
comment, resource type	IIgs 76
CompactMem	IIgs 71
configuration files	ATLK 8
configuration	FTN $5A
Console driver	GSOS 14
CONTROL	SmPt 6
Control ^	IIgs 65
Control key	IIgs 58
Control Manager	IIgs 26, 81
Control N Monitor command	IIgs 25
Control Panel CDA	IIgs 26
Control Panel control jumper	IIgs 30
Control Panel NDA	FTN $C7
control init message	IIgs 81
control panel devices	see CDevs
control panel	IIgs 106
control record	IIgs 81
control template	IIgs 81
controls, custom	IIgs 81, 86, SC 4, 9
controls, extended	IIgs 81
controls, inactive	IIgs 24
controls, tracking	IIgs 84
convert (resource attribute)	IIgs 86
ConvSeconds	FTN $D8/0000
Cool Cursor	FTN $5A/802F
cool math (quadratic)	Misc 12
COP instruction	IIgs 105
copying files	PDOS 30
CopyPixels	IIgs 72
COUT routine	Mous 6
COUT1 routine	IIgs 25
CPU cycle	IIe 2
CREATE (ProDOS 8)	UDsk 3
CROW0 and CROW1 signals	IIgs 21
CtlNewRes	IIgs 4
current resource application	IIgs 83
cursor	
QuickDraw II	IIgs 39, 76, 85
resource format	IIgs 76
shielding	IIgs 24
text	IIgs 65
cursor manipulation	IIgs 85
cursors	FTN $5A/802F
custom controls, extended	IIgs 81
Custom.Control	SC 4
Custom.Window	SC 5
CYA	IIgs 21
cycle timings, 65816	IIgs 70
Darts	SC 17
data base, AppleWorks	FTN $19
data bus	IIe 2
dates, ProDOS	PDOS 28
Davex archived volume	FTN $E0/8004
DCE devices	IIgs 30

DEALLOC_INTERRUPT	IIgs 18, PDOS 12
debugger (custom)	IIgs 1
DebugSymbols	IIgs 103
defProc, window	IIgs 42
Deityship, Certificate of	FTN $B6
DeleteFromQueue	IIgs 24
DeleteMItem	IIgs 24
dereferencing handles	IIgs 90
Desk Manager	IIgs 26, 101
desk accessories	FTN $B8, $B9, IIgs 53, 71, SC 19
DeskShutDown	IIgs 53
desktop pattern	IIgs 98
DESTROY (ProDOS 8)	FTN $01, PDOS 23
detune (sound)	HCGS 3
DEVCNT location	PDOS 20, 21
development tool (8 bit)	SC 21
DEVICE SELECT signal	IIe 4
device names	GSOS 4
device remapping	PDOS 23
device status, character	GSOS 13
device types (SmartPort)	SmPt 4
devices	
character	GSOS 4, 13
identifying	PDOS 21
ProDOS	PDOS 20
DEVLST table	PDOS 20, 21
DEVSEL signal	IIgs 68
Dialog Manager	IIgs 26, 38, 91, SC 6
DInfo	ATLK 1, GSOS 12, IIgs 69
direct memory access	see DMA
direct page (GS/OS)	GSOS 6, 7
directory structure	GSOS 4
Disk II	PDOS 21, 23
disk image	FTN $E0/0005, $E0/800A
disk port soft switches	IIgs 30
disk sector format (3.5)	IIgs 25
DiskCopy	FTN $E0/0005
disks	
3.5, Pascal	Pasc 16
3.5, single-sided	IIc 8
Macintosh	UDsk 4
switching	UDsk 4
display screens	IIe 10
DisposeAll	IIgs 17
DisposeHandle	IIgs 17
DLog	SC 6
DMA bank register	IIgs 30
DMA	IIe 2, IIgs 21, 68
DMA register ($C037)	IIgs 21
DOC chip	IIgs 11, 9
DOC mode	IIgs 19
DOC RAM (sound)	IIgs 23, 53
DOS 3.3	PDOS 3
DOSCMD (BASIC.SYSTEM)	PDOS 2
double hi-res page 2	IIe 3
double high-resolution graphics	FTN $08/0-3FFF
	IIc 2, IIe 3, 10
packed	FTN $08/4001, IIc 2, IIe 3

DOWNLOAD UDsk 4
DragWindow IIgs 71
DrawMenuBar IIgs 60
DrawPicture IIgs 46, 72
DRename GSOS 12
drive, Apple 3.5 see Apple 3.5 drive
drivers FTN $BB
GS/OS IIgs 69, 100, GSOS 6
port IIgs 36
printer IIgs 93, see printer drivers
DTE devices IIgs 30
DTS Tools and Libraries SC 20
Dvorak keyboard IIc 4
dynamic segments IIgs 22, 56
Dynamo SC 21
DYN_SLOT_ARBITER GSOS 4, IIgs 69, 102
Easy Access IIgs 91
EasyMount FTN $E2/FFFF
editLine (dialog item) IIgs 24
Eh? IIgs 83
EJECT SmPt 2
Elems (SANE) IIgs 8
EMStatus IIgs 71
emulation mode (65816) IIgs 90
EndFrameDrawing SC 5
EndInfoDrawing IIgs 3
EndOfPicture IIgs 46
EndUpdate IIgs 75
enhanced IIe ROMs IIe 8
ENVBL (soft switch) IIc 9
EraseDisk GSOS 11
error codes, QD Aux IIgs 24
ERRORDEATH macro IIgs 33
errors
  Pascal booting Pasc 10
  ProDOS devices PDOS 21
  SmartPort IIgs 25
escape key Misc 10
Event Manager IIgs 24, 26, 71, 91
events, menu IIgs 24
ExpandPath GSOS 10
ExpressLoad IIgs 66
extended controls see controls, extended
extended file PDOS 25
extended serial port firmware IIgs 50
external commands and functions see XCMD
EZ Backup FTN $E0/8006
fakeModalDialog SC 20, 22, 9
FakeMouse IIgs 85
FamNum2ItemID IIgs 104
fastPort (QuickDraw II) IIgs 72
Fatal System Error, $0512 IIgs 24
FFGeneratorStatus IIgs 37
FFStartSound IIgs 11, 37
FFStopSound IIgs 37
File System Translators see FSTs
File Type Descriptors FTN $42
file level GSOS 13, IIgs 71

file transfer	Misc 14
file types	PDOS 19
filenames	GSOS 8
files, sparse	PDOS 30
fileSysID	GSOS 11
FIListSessions	PDOS 21
FilterProc	IIgs 38
FindControl	IIgs 84
Finder	IIgs 76, 108, FTN $42, $CA
Finder information, HFS	PDOS 25
finderSaysMItemSelected	IIgs 108
FInfo	PDOS 25
fInWindowOnly (flag bit)	FTN $C7
Firmware Reference Updates	IIgs 25
firmware	
80-column	IIgs 25
ID bytes	IIgs 25
mouse	IIgs 25
serial port	IIgs 16, 25, 26, 50
FixAppleMenu	IIgs 24, 71
Fixed data type	IIgs 79
FixFontMenu	IIgs 104
FixMenuBar	IIgs 43
floating point numerics	SC 21, see SANE
Floyd	Misc 14
Flush	GSOS 13
fNotControl (flag bit)	IIgs 92
folder selection	IIgs 96
Font Manager	IIgs 15, 26, 46, 53, 104, SC 14
font families	IIgs 41, 67
FontReport	SC 14
fonts	FTN $C8
custom	ImWr 1
large	IIgs 15, GSOS 1, IIgs 4, 41, 67, SC 14
FORMAT	SmPt 9, UDsk 4
Format (GS/OS call)	GSOS 11
formatter	
Pascal	Pasc 12
ProDOS	PDOS 5
formatting	
Macintosh disks	UDsk 4
ProDOS disks	PDOS 16
FotoFile	FTN $08/0-3FFF
FPI	IIgs 21
Frac data type	IIgs 79
fractions (cool diagrams)	IIgs 79
FrameRgn	IIgs 24
free-form synthesizer	IIgs 37
FREEBUFR routine (BASIC.SYSTEM)	PDOS 9
FreeMem	IIgs 51
FrontWindow	IIgs 91
FSTs (versions)	IIgs 100
FSTs	FTN $BD
function pointer table (FPT)	IIgs 101
FWEntry	IIgs 69
fZoomed (flag bit)	IIgs 24
game I/O	IIe 10
GETBUFR routine (BASIC.SYSTEM)	PDOS 9

GetDirEntry	GSOS 4, 13
GetFamNum	IIgs 41
GetFirstDItem	IIgs 38
GetFormatOptions	GSOS 13
GetFrameColor	IIgs 45
GetFSTInfo	ATLK 1
GetInfo	ATLK 1, 2, 4
GetInterruptState	IIgs 24
GetIntInfo	IIgs 25
GetLevel	GSOS 13, IIgs 53
GETLN (firmware routine)	IIe 10, IIgs 65
GETLN buffer	PDOS 1
GETLN1 (firmware routine)	IIe 10
GETLNZ (firmware routine)	IIe 10
GetLoadSegInfo	IIgs 66
GetMenuFlag	IIgs 24
GetMenuTitle	IIgs 60
GetMItem	IIgs 24, 60
GetMItemName	IIgs 24
GetModeBits	IIgs 18
GetNewID	IIgs 71
GetNextEvent	IIgs 71, 91
GetPenState	IIgs 44
GetPortRect	IIgs 80
GetPrefix	GSOS 10
GetPrivileges	SC 18
GetUserPath	ATLK 8, GSOS 10, FTN $5A
GetVector	IIgs 1
GetVectors	GSOS 14
GetWaitStatus	GSOS 13
GetWAP	IIgs 73
GetWTitle	IIgs 61
GET_FILE_INFO (ProDOS 8)	UDsk 3
GET_NAME, GetNameGS	FTN $B6, $B7
Golden NDA Guideline (ask Matt)	IIgs 53
GPIB (General-Purpose interface bus)	IIe 7
GR (soft switch)	IIe 3, IIgs 29
GrafOff	IIgs 72, FTN $B3, $B5
GrafOn	IIgs 72
grafPort	IIgs 35
current	IIgs 24, 82
printing	IIgs 35, 80, 91, 93
grafProcs	IIgs 34
Graphic Disk Labeler document	FTN $53/8002
graphics	IIgs 70
ground noise	IIgs 68
GS/OS application	FTN $B3
GS/OS	ATLK 8
versions	IIgs 49, 69, 93, 100, FTN $BD, GSOS all, PDOS 23, 27, SC 16, 18, 22
GS/OS aware	FTN $B3, $B5
GS/OS drivers	FTN $BB, IIgs 100
GS/OS	Misc 14
GS/OS shell application	FTN $B5
GSBug	IIgs 103
half-dot shift	IIe 3
HALT (SANE)	IIgs 7
handles, dereferencing	IIgs 90

```
hardware access                       IIe 10
hardware reference updates            IIgs 30
hardware, serial port                 IIgs 30
heartbeat tasks                       IIgs 57, FTN $B6, SC 3
HFS (Hierarchical File System)        UDsk 4
HideCursor                            IIgs 107
high-resolution graphics              FTN $08/0-3FFF
  packed                              FTN $08/4000, IIgs 29
HiliteMenu                            IIgs 24, 60
HIRES (soft switch)                   IIe 3, IIgs 29
HPIB (Hewlett-Packard interface bus)  IIe 7
hybrid applications                   PDOS 27
HyperCard IIGS
  bugs                                HCGS 2
  HyperCard IIGS Script Language Guide
                                      HCGS 1
HyperStudio sound                     FTN $D8/8001
I/O redirection (BASIC.SYSTEM)        PDOS 4
I/O SELECT signal                     IIe 4
I/O STROBE signal                     IIe 4
I/O subroutines                       IIgs 2
icon scrap                            IIgs 99
icons, Finder                         IIgs 108, FTN $CA
ID bytes                              Misc 2
  mouse                               Mous 5
  UniDisk                             UDsk 2, 7
ID nibble                             PDOS 21
identification of CPUs                Misc 2
identifying devices                   PDOS 21
iDev values                           IIgs 35, 93
idle events                           IIgs 84
IDROUTINE routine                     Misc 7
IDSEARCH routine                      Pasc 14
IEEE-488 card                         IIe 7
IFF                                   FTN $D8/0001
IIc                                   IIc all
  slots                               IIc 7
  versions                            IIc 7
IIc Plus                              IIc 7, SmPt 7
IIe                                   IIe all
IIgs                                  IIgs all
ImageWriter II                        ATLK 9
ImageWriter                           ImWr 1, SC 9
ImageWriter LQ                        ATLK 9
index                                 TN 0
information bar                       IIgs 24, 3
INH line                              IIe 5, IIgs 32
INIT                                  SmPt 2, 9, UDsk 4
initialization files                  FTN $B6, $B7
InitialLoad                           IIgs 66, 73
InitialLoad2                          IIgs 86
InitMouse                             Mous 2
inline procedure names                IIgs 103
Installer                             IIgs 64
InstallFont                           IIgs 15
InstallNDA                            IIgs 71
InstallTimer                          ATLK 5
INSTALL_DRIVER                        GSOS 6
```

instruction timing, 65816 — IIgs 2
instrument — FTN $D6
Integer Math — IIgs 26, 79, SC 13
Integrated Woz Machine — see IWM
INTEN (soft switch) — IIgs 49
interchange formats — FTN $D8/0000
interfaces — IIe 7
interlace mode — Misc 17
interleave — UDsk 4
international country codes — IIgs 76
interrupt state record — IIgs 24
interrupt status, SmartPort — SmPt 2
interrupt, non-maskable — IIgs 1
interrupts — IIgs 57, ATLK 2
  SCC — IIgs 18, 71, 91, 105, Misc 14
  serial — IIc 6
  unclaimed — IIe 8, IIgs 25, 70, GSOS 9, Mous 1, 4, PDOS 12
IntSource — IIgs 71
InvalRect — IIgs 24, 75
InvalRgn — IIgs 10, 75
Invisible bit — ATLK 6
IOSEL signal — IIgs 68
IOSTRB signal — IIgs 68
IOUDIS (softswitch) — IIe 10
IRQ signal — IIgs 68
IRQ. vectors — IIgs 105
item draw routine (List Manager) — IIgs 74
ItemID2FamNum — IIgs 104
IWM — IIgs 30, Misc 6, UDsk 5
J (middle initial) — IIgs 91
Jiffy.Windows — SC 7
joystick connector — IIe 10
joystick — IIe 6
jumper, Control Panel (S1) — IIgs 30
keyboard buffering — IIc 6, IIgs 106
keyboard — IIgs 106
keyboard layout — IIe 10
keyboard modifiers register — IIgs 58
keyboard mouse — remove
keyboard, Dvorak — IIc 4
keyboard, foreign — IIc 3
keyboard, reading — IIgs 91
  from CDAs — IIgs 71
KEYMODREG (soft switch) — IIgs 58
Krunching — Pasc 10
KVERSION location — PDOS 23
language card — PDOS 12
LaserWriter — ATLK 9, IIgs 41, 67, SC 9
LaserWriter font mapping — IIgs 67
LCBNK2 (soft switch) — IIgs 30
LEFromScrap — IIgs 59
LEIdle — IIgs 84
LEKey — IIgs 84
level — see file level
level_mode — GSOS 13
LGetPathname2 — IIgs 71
Line Edit controls — IIgs 81

Line Edit	IIgs 26, 81, SC 8
List Manager	IIgs 24, 26, 38, 74, SC 17, 8
list controls	IIgs 38
list members, inactive	IIgs 24
List.Line.Edit	SC 8
Lister	SC 9
lists in dialogs	IIgs 38
load files	FTN $BC
loaders	IIgs 22, 52, 66
LoadOneTool	IIgs 100, 53
LoadResource	IIgs 83
LoadSegName	IIgs 22
LoadTools	IIgs 100
locInfo	IIgs 6, 80
longStatText2 (dialog item type)	IIgs 91
lowercase	GSOS 8
M2B0 signal	IIgs 68
M2SEL signal	IIgs 68
MACHID location	PDOS 11, 15
Macintosh Audio Compression and Expansion (MACE)	FTN $D8/0001
Macintosh disks	UDsk 4
Manners, Missed	PDOS 26
MarkResourceChanged	IIgs 83
mask scrap	IIgs 99
Math	SC 13
MaxBlock	IIgs 51
maxWidth	IIgs 72
Media Control drivers (versions)	IIgs 100
Media Control	IIgs 12, GSOS 1
Medley document	FTN $54/DD3E
Mega II	IIgs 32, 68
Mega II video counters	IIgs 39
memFlag (List Manager)	IIgs 74
Memory Expansion Card	IIe 10, MemX 1
Memory Manager	IIgs 17, 26, 57, 107
Memory Peeker desk accessory	IIgs 25
memory compaction	IIgs 57
memory expansion	IIc 5, IIgs 21
memory expansion slot	IIgs 21
memory ID	see User ID
memory management	GSOS 3
and interrupts	IIgs 4, 17, 22, 51, 52, 57, 78, 91, PDOS 26, 27
memory purging	IIgs 57
Menu Manager	IIgs 26, 43, 60, 81, SC 12
menu bars, system vs. window	IIgs 60
menu caching	IIgs 24
menu events	IIgs 24
menu strings	IIgs 60
menu titles	IIgs 5
menu, custom	SC 12
MenuGlobal	IIgs 60
MenuNewRes	IIgs 4
Menus	SC 12
MenuSelect	IIgs 60
MenuStartUp	IIgs 60
MessageByName	IIgs 89

MessageCenter	FTN $B3, $B5, IIgs 89, 98
MFS (Macintosh File System)	UDsk 4
MIDI data	FTN $D7
MIDI drivers	IIgs 54
MIDI	IIgs 105
MIDI Synth	IIgs 12, 54, GSOS 1
MIDI Tools drivers	FTN $BB
MIDI tools	IIgs 23, 54
Miscellaneous Tools	IIgs 26, 94
MIXED (soft switch)	IIe 3
MLIACTV location	ATLK 7, PDOS 23
MMStartUp	IIgs 17, 53, PDOS 27
modal dialog	IIgs 91, SC 6
modal windows in NDAs	IIgs 71
ModalDialog	IIgs 38
mode switching (320/640)	IIgs 4
mode, mouse	Mous 3
modeless dialog	SC 6
modifiers	IIgs 58, 106
MoMan	SC 22
Money, It's Only	Misc 13
Monitor	IIgs 25, 26
monochrome hi-res graphics	IIgs 29
Moof!*	SC 10
moreFlags	IIgs 81
motherboard	IIe 9
Mouse Keys	IIgs 91
mouse button status	IIgs 25
mouse card	IIc 1, Mous 5
mouse firmware	IIgs 25
mouse	IIe 10, mous all
behavior	IIc 1
clamping	Mous 7
identification	Mous 5
positioning	IIgs 85
scaling	IIc 1
mouse mode	Mous 3
MouseText	Mous 6
mouseUp events	IIgs 71
MOVE_INFO	IIgs 102, GSOS 6
MPW IIgs	IIgs 33
MSLOT location	IIgs 16, Misc 3, 15, PDOS 23
MTR (BASIC.SYSTEM)	PDOS 24
multimedia drivers	FTN $BB
Music Construction Set	FTN $D5, $D6
music sequence	FTN $D5
name (macro)	IIgs 103
named resources	HCGS 2
names, inline procedure	IIgs 103
NDAs	FTN $B8, IIgs 71, SC 19
network volumes	ATLK 8
New Desk Accessories	see NDAs
NewControl2	IIgs 81
NewHandle	IIgs 17
NewMenu	IIgs 60
NewMenuBar2	IIgs 60
NEWVIDEO (soft switch)	IIgs 70
NewWindow	IIgs 24, 47

NewWindow2	IIgs 24, 3, 82, SC 2, 9
NextMember2	IIgs 24
Nifty, Mr.	FTN $B9
NIL in resource maps	IIgs 83
NMI	IIgs 1
NMI signal	IIgs 68
no-special-memory attribute	IIgs 52
no-wait mode and console driver	GSOS 14
no-wait mode	GSOS 13
non-maskable interrupt	IIgs 1
Note Sequencer	IIgs 23
Note Synthesizer	IIgs 19, 23
notification procedures	FTN $B6, GSOS 12, 7
NOTRACE	PDOS 24
NTSC color	IIe 3
NuFX	FTN $E0/8002, Misc 14
Object Module Format	FTN $BC, IIgs 66
offscreen ports	SC 20
OMF	FTN $BC, IIgs 66
OMF KIND field	IIgs 52, 78
onlyGetSelection	IIgs 92
ON_LINE (ProDOS 8)	IIc 5, PDOS 8, 21, 23
OOM Queue routines	IIgs 51
Open	SC 16
OpenNDA	IIgs 71
OpenPicture	IIgs 72
OpenPoly	IIgs 72
OpenPort	IIgs 91
OpenRgn	IIgs 72
optionalCloseAction (NDA Action code)	IIgs 71
option_list	FTN $BD, GSOS 4, 13
OSShutDown	GSOS 2, IIgs 49
OS_BOOT location	PDOS 27
out-of-memory queue, corrupted	IIgs 24
out-of-memory routine	IIgs 51, 78
P-Machine	Pasc 10
PackBytes format	IIgs 27
PackBytes	FTN $08/0-3FFF, $08/4000, $08/4001, $C0/0001, $C0/0002, IIgs 94
packed super-hires image	FTN $C0/0001
packets, SmartPort	SmPt 8
paddles	IIe 6
PAGE2 (soft switch)	IIe 3, IIgs 68
PaintWorks packed picture	FTN $C0/0000
PAL timing chip	IIe 2
PAP	ATLK 3
PAPOpen	ATLK 9
PAPStatus	ATLK 9
Parallel Interface Card	IIe 7, Misc 16
parameter blocks, controls	IIgs 81
parameter blocks, ProDOS	PDOS 22
ParamText	IIgs 91
Pascal 1.1 firmware protocol	IIgs 16, Misc 8
Pascal and MouseText	Mous 6
Pascal area (on ProDOS volume)	PDOS 25
Pascal	Pasc all
48K run-time system	Pasc 15
Patch	Pasc 17

run-time boot disks — Pasc 10
Pascal Profile manager — PDOS 25
Pascal run-time system — Pasc 10
Pascal volumes — Pasc 16
Patch (Pascal) — Pasc 17
patching, tool dispatcher — IIgs 87
patching, tools — IIgs 101
pathnames — GSOS 4
pattern, desktop — IIgs 98
patterns — FTN $C0/0002
  window backgrounds — IIgs 62
  list — IIgs 76
PDLTRIG (softswitch) — IIe 10
PEEK (Applesoft function) — Misc 11
pen pattern — IIgs 6
pen state record — IIgs 44
permanent initialization files — FTN $B6, $B7
PFI — PDOS 21
PH0 signal — IIe 4, IIgs 68
PH1 signal — IIe 4
PH2 signal — IIe 4, IIgs 21, 68
PicComments — IIgs 97, SC 9
PICT — FTN $C1/0001, IIgs 27, 46
picture (QuickDraw II) — IIgs 72, FTN $C1/0001
  Macintosh — IIgs 46
picture comments — IIgs 97
picture data format — IIgs 46
Picture, QuickDraw II — IIgs 27
PINIT entry — IIgs 16
pinouts — IIe 7
PinRect — IIgs 24
pitch (sound) — HCGS 3
Pixel Map Tools — SC 13, 20
pixelMap2Rgn — SC 20
PlaySound — SC 10
PMCloseSession — ATLK 3
PMLoadDriver — IIgs 77, SC 11
PMSetPrinter — ATLK 3
PMUnloadDriver — IIgs 77
polygons — IIgs 72
Pop-Up Menu controls — IIgs 81
port drivers — IIgs 36
ports, offscreen — SC 20
portSCB — IIgs 80
PosMouse — IIgs 85, Mous 1, 3
PostScript — IIgs 67, 97
power consumption, cards — IIgs 68
power-up byte — MemX 1
POWERUP location — IIgs 49
PPToPort — IIgs 80
PREAD (firmware routine) — IIe 6, 10
preferences — FTN $5A
prefixes and Standard File — IIgs 14
prefixes — IIgs 14, GSOS 10
preload (resource attribute) — IIgs 86
Prevent Removal — SmPt 9
PrGetUserName — IIgs 77
Print Manager drivers — FTN $BB

Print Manager	IIgs 35, 77, 93, SC 11, 9
print loop	IIgs 93
print record	IIgs 35, 93, SC 11
resource	IIgs 76
Printer Access Protocol	see PAP
printer driver	IIgs 35, 93, 97, SC 11
printer drivers (versions)	IIgs 100
Printer.Setup file	IIgs 77
printing	IIgs 35
background	IIgs 16
transparent	ATLK 4, 97
printing text	IIgs 93
printX subrecord	IIgs 93
procedure names, inline	IIgs 103
procname (macro)	IIgs 103
ProDOS 8	
	ATLK 2, PDOS 4, FTN $01, GSOS 2, 8, SC 21
bugs	PDOS 23
ProDOS file system limitations	GSOS 4
ProDOS Filing Interface	see PFI
ProDOS FSM (Macintosh)	PDOS 25
ProDOS	PDOS all
PrRecord.Spy	SC 11
PrValidate	IIgs 93
ptrCheck	SC 9
PTRIG (soft switch)	IIc 9
ptrToPixImage	IIgs 80, 91
ptrToPixImage	IIgs 91
purgeable memory	IIgs 51
purging memory	IIgs 78
PWRITE entry	IIgs 16
Q Monitor command	IIgs 25
Q6 and Q7 (IWM)	IIgs 30
QDStartUp	IIgs 72
QDStartUp	IIgs 72
QDStatus	IIgs 72
QDVersion	IIgs 72
QuickDraw II Aux, error codes	IIgs 24
QuickDraw II Auxiliary	IIgs 24, 46, 53, 75, 104
QuickDraw II	IIgs 4, 10, 26, 34, 44, 6, 72, 80, 91
tutorial	IIgs 80
QuickDraw II picture	FTN $C1/0001
QUIT code	PDOS 23, 23
R/W line	IIe 4
R/W* line	IIe 2
RAM Disk	GSOS 2, PDOS 8
RAMRD (soft switch)	IIe 3, IIgs 68, Misc 15
RAMWRT (soft switch)	IIe 3, IIgs 68, Misc 15
rComment	IIgs 76
rCursor resources	FTN $5A/802F
RDALTZP (soft switch)	IIgs 25
RDCARDRAM (softswitch)	Misc 15
RDDHIRES (softswitch)	IIe 10
RDIOUDIS (soft switch)	IIe 10, IIc 9
RDY line	IIe 4
RDY signal	IIgs 68
READ (BASIC.SYSTEM)	PDOS 24

READ (firmware routine)	IIe 10
ReadAsciiTime	IIgs 24
ReadMouse	IIc 1, IIgs 85, Mous 1, 3
RealFreeMem	IIgs 51
rebooting	IIgs 49
recharge routine	IIgs 16
rectangle list resource format	IIgs 76
REDIRECT, shell call	IIgs 30
refCon, list control	IIgs 38
regions	IIgs 10, 72
registers, SCC	IIgs 18
relPitch	HCGS 3
remapping devices	PDOS 23
Remote Print Manager	ATLK 3, 4, IIgs 77, 93
RemoveFromRunQ	IIgs 24
Replicator	FTN $5A/8031
Replicator	FTN $E0/800A
requests, system	FTN $C7, IIgs 108
resChanged (resource attribute)	IIgs 83
RESET routine	IIgs 49
Reset (system)	IIgs 91
reset key	IIe 5
ResetHook	IIgs 25, SmPt 6
ResetMember2	IIgs 24
resolution, changing	IIgs 4
Resource File	FTN $5E/0001, PDOS 25
Resource Manager	IIgs 53, 76, 83, 86, 107
resource application	IIgs 3, 83
resource application, current	IIgs 83
resource converters	IIgs 83, 86
resource files	GSOS 13
resource forks	GSOS 5, 13, IIgs 71
resource formats	IIgs 76
resource reference scrap	IIgs 99
resource search depth	IIgs 83
resource search order	IIgs 83
resource search path, CDevs	FTN $C7
resource, sound	FTN $D8/0003
resourceBlocks, resourceEOF	GSOS 13
resources	SC 2
as templates	IIgs 83
resources, named	HCGS 2
ResourceStartUp	IIgs 53, 71
Restart	IIgs 52, 73
ReturnStat (HyperCard IIGS parameter)	HCGS 1
Revision B motherboard	IIe 3
rFont resources	IIgs 76
rListRef resources	IIgs 74
RMFindNamedResource	IIgs 83
RMLoadNamedResource	IIgs 83
ROM Disk	IIgs 25
ROM revisions	IIgs 26
ROMs, IIe enhanced	IIe 8
rowBytes	IIgs 80
RS-232-C	IIgs 30
RS-422	IIgs 30
RSHIMEM routine (BASIC.SYSTEM)	PDOS 9
rSoundSample	HCGS 3, IIgs 76, FTN $D8/0003

RST signal	IIgs 68
rStyleBlock	IIgs 99, FTN $50/5445
rTextForLETextBox2 (resource type)	IIgs 24
RTI instruction	PDOS 12
rTwoRects resources	IIgs 24
run queue	IIgs 24, 57, FTN $B6
run-time system, Pascal	Pasc 10
rVersion	IIgs 76
sampled sound	FTN $D8
sampling frequency	IIgs 37
SANE	FTN $1B, IIgs 26, 7, 8, PDOS 26, SC 13
sapling files	PDOS 30
scan line control bytes	IIgs 97
scanline counter	IIgs 39
SCC (chip)	IIgs 18
SCC AREG soft switch	IIgs 49
SchAddTask	IIgs 71
scheduler	IIgs 16, 26, 71
Scrap Manager	IIgs 26, 53
scrap types	IIgs 99
screen dump	IIgs 27
80 columns	Misc 1
screen holes	IIe 10
screen image	FTN $C1/0000
packed	FTN $C0/0001, IIgs 27
scripts, Installer	IIgs 64
ScrollRect	IIgs 72
SCSI	SmPt 5, 9
seedling files	PDOS 30
segments	IIgs 66
dynamic	IIgs 22, 86
selection (Text Edit)	IIgs 92
selector/dispatcher (ProDOS 8)	see QUIT code
self test	IIe 9, IIgs 95
SendInfo	IIgs 106
SendQueue	IIgs 16
separators	GSOS 4
SerFlag	IIgs 18
serial buffering	IIc 6, IIgs 25
serial communication	IIgs 105
serial connection	Misc 14
serial controller chip	IIgs 18
serial interrupts	IIc 6
serial port firmware	IIgs 16, 25, 26, 50
serial port hardware	IIgs 30
serial ports	IIe 10
ServeMouse	Mous 1, 4
SetArcRot	IIgs 6
SetAutoKeyLimit	IIgs 24
SetBufDims	IIgs 72
SetContentOrigin	IIgs 47
SetCtlMoreFlags	IIgs 81
SetCtlTitle	IIgs 81
SetCurResourceFile	IIgs 83
SetDataSize	IIgs 47
SetDefaultTPT	IIgs 101
SetDTR example	IIgs 50
SetFrameColor	IIgs 98

SetGrafProcs	IIgs 34, 35
SetInfoDraw	IIgs 3
SETINTC3ROM (soft switch)	GSOS 2
SetInterleave	IIgs 25
SetInterruptState	IIgs 24
SetIntInfo	IIgs 25
SetLevel	GSOS 13, IIgs 53
SetMenuFlag	IIgs 24, 60
SetModeBits	IIgs 25
setModes	IIgs 106
SetMouse	IIc 1, Mous 3, 4
SetMTitleStart	IIgs 5
SetOrigin	IIgs 80
SetOutBuff	IIgs 16
SetPenState	IIgs 44
SetPort	IIgs 24
SetPrefix	GSOS 10
SetPrivileges	SC 18
SetResLoad	IIgs 83
SetResourceAttr	IIgs 83
SetResourceFileDepth	IIgs 83
SETSLOTC3ROM (soft switch)	GSOS 2
SetStdProcs	IIgs 34, 35
SetSysBar	IIgs 60
settings, Sound control panel	FTN $5A/Sound.Settings
SetTSPtr	IIgs 24, 73
SetVector	IIgs 1
SetWAP	IIgs 73
SetWTitle	IIgs 61
SetZoomRect	IIgs 24
SET_DISKSW	GSOS 12, 7
SET_DOWN_ADDR	UDsk 4
SET_EOF	PDOS 30
SET_FILE_INFO	PDOS 29
SET_MARK	PDOS 30
SET_SYS_SPEED (vector)	GSOS 6, 102
SFMultiGet2	IIgs 14, SC 18
SFPGetFile2	IIgs 96
SFPMultiGet2	IIgs 14
SHADOW (soft switch)	IIgs 70
shadowing, super-hires	IIgs 72
sheet feeder	ATLK 9
Shell	SC 1
shell application, GS/OS	FTN $B5
shell programs and resources	IIgs 83
ShieldCursor	IIgs 34
Shift key	IIgs 58
shift key modification	IIe 9, 10
Sholes keyboard	IIc 4
Shortcuts (file)	GSOS 1
SHORTGRAPHICS unit	Pasc 15
ShowBootInfo	FTN $B6, $B7
ShowPen	IIgs 72
ShrinkIt	Misc 14
ShutDownTools	IIgs 107
SIGNAL	GSOS 6
signals, GS/OS	IIgs 57
SizeWindow	IIgs 4

```
slot 3                                  PDOS 15
slot arbitration                        GSOS 4, 6, IIgs 69
slot mapping (ProDOS devices)           PDOS 3
slot ROM space                          ATLK 4
SLOTC3ROM (soft switch)                 GSOS 2
SLTROMSEL soft switch                   IIgs 69
Small Computer Systems Interface        see SCSI
SmartPort errors                        IIgs 25
SmartPort                               IIe 10, IIgs 25, 26, PDOS 20, 21, 23,
SmPt all
SmartPort version                       IIgs 100
soft switches                           IIe 10, Misc 15
  disk port                             IIgs 30
Sonata                                  IIgs 67
Sound settings                          FTN $5A/Sound.Settings
Sound Tools                             IIgs 26, 37, SC 10, 23
sound
  multi-channel                         IIgs 19
  resource format                       IIgs 76
  sampled                               FTN $D8
sound clicks                            IIgs 11
sound connector                         IIgs 9
sound RAM                               see DOC RAM
sound resource                          FTN $D8/0003
sound scrap                             IIgs 99
sound, compressed                       FTN $D8/0001
sound, pitching                         HCGS 3
SoundEx                                 SC 23
SoundSmith                              FTN $D5/0007
source code file                        FTN $B0
sparse files                            PDOS 30
SPCommand                               ATLK 5
special memory                          IIgs 52, 53
spExtra                                 IIgs 72
SPGetStatus                             ATLK 1
spreadsheet, AppleWorks                 FTN $1B
SPWrite                                 ATLK 5
stack pointer                           ATLK 2, IIgs 25, 70, 88
stack space, CDAs                       IIgs 71
stack, page one                         IIgs 71, 88
Standard Apple Numerics Environment     see SANE
Standard File                           IIgs 14, 96, SC 18
StartFrameDrawing                       SC 5
StartInfoDrawing                        IIgs 3
startup order                           IIgs 12
StartUpTools                            IIgs 100, 12, 24, 53, 83, 107
STATUS bug in UniDisk 3.5               UDsk 3
STATUS                                  SmPt 7
  SmartPort                             PDOS 20, SmPt 2, 9, UDsk 2, 4
  status                                GSOS 13
status string                           ATLK 9
status, Pascal protocol                 ATLK 6
statusBits                              ATLK 9
stereo sound                            IIgs 19
Sticky Keys                             IIgs 91
storage types                           PDOS 25
STORE (BASIC.SYSTEM)                    PDOS 24
strings, tagged                         IIgs 76
```

STROBE' (signal)	IIe 10
styled fonts	IIgs 104
StyleWriter	IIgs 93
subtype, SmartPort	SmPt 7
Super Serial Card	IIe 7, Misc 3
super hi-res graphics	Misc 17
super hi-res screen	FTN $B3, $B5
super hi-res screen location	IIgs 91
super-hires screen image	FTN $C1/0000
SuperInfo modules	FTN $BC/4002
SW0 (soft switch)	IIe 9
SW1 (soft switch)	IIe 9
SW2 (soft switch)	IIe 9
swap mode (sound)	IIgs 11
synthLAB	GSOS 1
SysBeep2	FTN $5A/Sound.Settings
SysFailMgr	IIgs 33
system Monitor	IIgs 25
system software	GSOS 1
SYSTEM.APPLE	Pasc 17
SystemEvent	IIgs 24
systemOrUser (flag)	IIgs 101
SystemTask	IIgs 53, 91
tables of addresses	IIgs 90
tagged strings	IIgs 76
Tangent, Mr.	IIgs 71
TaskMaster	IIgs 42
task mask	IIgs 84, 47, 53, 75, 80, 81, 84, 91, SC 2, 9
TaskMasterDA	IIgs 84
Teach document	FTN $50/5445
Teach	SC 24
TEClick	IIgs 92
TEFormat	FTN $50/5445
TEFormat structure	IIgs 99
TEGetRuler	IIgs 92
TEGetText	IIgs 92
TEIdle	IIgs 84
TEInsert	IIgs 92
TEKey	IIgs 84
telecommunications	Misc 14
TENew	IIgs 92
TEPaintText	IIgs 92
TERuler	IIgs 92
TESetRuler	IIgs 92
TESetText	IIgs 92
TestControl	IIgs 92
Text Tools	GSOS 4, IIgs 26, 69, 71, 93
text buffer, QuickDraw II	IIgs 4
text color	IIgs 63
text cursor bug	IIgs 65
text printing	IIgs 93
TextEdit controls	IIgs 81
TextEdit	FTN $50/5445, IIgs 92, SC 24
TextEdit style scrap	IIgs 99
TimeData	Mous 2
timing	
65816 cycle	IIgs 70

  65816 instruction   IIgs 2
timing   IIgs 105
titles, window and menu   IIgs 5
TLMountVolume   IIgs 24
TLShutDown   FTN $B6, $B7, IIgs 53, 73, 101
TLStartUp   FTN $B6, $B7, IIgs 53
TLTextMountVolume   IIgs 24
TOBRAMSETUP   IIgs 25, 102
ToBusyStrip, vector   IIgs 73
Tool Locator   IIgs 26, 89, 101, 107
tool dispatcher, patching   IIgs 87
tool patching   IIgs 101
tool set interdependencies   IIgs 12
tool set pointer table (TPT)   IIgs 101
tool sets   FTN $BA
  user   IIgs 73, SC 20
tool sets versions, system   IIgs 100
Toolbox Reference updates   IIgs 24
toolbox
  and desk accessories   IIgs 53
  memory usage   IIgs 51
TOPRINTMSG8   IIgs 25
ToStrip, vector   IIgs 73
TOWRITEBRAM   IIgs 25
TrackControl   IIgs 84
TrackGoAway   IIgs 42
transparent network printing   ATLK 4
tree files   PDOS 30
TREESEARCH routine   Pasc 14
trust and verification   FTN $B8
TURTLEGRAPHICS unit   Pasc 15
type-ahead buffer   IIc 6
typeless files   FTN $00
UCSD Pascal   Pasc all
UnBindInt   GSOS 9
UNBIND_INT_VECTOR   GSOS 9
UniDisk 3.5 drive   PDOS 23
UniDisk 3.5   UDsk all
UniDiskStat   IIgs 25
UnionRgn   IIgs 24
UniqueResourceID   IIgs 83
UnitStatus (Pascal)   Pasc 17
UnitStatus   IIc 5
unit_number   PDOS 20, 21
Universal Access   IIgs 91
unknown files   FTN $00
UnloadOneTool   IIgs 53
UnloadSeg   IIgs 22
UnloadSegNum   IIgs 22
unlocked memory   IIgs 57
UnPackBytes   FTN $08/4000, $08/4001,
                                      $C0/0001, $C0/0002, IIgs 94
unpacked picture   IIgs 27
UnshieldCursor   IIgs 34
up, shut (you people!)   Misc 13
update routines, dialogs   IIgs 91
updateRgn   IIgs 75
updating windows   SC 7

User ID	IIgs 66, 71, PDOS 27
user input routine	GSOS 14
user names	IIgs 77
user tool sets	IIgs 53, 73, SC 20
UserCtlItem	IIgs 38
UserInfo	ATLK 8, SC 18
UserItem	IIgs 38
userItems in dialogs	IIgs 91
VBL (softswitch)	IIe 10
VBL interrupts	IIgs 39, 40, Mous 2
rate	Mous 2
VBL signal	IIc 9, IIgs 40
VCB	PDOS 23, 8
vectors	IIgs 105, Misc 14
vectors, various	IIgs 102
vendor ID, SmartPort	SmPt 2
version	IIgs 76
version number, SmartPort	SmPt 2
version, resource type	IIgs 76
versions	IIgs 100
vertical blanking	see VBL
Video Keyboard	IIgs 91
Video Overlay Card	Misc 17
video counters	IIgs 70
Visit Monitor desk accessory	IIgs 25
visRgn	IIgs 75, 80, 91
Volume Control Block	see VCB
volume control	IIc 7
volumes (Pascal)	Pasc 16
WAIT (firmware routine)	IIe 10
WAIT routine	Misc 12
wait mode	GSOS 13
wFrame	IIgs 3, 42
WindNewRes	IIgs 4
Window Manager	IIgs 26, 47, 61, 62, 75
window background patterns	IIgs 62
window defProc	IIgs 42
window information bar	IIgs 24, 3
window record	IIgs 24, 42
window titles	IIgs 5, 61
windows, color tables	IIgs 98
windows, custom	IIgs 42, SC 5
windows, updating	IIgs 83, SC 7
WindStatus	IIgs 75
wInfoDefProc	IIgs 3
wInfoHeight	IIgs 3
wInSpecial (TaskMaster result)	IIgs 24
word processor	
AppleWorks	FTN $1A
AppleWorks GS	FTN $50/8010
Medley	FTN $54/DD3E
Text Edit	FTN $50/5445
WordPerfect	FTN $A0/0000
WordPerfect file	FTN $A0/0000
work area pointer	IIgs 73
WRCARDRAM (soft switch)	IIgs 70, Misc 15
WRITE (firmware routine)	IIe 10
WRITE (SmartPort)	UDsk 4

write-protect bug (UniDisk 3.5)   UDsk 3
XCMD/XFCN (HyperCard IIGS)   IIgs 86, HCGS 1
xFInfo   PDOS 25
XorRgn   IIgs 24
years, ProDOS   PDOS 28
Z8530 serial chip   IIgs 18
zero page   PDOS 22
  SmartPort use of   SmPt 6
zero-crossing byte   IIgs 1

### END OF FILE ATN.INDEX.92.06

```
#####################################################################
### FILE: ATN.Thanks
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Text File Thanks and Credit

Developer Technical Support (DTS) has always written Apple II Technical Notes and File Type Notes to be read on paper, where typographical conventions, formatting and diagrams can often make points much, much clearer than simple ASCII text can. However, for internal reasons of author convenience this makes the source material in a format unusable on the Apple II, and the lowest-common denominator for conversion to a native format is ASCII text files.

Converting these files to text is hard, tedious and thankless, and DTS wishes to thank those people who have spent their own time and effort to provide this service for the Apple II development community.

For June/July 1992:

Developer Technical Support wishes to thank Eileen Crawford and Tammy Dimas in Apple's Engineering Support group for working to make the tools for text file conversion of these Notes available quickly.

We also thank Softdisk Publishing for the preliminary use of a program that makes converting Apple IIgs/Macintosh high-ASCII characters to standard ASCII a lot more painless than it used to be.

But the biggest thanks go to Steve Gunn of New Castle, Indiana, who's worked long and hard (and given up part of his summer) to make these Notes available before the big conference in Kansas City. Now Steve's agreed to reconvert the existing Technical Notes and File Type Notes so they're all consistent and accurate. Thanks, Steve!

Apple II Technical Note and File Type Note Text File Hall of Fame

Mark B. Johnson
Jim Luther
Matt Deatherage
Eric Mueller
Dave Ely
Tim Swihart
Steve Gunn

### END OF FILE ATN.Thanks

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

Apple II
Technical Notes

Developer Technical Support

AppleTalk
#1:   Identifying AppleTalk

Revised by: Jim Luther   March 1990
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 July 1989: Added warning concerning ProDOS 8, version 1.4.

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
```

Warning: Due to a bug in ProDOS 8, version 1.4, using the $42 call crashes ProDOS 8 if AppleTalk is not installed. Applications that use this routine to check for AppleTalk should ship with ProDOS 8 version 1.5 or greater, thus avoiding this bug. (ProDOS 8 Technical Note #21, Identifying ProDOS Devices contains a routine which correctly identifies the presence AppleTalk under all versions of ProDOS 8.)

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

- o Inside AppleTalk
- o AppleShare Programmer's Guide for the Apple IIGS
- o GS/OS Reference
- o Apple II AppleTalk Technical Note #4, Printing Through the Firmware
- o ProDOS 8 Technical Note #21, Identifying ProDOS Devices

### 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 1990
Written by: Matt Deatherage & Jim Luther — July 1989

This Technical Note discusses considerations of printing through the AppleTalk Remote Print Manager (RPM) interface from ProDOS 8 applications.
Changes since March 1990: Revised code sample to simplify finding the transparent network printing slot with the ROM 03 Apple IIgs. Please note that the method of finding the transparent network printing slot shown in the March 1990 revision of this Note does work correctly, the new method is just simpler. In addition, revised the wording of the Note to clarify that transparent network printing is the RPM interface.

The AppleShare Programmer's Guide to the Apple IIgs stated that the Remote Print Manager (RPM) interface allowed transparent network printing through Super Serial Card entry points in slot 7. This statement is pretty short-sighted. It's much like saying printing to 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.

Note: The AppleShare Programmer's Guide to the Apple IIgs has been superseded by the AppleShare Programmer's Guide to the Apple II Family.

An Apple IIe Workstation Card, although recommended for slot 7, can work in almost any slot (just like an ImageWriter II with a Super Serial Card 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 the firmware used by the RPM interface in slot 7. An Apple IIgs with ROM version 03 may only have the firmware used by the RPM interface in either slot 1 or 2.

Before printing through the RPM interface slot, take the same precautions you would take before printing to any slot--check to make sure you see the requested slot is 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.

ProDOS 8 applications which offer network printing through the RPM interface should give users the choice of printing to any of the seven slots as well as the Network Printer. When Network Printer is selected, the application can find the slot used by the RPM interface by using the 6502 code sample included in this Note. Allowing the selection of Network Printer is especially important for applications that keep a configuration file containing a user's default printer setup. If an application keeps only the slot number in the configuration file, users may need to change the printer selection often if they print from several different machines.

Warning: Printing to a slot with no firmware generally results in a crash.

The code sample uses two methods to determine the slot the RPM interface is using. The first method works with the Apple IIe Workstation card and the ROM 01 Apple IIgs. It looks at the AppleTalk completion routine address returned by the AppleTalk GetInfo call, and if that address is in the slot ROM space, then that slot is the slot used by the RPM interface. 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 when printing through the RPM interface. If the completion routine address is not in the slot ROM space, then the application cannot determine what slot the RPM interface is using and must query the user. The second method works only with the ROM 03 Apple IIgs. It retrieves the slot the RPM interface is using from location $E101C2.

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 (.RPM) driver, which can be identified by a deviceID of $001F as returned from the DInfo call.

```
;
; This routine will identify AppleTalk and the RPM interface slot
; (if possible).
; This routine is for ProDOS 8 applications only.
;
                    keep FindRPMSlot
                    longa off
                    longi off

FindRPMSlot         start
                    lda #$00
                    sta RPMSlot         default to no RPM interface slot

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

                    jsr $BF00           ProDOS 8 MLI
                    dc  h'42'           $42 command for network calls
                    dc  a'InfoParams'   Parameter list address
                    bcs NoATalk         no AppleTalk; handle the error

; Get machine type & ROM version (see Apple II Miscellaneous Tech Note #7)

                    sec
                    jsr $FE1F           What kind of machine are we on?
                    bcs CheckCom        Not a IIGS, check completion address
                    cpy #$03
                    bcc CheckCom        Earlier than ROM 03 IIGS, check
;                                         completion address

ROM03               anop                ROM 03 or greater IIGS' use location
;                                       $E101C2 to find the RPM interface slot
                    lda $E101C2         Get the RPM interface slot
                    sta RPMSlot

                    beq AskForSlot
                    bra HaveSlot
```

```
CheckCom            anop                  use completion address to find slot
                    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
                    bcs AskForSlot        greater or equal to $C8 is bad
                    cmp #$C1
                    bcc AskForSlot        less than $C1 is bad
                    and #$0F              $Cn = $0n
                    sta RPMSlot

HaveSlot            anop                  AppleTalk is installed and RPM is
;                                           using slot #RPMSlot

AskForSlot          anop                  AppleTalk is installed but RPM
 ;                                          interface slot cannot be determined

NoATalk             anop                  AppleTalk is not installed

                    rts                   so this sample returns

RPMSlot             entry
                    dc  h'00'             Slot RPM interface is using

InfoParams          dc  h'00'             Synchronous only
                    dc  h'02'             GetInfo call number
                    ds  2                 result code
ComReturn           ds  4                 completion return address
                    ds  8                 space for other result info

                    end
```

Further Reference

- AppleShare Programmer's Guide for the Apple II Family
- Apple II AppleTalk Technical Note #1, Identifying AppleTalk
- Apple II Miscellaneous Technical Note #7, Apple II Family Identification
- 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.

- o 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.

- o 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

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

### END OF FILE TN.ATLK.006

```
#####################################################################
### FILE: TN.ATLK.007
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

AppleTalk
#7: MLIACTV Flag and the IIe Workstation Card

Written by: Mark Day & Dan Strnad November 1989

This Technical Note describes a problem using the MLIACTV flag with the IIe Workstation Card.

When using the Apple IIe Workstation Card, the MLIACTV flag does not always show that the MLI (or PFI) is active. This inconsistency can cause programs that use the MLIACTV flag to fail when making MLI calls from interrupt routines. Programs can correct for this problem by making all MLI calls through the NewMLI routine listed in this Note and checking the NewMLIActv flag instead of the MLIACTV flag. This approach solves the problem only if all MLI calls, including those made by any interrupt routines, are made through this routine.

The following routine is a replacement for the MLI entry point at $BF00. Programs using this routine can perform a JSR to NewMLI instead, which fixes the problem. Section 6.2.1 of the ProDOS 8 Technical Reference Manual details how programs can cause the MLI to return the their routine rather than the routine that originally called it. For programs using this technique that are also using the routine below, the location below labeled NewCmdAddr replaces CmdAdr ($BF9C). The steps involved in patching the MLI return location still apply, as specified in Section 6.2.1 of the ProDOS 8 Technical Reference.

```
; MLI patch for Apple II Workstation Card
; by Mark Day
;
; code shown is compatible with MPW IIGS cross-assembler
;
; Your program should use the NewMLIActv flag instead of
; MLIACTV ($BF9B), and should JSR NewMLI instead of
; JSR MLI ($BF00).
;

           machine M6502            ; 6502 code for //e
           longa  off
           longi  off

parmptr    equ    0                 ; two bytes on zero page
MLI        equ    $BF00             ; entry to the real MLI

NewMLI     proc

           php                      ; save old interrupt status to
           pla                      ; temporarily disable interrupts
           sta    oldp              ; so that NewCmdAddr is always valid
           sei                      ; when an interrupting routine sees
                                    ; NewMLI active.

           sec
           ror    NewMLIActv        ; NewMLI is now active!

;
; We need to get the return address from the stack so we can
; get the command number and parameter block address which
; follow the JSR NewMLI, and so we can save NewCmdAddr.
;
           clc
           pla                      ; get low byte of parm address - 1
           sta    parmptr
           adc    #4                ; get real return address
           sta    NewCmdAddr
           pla
           sta    parmptr+1         ; save high byte of parm address - 1
           adc    #0
           sta    NewCmdAddr+1      ; save real return address

           lda    oldp
           pha
           plp                      ; reinstate old interrupt status
;
; Now, we copy the call number and parameter list pointer that followed
; the JSR NewMLI, and copy them after a JSR to the real MLI.
;
           tya                      ; save Y on stack
           pha
           ldy    #1                ; offset to command number
           lda    (parmptr),y       ; get command number
           sta    NewCmdNum
           iny                      ; point to parm list ptr (low)
           lda    (parmptr),y
           sta    NewParmPtr
           iny
           lda    (parmptr),y
           sta    NewParmPtr+1
           pla                      ; unstack value of y register
           tay

;
; Now, call the real MLI with the user's command and parameter list
; and jump back to our caller.
;
           jsr    MLI               ; call the real MLI
NewCmdNum  dc.b   0                 ; command number
NewParmPtr dc.w   0                 ; parameter list pointer

           php                      ; save C because LSR changes it!
           lsr    NewMLIActv        ; MLI is no longer active
           plp                      ; restore C
           dc.b   $4C               ; JMP absolute instruction
NewCmdAddr dc.w   0                 ; target of jump, caller's return address
```

```
NewMLIActv dc.b   0                 ; $80 bit set if MLI active
oldp       ds.b   1                 ; used to preserve processor status

           endp
           end
```

Note that this routine also works on the Apple IIGS, even though the problem with the MLIACTV flag only affects Apple IIe Workstation Cards.

Further Reference

- AppleShare Programmer's Guide for the Apple IIGS
- ProDOS 8 Technical Reference Manual

### END OF FILE TN.ATLK.007

```
########################################################################
### FILE: TN.ATLK.008
########################################################################
```

Apple II
Technical Notes

Developer Technical Support

AppleTalk #8: Using The @ Prefix

Written by: Jim Luther & Dan Strnad

September 1990

This Technical Note discusses use the @ prefix with multiple users.

Apple II computers on AppleShare networks feature a unique folder for each user on the server volume, called the user's folder. Server volumes containing these user's folders are called user volumes. User's folders exist on user volumes so that the system and applications have a standard place to store user-specific data on the network. All network volumes an Apple II can boot from are user volumes.

Under GS/OS, the @ prefix allows applications to automatically work with the user's folder. If a user launches your application from a server volume with a user volume mounted, GS/OS sets the @ prefix to the user's folder; otherwise it sets it to the application folder. The @ prefix can reduce design and coding effort for multilaunch features by providing the application with the system's best guess at where user-specific information should be stored. To safely use the user's folder feature, programmers need only remember to use the @ prefix with the GS/OS class 1 Open call (requestAccess = 1, 2, or 3). Using the @ prefix with the class 1 Open provides safe access to the file for as long as it remains open, without requiring any network-specific code.

Using the @ prefix is appropriate for applications that want to avoid network-specific programming while being reasonably well-behaved in a network environment. For example, applications may store printer defaults in the @ directory or use it as a default when prompting the user to choose a directory.

There are situations writing data to a file in the @ directory could result in other users overwriting the data; however, applications may reasonably require that users not allow these situations to occur   In Table 1, the third through fifth cases are all situations in which this problem could occur. For best results, when opening a file for writing with the @ prefix, use access privileges that deny write access to other users. The GS/OS class one Open call always does this when requestAccess is non-zero. Without this precaution of denying write access to other users, the user's data is not protected from being overwritten while it is in use.

Application launched from...	Is a user volume present?	User name	@ prefix set to...	Is this case detectable?
local	maybe	any name	application prefix	yes

net	yes	(not guest)	user folder	yes
net	no	any name	application prefix	yes
net	yes	guest	guest folder	yes
net	yes	same as another user	user folder	special programming required

Table 1-Possible @ Prefix Configurations

Consider the third case. Although the application was launched from a server, the server does not contain a user's folder, nor is there any other mounted server containing a user's folder. In this case, if multiple users launch a single copy of the application from the same folder at the same time, each user's @ prefix would point to the same folder from which they all had launched the application. By denying other users write access when opening the file, you prevent users from overwriting each other's data. However, the other users are no longer prevented from overwriting the data when the user with write access closes the file, at which point a different user can write to the file. Therefore, using access privileges in combination with the @ prefix deters one user's data from being overwritten by another, but only so long as the file remains opened by the user with write access. This approach may provide sufficient protection for saving certain user configuration and preference information.

When saving work the user plans to resume later, this approach may not provide sufficient protection. In this situation, conflicts can also arise if the @ prefix is set to the application prefix rather than to the user's folder. It is up to the programmer to determine whether to use the @ prefix, how to use it, and whether this level of protection is sufficient for the particular data involved.

In addition to using the @ prefix (or the user path to which it attempts to refer) with access that denies other users permission to write to the file, applications can check to see if the user path could point to the same folder for multiple users at the same time. To do this, the application first checks to see if it was launched across the network. This is the case when class one GetFileInfo on the user path returns fileSysID = $0D. If the application was launched across the network, the user path could be set the same for multiple users if the user has logged on as a guest (UserInfo returns a null userName, the fourth case in Table 1) or if you are using the @ prefix and the system has set it to the application prefix on a non-user network volume (error $60 from GetUserPath, the third case in Table 1). If the application determines that the user prefix may be set the same for multiple users, then it could use an alternate approach to determine where the data is to be stored, by prompting the user for example.

Although it would be comparatively difficult for an application to determine whether multiple users with the same name were running the application from the same server (the fifth case in Table 1), the documentation for the application could warn against this. The system does not provide any specific information about when this condition occurs.

One More Caution

One other caution to observe when using the @ prefix: since other applications are also storing data in the same user's folder, each application should be careful to reference distinct files. Regardless of how the application chooses to do this--by checking that the file being created does not already exist, by choosing a distinct name for the file, or by some other method--it should usually operate only on files of its own creation.

Programmers should keep in mind that the @ prefix is provided as a programming convenience. The AppleShare FST also provides the GetUserPath and UserInfo calls. In combination with GetFileInfo, these calls allow programmers to devise other, more customized approaches for determining where to save the user's data.

Further Reference

- AppleShare Programmer's Guide for the Apple II Family
- GS/OS Reference
- GS/OS Technical Note #10, How Applications Find Their Files

### END OF FILE TN.ATLK.008

```
#####################################################################
### FILE: TN.ATLK.009
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

AppleTalk #9: The PAP Status Buffer

Written by: Jim Luther   November 1990

This Technical Note shows the format of the status data returned into the application-supplied status buffer by the PAPStatus and PAPOpen Printer Access Protocol (PAP) AppleTalk commands. The status buffer format is shown for both LaserWriter and ImageWriter (with the ImageWriter II/LQ LocalTalk Option card installed) printers.

The PAPStatus and PAPOpen AppleTalk commands must supply a pointer to a 260-byte status buffer. When the PAPStatus or PAPOpen commands complete, the status buffer contains the ATP data portion of a Status (TResp) packet. The first four bytes of that data are unused, so the actual status data starts at offset $04 in the status buffer.

The LaserWriter printer returns its status data in the form of a Pascal string. That string is usually something suitable to display on the screen (e.g., "status: idle" or "job: Fred; document: My LaserWriter is on fire; status: busy; source: AppleTalk"). In fact, the status text displayed in the Print Manager LaserWriter dialog boxes is usually the statusString returned by PAPStatus or PAPOpen. Figure 1 shows the contents of the status buffer returned by a LaserWriter.

```
          +-------------------+
     $00  |_                 _|    Longword       Unused
          |_      unused     _|
          |_                 _|
          |                   |
          +-------------------+
     $04  |_ (string length) _|
          |_                 _|
          |_                 _|
          .                   .
          .   status string   .    String         The PAP status string
          .                   .                   (Pascal string, ASCII,
          |_                 _|                   high-bit clear)
          |_                 _|
          |                   |
     $103 +-------------------+
```

Figure 1-PAP Status Buffer from a LaserWriter

The ImageWriter II/LQ LocalTalk Option card does not return a status string for display. Instead, it returns a statusBits word where each bit within that word has a specific meaning. Your application can interpret the statusBits word and generate an appropriate message to display. Figure 2 shows the contents of the status buffer returned by the ImageWriter II/LQ LocalTalk Option card and the individual bit definitions of the statusBits word.

```
          +------------------+
     $00  |_                _|     Longword       Unused
          |_     unused     _|
          |_                _|
          |                  |
          +------------------+
     $04  |string data length|     Byte           Always = 2
          +------------------+
     $05  |_   statusBits   _|     Word           Status bits returned by
          |                  |                    LocalTalk ImageWriter
          +------------------+                    Option card (see following
     $07  |_                _|                    definition)
          |_                _|
          |_                _|
          .                  .     253 Bytes      Unused
          .                  .
          .                  .
          |_                _|
          |_                _|
          |                  |
     $103 +------------------+

                            +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                            |15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
                            +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                              |  |<-- Reserved --->|  |  |  |  |  |  |  |  |
1 = Printer is busy ----------+  |_________________|  |  |  |  |  |  |  |  |
                                                      |  |  |  |  |  |  |  |
1 = Color ribbon installed ---------------------------+  |  |  |  |  |  |  |
1 = Sheet feeder installed ------------------------------+  |  |  |  |  |  |
1 = Paper out error ----------------------------------------+  |  |  |  |  |
1 = Cover open error ------------------------------------------+  |  |  |  |
1 = Printer off line ---------------------------------------------+  |  |  |
1 = Paper jam error -------------------------------------------------+  |  |
1 = Printer fault ------------------------------------------------------+  |
1 = Printer active (head is moving) ---------------------------------------+
```

Figure 2-PAP Status Buffer from an ImageWriter II/LQ LocalTalk Option Card

There are two additional things to note when interpreting the statusBits word returned by a ImageWriter II/LQ LocalTalk Option card:

- If a sheet feeder is installed (bit 6 = 1), running out of paper results in a "Paper jam error" (bit 2 = 1) instead of a "Paper out error" (bit 5).
- The ImageWriter II/LQ LocalTalk Option card has been known to randomly return all ones in the low byte (bits 0-7) of the statusBits word. When this happens, the statusBits word is invalid and an application should repeat the PAPStatus call to get valid information.

Further Reference

- o Inside AppleTalk, Second Edition
- o AppleShare Programmer's Guide for the Apple II Family

### END OF FILE TN.ATLK.009

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

Apple II
Technical Notes

Developer Technical Support

GS/OS
#1: Contents of System Software Distribution Disks

Revised by: Matt Deatherage June 1992
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 1991: Now describes System Software 6.0. Changed the title to not reflect disk names.

This Note gives a description of each of the files in the Apple IIgs System Software 6.0 package. This package includes six disks: Install, SystemTools1, SystemTools2, Fonts, synthLAB and System.Disk. System Software 6.0 requires at least 1 MB of memory, one 3.5" drive and another storage device (either a second 3.5" drive or a larger capacity device). 2 MB of memory and a hard disk are highly recommended.

System.Disk is a pre-configured boot disk for floppy-based users. Because all the files on System.Disk appear on other disks in the 6.0 set, they are only listed and not described a second time.

Contents of Install

ProDOS	Every file system boots differently; the boot blocks for ProDOS disks look for a file name ProDOS. This is that file. It is the GS/OS file system stub necessary to start the boot process.
System	The directory containing most of the GS/OS files.
CDevs	The directory containing all Apple IIgs Control Panel Devices (CDevs) required for installing 6.0.
General	Allows setting of general system parameters.
RAM	Controls the size of the RAM disk and the GS/OS Disk Cache.
SetStart	Lets you choose which application to boot into.
Desk.Accs	The directory containing all the classic and new desk accessory files to be loaded at boot time.
ControlPanel	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. Also drives SuperDrives connected to the Apple II SuperDrive interface card.
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. In 6.0, it is up to 300% faster than in earlier versions of system software.
Console.Driver	The text screen and keyboard device driver for GS/OS.
SCSI.Manager	The GS/OS SCSI Manager, the supervisory driver that arbitrates hardware-level usage of Apple's Apple II SCSI cards.
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.
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.
Fonts	The directory containing all system fonts to be used.
FastFont	This makes Shaston 8 text drawing much faster.
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.
SetStart.data	An invisible file created by the SetStart Control Panel, indicating which application the system should boot into. On this disk, this points to the Installer.
Start	The boot program. If this file exists, GS/OS always launches it upon booting. Under 6.0, this program usually reads the SetStart.data file and launches the indicated application.
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.
Resource.Mgr	The Resource Manager. This is an initialization file; the design of the Resource

	Manager requires it to be present even when an application has not specifically loaded it. The system does not boot if this file is not present.
Sys.Resources	A file containing system resources, available to the system software and to applications.
Tool.Setup	A required file that loads files which contain all the patches to tools in ROM for ROM levels 01 (TS2) and 03 (TS3). 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. Tool.Setup also contains patches common to both ROM 1 and ROM 3.
TS2	Patches to ROM tools for ROM 1.
TS3	Patches to ROM tools for ROM 3.
Tools	The directory containing tool files for all tools not in ROM.
Tool014	Window Manager .
Tool015	Menu Manager.
Tool016	Control Manager.
Tool018	QuickDraw Auxiliary.
Tool019	Print Manager.
Tool020	LineEdit.
Tool021	Dialog Manager.
Tool022	Scrap Manager.
Tool023	Standard File.
Tool027	Font Manager.
Tool028	List Manager.
Tool034	TextEdit.
Icons	The directory containing all the Finder's old-style icon files as well as new Desktop database files and file type descriptors.
FType.Apple	The file type names used by the Finder (on all systems).
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.
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. It also reads MessageCenter message #1.
A2.RAMCard	Script to install the driver for the Apple II Memory Expansion Card (the slot-based, or "slinky" card).
Adv.Disk.Util	Script to install the Advanced Disk Utility program.
Apple.Bowl	Script to install the Apple Bowl game.
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.

AppleShare3.5	Script that creates an 800K or 1440K GS/OS startup disk which contains AppleShare.
Archiver	Script to install Archiver, the new GS/OS-based backup program.
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.
Calculator	Script to install the Calculator new desk accessory.
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.
CloseView	Script to install the CloseView NDA, which makes the screen more legible to some visually-impaired users.
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.
DOS3.3.FST	Script to install the read-only DOS 3.3 file system translator.
Easy.Access	Script to install the EasyAccess init, which provides sticky keys and keyboard mouse to ROM 1 users.
Epson	Script to install the Epson printer driver for the Print Manager, as well as the parallel card driver.
Fonts	Script to install the minimum suggested font set.
Fonts.Max	Script to install all fonts provided with System 6.0.
Fonts.Std	Script to install the standard font set.
HFS.FST	Script to install the Hierarchical File System (HFS, used on the Macintosh) file system translator.
Inst.Sys.Min	Script to install a minimal GS/OS system on an 800K volume. Note that this is different than 5.0.x's "Inst.Sys.Min" script, the 6.0 version of which is in the file named "AppleShare3.5".
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 complete System Software 6.0 configuration, including new features, 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

	minimal system software that boots into a server selection program (the network "Start" program from SystemTools2).
MediaControl	Script to install the Media Control toolset and all Media Control drivers supplied with System 6.0.
MediaCtrl.CDSC	Script to install the Media Control toolset and the drivers to work with the Apple CD SC drive.
MediaCtrl.P2000	Script to install the Media Control toolset and the drivers to work with the Pioneer 2000 series laserdisc players.
MediaCtrl.P4000	Script to install the Media Control toolset and the drivers to work with the Pioneer 4000 series laserdisc players.
Namer	Script to install the printer Namer Control Panel. Namer II (a ProDOS 8 application) is not included with System 6.0.
Pascal.FST	Script to install the read-only Apple II Pascal file system translator.
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.
SCSI.Scanner	Script to install the SCSI Manager and SCSI scanner driver for GS/OS.
SCSI.Tape	Script to install the SCSI Manager and SCSI tape driver for GS/OS.
Server.Sys.File	Script to install System Software 6.0 on an AppleShare File Server.
Sounds.All	Script to install all sounds provided with System Software 6.0 into the "System:Sounds" folder of the designated volume.
StyleWriter	Script to install the StyleWriter printer driver for the Print Manager, as well as the files necessary to connect it to a serial port.
Teach	Script to install the application Teach, which displays and edits Teach files, text files, AppleWorks files, MacWrite files and Installer scripts.
UniDisk3.5	cript to install the UniDisk 3.5 driver for GS/OS.
VideoKeyboard	Script to install the Video Keyboard new desk accessory, which allows users to type by using the pointing device instead of the keyboard.
VideoMix	Script to install the latest versions of the Apple II VideoMix software and tools.

Contents of SystemTools1

Icons	Additional icons for the Finder. This folder is currently empty.
System	A directory containing additional parts of the system software.
Finder	The Apple IIgs Finder, version 6.0.
CDevs	Directory with additional Control Panel Devices.
DirectConnect	Allows selection of direct-connected printers.

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.
Printer	Controls printer port settings.
Slots	Allows selection of slot settings and startup slot.
Sound	Sets user preference for sound pitch and volume. Also allows the user to assign digitized sounds to events that happen while using the computer.
Time	Sets the internal clock's time and display format and optionally tracks Daylight Savings Time.
Desk.Accs	Directory with additional desk accessories.
CDRemote	An updated version of the CD Remote new desk accessory which ships with the AppleCD SC.
FindFile	A new desk accessory that finds files on volumes GS/OS can read.
Calculator	A calculator new desk accessory.
Drivers	Directory with additional device drivers for GS/OS and the Toolbox.
A2.RAMCard	The GS/OS driver for slot-based memory expansion cards. This driver is not required to use these cards with GS/OS, but it does provide a substantial speed improvement.
Apple.MIDI	The Apple MIDI Interface driver for the MIDI Tools.
Card6850.MIDI	The driver for 6850-based MIDI interface cards for the MIDI Tools.
Epson	The Epson(R) printer driver for the Print Manager.
ImageWriter	The ImageWriter driver for the Print Manager.
ImageWriter.LQ	The ImageWriter LQ driver for the Print Manager. Starting with System Software 5.0.3, this driver uses all the capabilities of the ImageWriter LQ.
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.
SCSI.Manager	The GS/OS SCSI Manager, the supervisory driver that arbitrates hardware-level usage of Apple's 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.
SCSIScan.Driver	The GS/OS driver for the Apple Scanner or OneScanner. This driver is required for GS/OS to recognize Apple's scanners.
SCSITape.Driver	The GS/OS driver for the Apple Tape Backup 40SC. This driver is required for GS/OS to recognize Apple's now-discontinued Tape Backup 40 SC.
StyleWriter	The StyleWriter driver for the Print Manager.

```
     Fonts                    Directory with additional fonts
          Courier.09          9-point Courier font.
          Courier.10          10-point Courier font.
          Courier.12          12-point Courier font.
          Courier.14          14-point Courier font.
          Courier.18          18-point Courier font.
          Courier.20          20-point Courier font.
          Courier.24          24-point Courier font.
          Geneva.10           10-point Geneva font.
          Geneva.12           12-point Geneva font.
          Geneva.14           14-point Geneva font.
          Geneva.16           16-point Geneva font.
          Geneva.18           18-point Geneva font.
          Geneva.20           20-point Geneva font.
          Geneva.24           24-point Geneva font.
          Helvetica.9         9-point Helvetica font.
          Helvetica.10        10-point Helvetica font.
          Helvetica.12        12-point Helvetica font.
          Helvetica.14        14-point Helvetica font.
          Helvetica.18        18-point Helvetica font.
          Helvetica.20        20-point Helvetica font.
          Helvetica.24        24-point Helvetica font.
          Shaston.16          16-point Shaston font.
          Times.09            9-point Times font.
          Times.10            10-point Times font.
          Times.12            12-point Times font.
          Times.14            14-point Times font.
          Times.18            18-point Times font.
          Times.20            20-point Times font.
          Times.24            24-point Times font.
          Venice.12           12-point Venice font.
          Venice.14           14-point Venice font.
          Venice.24           24-point Venice font.
     FSTs                     Directory with additional File System
                              Translators.
          DOS.3.3.FST         The DOS 3.3 FST, which allows GS/OS to access
                              5.25" disks formatted in DOS 3.3 format.  This
                              FST is read-only; it only performs read
                              operations.
          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.
          HFS.FST             The HFS FST, which allows GS/OS to read and
                              write any disk in the Macintosh's HFS format.
          Pascal.FST          The Apple II Pascal FST, which allows GS/OS to
                              access any disk formatted in Apple II Pascal
                              format.  This FST is read-only; it only
                              performs read operations.
     Tools                    Directory with additional tools.
          Tool025             Note Synthesizer.
          Tool026             Note Sequencer.
          Tool029             ACE Tools.
          Tool032             MIDI Tools.
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.
BASIC.System	The ProDOS 8 BASIC command interpreter.

Contents of SystemTools2

Icons	Additional icons for the Finder. This folder is currently empty.
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	An update to the Aristotle program installed by the "Aristotle.Patch" script.
QuickLogoff	An initialization file used to add a quick logoff feature to AppleShare.
Start	The AppleShare startup program which is installed instead of the standard Start program on AppleShare volumes. It allows the user to log on and then launches the server startup program for the user's machine.
System	A directory containing additional parts of the system software.
CDevs	Directory with additional Control Panel Devices.
AppleShare	Allows users to choose and log onto AppleShare file servers.
FolderPriv	Allows users to set default folder privileges on AppleShare file server volumes.
MediaControl	Allows users to set up the Media Control tool set and the drivers they wish to use.
Namer	Allows users to rename AppleTalk-based ImageWriter, ImageWriter LQ and LaserWriter printers.
NetPrinter	Allows users to choose AppleTalk-based ImageWriter, ImageWriter LQ and LaserWriter printers.
Desk.Accs	Directory with additional desk accessories.
MediaControl	A new desk accessory that's like a "super" remote control for all devices the Media Control toolset can control.
VideoKeyboard	A new desk accessory that allows users to type with the pointing device instead of with the keyboard.
VideoMix	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.
AppleTalk	The AppleTalk port driver for the Print Manager. It works with either serial port when configured for AppleTalk.
ATalk	The main AppleTalk GS/OS driver.
ATP1.ATROM	AppleTalk protocols to patch the IIgs ROM.

ATP2.ATRAM	AppleTalk protocols not in ROM.
IWEM	PostScript(R) program which allows a LaserWriter emulate an ImageWriter. A user can load it into the LaserWriter with the LaserWriter Control Panel, and it is automatically invoked when printing through the slot associated with AppleTalk.
LaserWriter	The LaserWriter driver for the Print Manager. This driver works with any LaserWriter with PostScript. It does not work with the LaserWriter IIsc or Personal LaserWriter LS. This driver doesn't always print color patterns correctly to PostScript Level 2 printers, such as the LaserWriter IIf, LaserWriter IIg or Personal LaserWriter NTR.
Media.Control	Drivers for the Media Control toolset
AppleCDSC	Media Control driver for the Apple CD SC drive.
Pioneer2000	Media Control driver for the Pioneer 2000 series of laserdisc players.
Pioneer4000	Media Control driver for the Pioneer 4000 series of laserdisc players.
SCC.Manager	The GS/OS supervisory driver that arbitrates hardware-level usage of the serial communications controller in the Apple IIgs.
Fonts	Directory with additional fonts. Currently, this directory on this disk is empty.
FSTs	Directory with additional file system translators.
AppleShare.FST	The AppleShare FST which allows GS/OS to access AppleShare file servers.
Sounds	A folder with sounds provided for the new Sound Control Panel. The file names are fairly self-explanatory; the sounds are not described here.
Ahh	
Doorbell	
Droplet	
Eastern	
Frog	
PipeOrgan	
Quack	
SimpleBeep	
Sosumi	
Swish	
Trumpets	
Whoosh	
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.
CloseView	A new desk accessory (installed by an init) that magnifies the screen to make it more visible to some users with visual impairments.
EasyAccess	An initialization file that brings Sticky Keys

```
                              and Keyboard Mouse to ROM 1 users.
          EasyMount           An initialization file that creates file server
                              aliases in the Finder.
     Tools                    Directory with additional tools.
          Tool033             VideoMix toolset (for the Video Overlay Card).
          Tool038             Media Control toolset.
Archiver                      A GS/OS based backup and restore program.
Teach                         A simple editor that uses TextEdit to display
                              and edit text files, Teach files, Installer
                              scripts and AppleWorks and MacWrite documents.
Read.Me                       Last-minute news and information about the
                              System Software.  Read with Teach.
Shortcuts                     A Teach file with time-saving system tips and
                              information.
```

Contents of Fonts

```
Goodies                       A directory with files that are only related to
                              system software in the vaguest sense.
     Apple.Bowl               A GS/OS conversion of an old Apple II bowling
                              game.
     Read.Me                  Documentation on Apple Bowl.
Icons                         Additional icons for the Finder.
     AppleBowl.Icon           The icon for the Apple Bowl game.
System                        A directory containing additional parts of the
                              system software.
     Fonts                    Additional fonts.
          Courier.27          27-point Courier font.
          Courier.28          28-point Courier font.
          Courier.30          30-point Courier font.
          Courier.36          36-point Courier font.
          Courier.42          42-point Courier font.
          Helvetica.27        27-point Helvetica font.
          Helvetica.28        28-point Helvetica font.
          Helvetica.30        30-point Helvetica font.
          Helvetica.36        36-point Helvetica font.
          Helvetica.42        42-point Helvetica font.
          Helvetica.48        48-point Helvetica font.
          Helvetica.60        60-point Helvetica font.
          Helvetica.72        72-point Helvetica font.
          Helvetica.96        96-point Helvetica font.
          Times.27            27-point Times font.
          Times.28            28-point Times font.
          Times.30            30-point Times font.
          Times.36            36-point Times font.
          Times.42            42-point Times font.
          Times.48            48-point Times font.
          Times.60            60-point Times font.
          Times.72            72-point Times font.
          Times.96            96-point Times font.
```

Contents of synthLAB

```
synthLAB                      The synthLAB application, a demonstration
                              sequencer for the MIDI Synth toolset.
Tool035                       MIDI Synth toolset.
```

MIDI	The MIDI Control Panel. Lets you choose a MIDI driver.
Seq.and.Instr	A directory containing demonstration sequences (files that end in ".seq"), wave forms (files that end in ".wav") and sound banks (files that end in ".bnk") for use with synthLAB and MIDI Synth. The files are only listed; their sound is not described here.
Synth.bnk	
Synth.seq	
Synth.wav	
Bee.seq	
Capri.seq	
Combo.bnk	
Combo.wav	
Demo.bnk	
Demo.wav	
Fugue.seq	
Midsummer.seq	
Orch.bnk	
Orch.wav	
Piano.bnk	
Piano.wav	
Rhythm.seq	
Sonata.seq	
Reference	A Teach document with the electronic manual for synthLAB.

Contents of System.Disk

Files are only listed here; they are described earlier in this Note where they first appeared.

```
ProDOS
System
     Start.GS.OS
     GS.OS
     Error.Msg
     GS.OS.Dev
     FSTs
          Pro.FST
          Char.FST
     Drivers
          AppleDisk3.5
          AppleDisk5.25
          Console.Driver
     System.Setup
          Tool.Setup
          TS2
          TS3
          Resource.Mgr
          Sys.Resources
     Desk.Accs
          ControlPanel
     CDevs
          Printer
     Time
     Start                    This is the Finder, not the SetStart program
                              or the AppleShare program.
     Tools
          Tool014
          Tool015
          Tool016
          Tool018
          Tool019
          Tool020
          Tool021
          Tool022
          Tool023
          Tool025
          Tool027
          Tool028
          Tool034
     Fonts
     P8
Icons
     Ftype.Apple
BASIC.System
```

Minimum GS/OS System Disk Requirements

The following files are required for GS/OS to boot from a local disk. 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
          *HS.FST             Required for High Sierra or ISO 9660 discs.
          Char.FST
          *AppleShare.FST     Required to use AppleShare file servers
          *DOS3.3.FST         Required to use DOS 3.3 disks
          *Pascal.FST         Required to use Apple II Pascal disks
          *HFS.FST            Required to use HFS disks
     Drivers
          *AppleDisk3.5       Required for Apple 3.5 Drives or SuperDrives.
          *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.
          *SCSIScan.Driver    Required for Apple scanners.
          *SCSITape.Driver    Required for Apple Tape backup.
          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
          Tool.Setup
          TS2
          TS3
          Resource.Mgr
          Sys.Resources
     CDevs
          *AppleShare         Required for selecting AppleShare file servers.
          *NetPrinter         Required for choosing printers.
          *DirectConnect      Required for choosing printers.
          *General
          *RAM                Should always be included if space allows.
                              Provides the only way to set the size of the
                              GS/OS Disk Cache.
     Desk.Accs                Required for desk accessories; any desk
                              accessories should be installed in this
                              directory.
          *ControlPanel       Required if you ship any Control Panels (CDevs).
     *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.
```

Further Reference

- GS/OS Reference
- Apple IIgs Technical Note #100, VersionVille

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: Matt Deatherage March 1991
Written by: Matt Deatherage January 1989

This Technical Note discusses GS/OS concepts and practices.

Changes singe July 1989: Includes more information about thinking for non-ProDOS file systems.

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, suchas volume names, be sure to use the colon as the separator, because if you donot, filenames with slashes will cause problems. Similarly, don't assume any ofthefollowing:

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

Your application may have hidden file system assumptions as well. For example, while a directory behaves like a directory under all GS/OS filesystem translators, reading from a directory is not always as fast as it isfor ProDOS disks. ProDOS directories are fairly linear and can be searched quickly; but other file systems may have more complicated directory structures (HFS and AppleShare, for example, have B-trees that store directory entries in alphabetical order). To get optimal speed, try to do as many GetDirEntry calls as you can in succession without other GS/OS calls intervening this allows Apple to optimize file system translators for fast directory reading.

Also remember that other file systems may not support the concept oforderable directories, so don't depend on directory order in your application.

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 unableto 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 todo 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.

However, if you want to change something in an existing file's GetFileInfo list, do not use an option_list. The option_list could override the other parameters to SetFileInfo without your knowledge.

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:	Matt Deatherage	January 1991
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 September 1990: Updated the list of calls which do not require the GS/OS direct page and updated the documentation references.

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, DYN_SLOT_ARBITER, MOVE_INFO, SIGNAL, and INSTALL_DRIVER.

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

Further Reference

---

o GS/OS Device Driver Reference

### 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

o 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

- GS/OS Reference

### END OF FILE TN.GSOS.008

```
#####################################################################
### FILE: TN.GSOS.009
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

GS/OS
#9: Interrupt Handling Anomalies

Revised by: Matt Deatherage   May 1992
Written by: Dave Lyons   January 1990

This Technical Note discusses anomalies in the way GS/OS handles interrupts.

CHANGES SINCE MAY 1990: Added discussions about changes to GS/OS interrupt handling since System Software 5.0.2.

PROBLEMS INSTALLING INTERRUPT HANDLERS

If your application calls ALLOC_INT to install an interrupt handler for an external interrupt source, it works fine unless the SCSI Manager (GS/OS file SCSI.Manager) is installed, in which case the system eventually grinds to a halt with a message about 65536 unclaimed interrupts.

THE PROBLEMS

If any interrupt handlers are bound (using BindInt) to reference number $17 (IRQ.OTHER), the unclaimed interrupt count gets incremented if none of the BindInt routines claims the interrupt, even though any handlers installed with ALLOC_INT routines still need a chance to claim it. The 5.0.2 SCSI.Manager triggers this problem because it calls BindInt with vector reference number $17.

In addition, if one or more interrupt handlers are bound to the IRQ.OTHER vector (VRN $17), the interrupt is passed to the ALLOC_INT handler even if it was already claimed by a BindInt routine. If no ALLOC_INT routine claims the interrupt, the unclaimed-interrupt count is incremented. As documented in Apple IIgs Technical Note #18, Do-It-Yourself SCC Interrupts, you cannot successfully call BindInt with vector reference number $0009.

THE SOLUTION

An application may install both a BindInt routine and an ALLOC_INT routine. If they both claim the external interrupt, the unclaimed count does not get incremented. The solution is compatible with future System Software releases, since it does not depend upon the ALLOC_INT routine ever getting called.

Your application's BindInt routine sees the interrupt before your ALLOC_INT routine does, so the BindInt routine should figure out whether the interrupt was caused by your external device, and claim it if so. Your ALLOC_INT routine should claim an interrupt it sees if and only if your BindInt routine claimed the last interrupt it saw.

Starting with GS/OS version 3.2 (released with the Apple II High-Speed SCSI Card), the system no longer treats too many unclaimed interrupts as a fatal error. However, before version 6.0, it still counts the unclaimed interrupts so it can do something like display a dialog asking you to restart even though choosing "restart" returns you to the application unharmed (GS/OS version 3.2), or sometimes display a dialog box sending you to your dealer and sometimes not (version 3.3), or do nothing about it at all (version 4.0 and later). This is obviously as confusing to most of us as it was to the system itself, so fortunately GS/OS now ignores unclaimed interrupts and doesn't even bother counting them.

PROBLEMS REMOVING INTERRUPTS HANDLERS

The GS/OS Reference suite says that device drivers may make BindInt and UnbindInt calls, noting this as an exception to the general rule that drivers may not make GS/OS system calls. What the references fail to note is that these calls may fail for an incredibly annoying reason--the OS may be busy.

GS/OS takes special pains to avoid this while starting and while switching to ProDOS 8, but it does not avoid this condition during an OSShutDown--a real shutdown of the OS, not a switch to ProDOS 8.

Driver authors can work around this problem by using a new system service call provided in GS/OS version 3.2 and later. The call, named UNBIND_INT_VECTOR, provides the functionality of UnbindInt to FSTs and drivers only to avoid the OS reentrancy issue. The vector is at $01/FCD8 and takes an interrupt identification number (as returned from BindInt) in the accumulator.

Further Reference

- GS/OS Reference
- Apple IIgs Technical Note #18, Do-It-Yourself SCC Interrupts

### END OF FILE TN.GSOS.009

```
#####################################################################
### FILE: TN.GSOS.010
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

GS/OS

#10: How Applications Find Their Files

Revised by: Matt Deatherage May 1992
Written by: Dave Lyons January 1990

This Technical Note explains how applications should find configuration and other application-related files.

CHANGES SINCE SEPTEMBER 1990: Lists new ways to access the @ prefix under System Software 6.0 and later.

When an application is launched, GS/OS sets prefix 9 to the application's parent directory. It also sets prefix 1 to the same directory if the length of the pathname is within a 64-character limit. It does not set prefix 0 to any special value.

If your application uses a partial pathname and depends upon prefix 0 to find files at the same directory level, it may be working by accident (prefix 0 is accidently set to the right directory), and sooner or later it won't work.

If your application needs to load a file named TitleScreen, the best way is to use the pathname 9:TitleScreen. If you just use TitleScreen, you are using prefix 0, and you may or may not be looking in the right directory.

Files storing user-specific data should be stored in the at sign (@) prefix--this is just like prefix 9, except that it is set to the user's user folder on an AppleShare server if the application was launched from a server. Use @:MySettings rather than 9:MySettings or MySettings. (If you want to retrieve the value of the @ prefix, you can call ExpandPath on the pathname "@:".) Note that the @ prefix was introduced in System Software 5.0.

The @ prefix is useful only for applications, not for Desk Accessories, CDevs, initialization files, or anything else; this type of code can get the path of the user's folder by using the AppleShare FST's FST-Specific call GetUserPath.

Starting with System Software 6.0, you can also retrieve the value of the @ prefix by passing $FFFF (-1) to GetPrefix. You may also set the value of the @ prefix by passing $FFFF to SetPrefix, but only applications or system-wide utilities should ever change the @ prefix. Specifically, any DAs, CDevs, initialization files or others should not mess with the @ prefix to make their own file handling simpler.

Further Reference

- GS/OS Reference
- AppleTalk Technical Note #8, Using the @ Prefix

### END OF FILE TN.GSOS.010

```
#####################################################################
### FILE: TN.GSOS.011
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

GS/OS #11: About EraseDisk and Format

Revised by: Matt Deatherage November 1990
Written by: Dave Lyons & Matt Deatherage July 1990

This Technical Note explains how an application can tell when a user chooses Cancel from an EraseDisk or Format dialog box and explains why thefile_sys_ID field is ignored in class-zero calls.
Changes since July 1990: Noted that System Software 5.0.3 fixes some of these anomalies.

Detecting a Canceled Erase or Format Dialog Box

GS/OS Reference says that EraseDisk and Format return with the carry flag set and A equal to zero when the user cancels the operation. This is great, except that the calls actually return with the carry clear, making a Cancel hard to distinguish from a successful EraseDisk or Format operation. This happens in System Software 5.0.2 and earlier; it works as documented in GS/OS Reference in System Software 5.0.3 and later.

If you must use 5.0.2 or earlier versions of the system software, this Note presents a safe way around the problem, which works with all versions of the System Software:

1. In the parameter block for class-one EraseDisk or Format, set the fileSysID field to 0. (See note below.)
2. Make the call.
3. If the error code is non-zero, there was an error. Handle it.
4. Otherwise, the error code is zero. Check the fileSysID field in the parameter block. If it is still zero, the user chose to cancel the operation.

Note that this method only works for class-one calls. For the class-zero ERASE_DISK and FORMAT calls, the file_sys_ID word is only an input parameter and always remains unchanged.

About the Class-Zero file_sys_ID Parameter

Even though fileSysID is an input parameter for the class-zero calls ERASE_DISK and FORMAT, all versions of the system software ignore thesupplied value and always give the user a dialog for selecting a file system. This means no functionality is lost by putting a zero there.

The reasons for this decision are historical. Although the Apple IIgs ProDOS 16

Reference indicates that the input parameter file_sys_ID would be used in future versions to choose destination file systems, ProDOS 16 always returned an error if the file system specified was not $0001 (ProDOS).

Since this effectively means no ERASE_DISK or FORMAT call can be made under ProDOS 16 with any file_Sys_ID other than $0001, the GS/OS team chose to ignore the parameter and always give users the choice when using class zero calls. Otherwise, no program that existed when GS/OS was released would ever allow users to choose interleaves or file systems (they would always format for ProDOS, file system $0001). (Note that the class-one Format andEraseDisk calls have a new reqFileSysID parameter; if this field is present, the dialog box is bypassed.)

Further Reference

- GS/OS Reference
- Apple IIgs ProDOS 16 Reference

### END OF FILE TN.GSOS.011

###
### FILE: TN.GSOS.012
###

Apple II
Technical Notes

Developer Technical Support

GS/OS
#12: All About Notify Procs

Written by: Matt Deatherage
September 1990

This Technical Note discusses the GS/OS notification procedure new to System Software 5.0 and enhances the discussion of these procedures in the Addison-Wesley GS/OS Reference.

Why Do I Want To Be Notified?

GS/OS notification procedures (or "notify procs") are handy ways to let the operating system tell you when interesting things are happening. As documented in GS/OS Reference, they can tell you when you're switching to ProDOS 8 (and back), when disks are inserted or ejected, when GS/OS is shut down, and even when a change occurs to a volume.

However, getting these notifications is not as simple as installing a procedure. Some behaviors are due to the way device drivers are designed and some are due to the design of GS/OS or device hardware. This Note discusses a few slightly unusual situations you can encounter when dealing with notification procedures.

I Get "Parameter out of range," and There's Only One Parameter

It seems incongruous to get error $0053 ("Parameter out of range") when there's only one parameter, a pointer to the notification procedure. However, GS/OS checks the procedure header to ensure consistency. In particular, the flags field must not have any of the reserved bits set. Having any bits other than one through six set results in error $53; it ensures you do not get strange behavior or are not passed values you cannot comprehend.

I'm Not Getting Notified

You've written your notification procedure correctly and tested it, but when you run your application you can eject and insert disks until your arm falls off and your code is never called.

This is a side effect of the design of most Apple II peripherals--no hardware interrupt is generated when you eject a disk. Without an interrupt to grab the CPU's attention, the drive just sits there until someone actually asks the drive if a disk is present.

Well-designed GS/OS drivers look to see if a disk has been switched every time they get control and call the System Service routine SET_DISKSW, which in turn causes the notification procedures to be told the disk has been switched. However, the driver cannot set this chain in motion until it gets control.

The easiest way to do this is to loop through all on-line devices, issuing a device call to each in turn. When the driver gets control, it starts the ball rolling. Note that you must make a device call that actually causes driver code to be executed. This includes all the application level device calls with less than two parameters, except DRename and DInfo (the third parameter is a block count, which causes a Driver_Status call to the driver). These calls are handled entirely by the Device Manager without actually transferring control to any driver code. DStatus with a transferCount = 2 is a good choice.

I Get Notified About Insertion at Weird Times

When coming back to GS/OS from ProDOS 8, you get "insertion" notification even though no disks have actually been inserted. This is done for you by most drivers, which pretend that any media in the device has just come online at driver startup time--which is true as far as any application is concerned.

General Truths

Be careful when installing notification procedures from an application. Applications either go away or are made purgeable when they quit, and that means your notification procedure can get disposed. GS/OS tries to call the address anyway, and this is generally a bad idea. Make sure you remove all notification procedures before their code goes away.

Even though you have to poll to ensure you get disk insertion and ejection events, it's still useful to install notification procedures. The notification queue allows everyone who's interested in GS/OS events to be notified about them. Check the "disk has been switched" bit of the status word is not suitable, because this bit is only set once. If a desk accessory makes a status call to a switched device, it sees the "disk has been switched" bit and your application does not, so use the notification queue.

Operating system calls (i.e., Write) can generate volume changed events during execution; therefore, GS/OS could be busy when it calls your notification procedure. Volume changed events are not necessarily generated immediately. The AppleShare FST checks for volume changes approximately every 10 seconds, but it only generates these events for a given volume if it contains an open folder.

GS/OS can call your notification procedure from inside an interrupt, so make it short and sweet. One approach is setting a flag which you can check periodically from your main code; when the flag is set, you can process the event and clear the flag.

Further Reference

- GS/OS Reference

### END OF FILE TN.GSOS.012

```
####################################################################
### FILE: TN.GSOS.013
####################################################################
```

Apple II
Technical Notes

Developer Technical Support

GS/OS
#13: GS/OS Reference Update

Revised by: Matt Deatherage — May 1992
Written by: Matt Deatherage & Dave Lyons — November 1990

This Technical Note corrects and updates the Addison-Wesley Apple IIgs GS/OS Reference. Previous versions from APDA labeled Volume 1 or 2 are obsolete, and should no longer be used.

CHANGES SINCE DECEMBER 1991: Added new information about resource_eof and resource_blocks parameters.

CHAPTER 4, "ACCESSING GS/OS FILES"

PAGE 72: THE SYSTEM FILE LEVEL: HOW TO PROTECT AN OPEN FILE FROM THE APPLICATION

The class 1 SetLevel and GetLevel calls have a special option that allows you to open a file at an "internal" file level, so that it cannot be closed by an application making a Close call with reference number zero at any application level.

GetLevel and SetLevel actually accept two parameters, not just the one parameter (level) documented in Chapter 7. The second parameter, level_mode, is a Word that controls the internal range of the file level.

Only two values for level_mode are supported. A value of $8000 is the same as if the parameter wasn't present at all--the level calls behave just as documented in GS/OS Reference. A value of $0000 sets a special "system" or "internal" level--all files opened with an internal level are unaffected by any non-internal level.

The steps to open a file at an internal file level are:

1. Call GetLevel with pCount=2, level_mode=$0000. Save the returned level.
2. Call SetLevel with pCount=2, level = $0080 and level_mode = $0000.
3. Open a file or files with a class 0 or 1 Open call, or with OpenResourceFile (OpenResourceFile on System Software 5.0.4 and earlier does not try to protect your resource files from being accidentally closed by a Close(0)).
4. Call SetLevel with pCount=2, level_mode=$0000, and level = saved level.

You can use two parameters in all your level calls and set the second level_mode parameter to $8000 instead of omitting it if it will make writing your program easier.

To close your protected file, simply do a Close with the reference number. There is no need to fiddle with the file level when closing by reference number.

NDAs should close all their files at or before DeskShutDown time.

CHAPTER 6, "WORKING WITH SYSTEM INFORMATION"

PAGE 92: USING THE OPTIONLIST PARAMETER

The optionList parameter resembles a GS/OS output buffer in most important respects--it starts with a word indicating the size of the buffer, and each FST fills in the size of the actual data placed in the buffer in the second word.   If the buffer is too small to hold the data, the necessary size is placed in the second word and the FST returns the "buffer too small" error ($004F).

Usually, GS/OS input buffers only have one length word, because if you know how large the data is (and you do if you're the one passing it to GS/OS), you don't need another word telling you the same thing.   However, if you're trying to copy something like an optionList, you can wind up in a bit of a pickle.   Just because the buffer you've allocated is big enough to hold file system-specific information, that doesn't mean the information is necessarily present.

A good example of this problem is found in the System Software 6.0 ProDOS FST. In 6.0 and later, the ProDOS FST will take HFS Finder information (as returned by the AppleShare and HFS FSTs) in the optionList and place that information in an extended file's extended key block, so the file can be copied to and from ProDOS disks with no loss of Macintosh-specific information (such as the longer file types and creator types necessary to identify Macintosh files). The FST returns the same information (if present) in the output optionList.

However, previous versions of the ProDOS FST returned no information in the optionList. Suppose you archived a file and stored the optionList with the file's information under 5.0, and attempt to restore the file under 6.0 using a nice, large optionList buffer. The FST can't know whether the large buffer contains any information or not.

To remedy this problem, the second word of the optionList structure (reqSize in the figure on page 92) is now defined on input as well as output. On input, the word must contain the actual size of the data in the optionList; the first word continues to indicate the size of the entire buffer. If the buffer size and the actual data size are too small to make sense, any affected FSTs will ignore the input, knowing that it must be garbage.

Further details on how the ProDOS FST stores HFS Finder information can be found in ProDOS 8 Technical Note #25, "Non-Standard Storage Types."

CHAPTER 7, "GS/OS CALL REFERENCE"

PAGES 98-99: CHANGEPATH

On page 98, the Reference states that a subdirectory may not be moved into itself or into a directory the first subdirectory already contains.   For example, you may not change /v to /v/w or /v/w to /v/w/x.   Although this is correct, the System Software 5.0.x implementations of the ProDOS FST trash your disk if you try this with ChangePath. Do not try it on disks you want to keep.

On page 99, error $4E is described as "file not destroy-enabled." No, ChangePath doesn't destroy the file. The error should read "file not rename-enabled."

PAGE 120: DINFO CHARACTERISTICS WORD

The diagram for the characteristics word in the DInfo parameters has incorrect descriptions for bits 14 and 13. The diagram says bit 14 is set if the device is a linked device; in fact, bit 13 is set if the device is a linked device. Bit 14 is set if the device in question has a generated driver; the bit is clear for loaded drivers.

PAGE 129: THE CHARACTER DEVICE STATUS WORD

The diagram on the top of page 129 says that if bit 5 is set, the device is in no-wait mode. This is incorrect. To determine if a device is in no-wait mode, make the GetWaitStatus subcall described on page 130.

Bit 5 of the character device status word is set if there are one or more characters waiting to be read from the device. This is an assistance for developers, since generated character drivers don't support no-wait mode.

PAGE 132: GETFORMATOPTIONS FLAGS WORD

The diagram describing the flags word of GetFormatOptions is incorrect. Bits 0 and 1 are actually the format type, while bits 2 and 3 are the size multiplier. In other words, the two labels are backwards.

PAGE 142: FLUSH

The Flush call, under System Software 5.0.3 and later (GS/OS version 3.3) accepts a maximum of two parameters. If the second parameter is present, it is the flushType. The flushType Word specifies the type of flush to be performed. A flushType of $0000 is the standard flush, where all dirty blocks are written to disk. If flushType is $8000, however, only dirty data blocks are written to disk. Certain dirty system blocks (blocks that don't hold file data) may not be flushed in this fast flush, but volume and file integrity is maintained.

PAGE 151: GETDIRENTRY
PAGE 156: GETFILEINFO
PAGE 176: OPEN

Each of the above calls has optional resourceEOF and resourceBlocks paramters that are listed as "undefined" if the file has no resource fork. In System Software 6.0 and later, these fields are guaranteed to be zero if a given file has no resource fork.

APPENDIX A, "GS/OS PRODOS 16 CALLS"

PAGE 386: GETDIRENTRY BUFFER DESCRIPTION INCORRECT

On page 386, nameBuffer is described as a pointer to a buffer in which GS/OS returns a Pascal string containing the name of the file or directory entry (in

GetDirEntry). This is incorrect; all versions of GetDirEntry return GS/OS (word-length) strings for the directory entry.

Further Reference

- GS/OS Reference
- Apple IIgs Technical Note #71, DA Tips and Techniques
- ProDOS 8 Technical Note #25, Non-Standard Storage Types

### END OF FILE TN.GSOS.013

```
#####################################################################
### FILE: TN.GSOS.014
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

GS/OS
#14: The Console Driver Technical Note

Written by: Matt Deatherage May 1992

This Technical Note discusses the GS/OS Console Driver and related issues.

NEW 6.0 CHARACTER FEATURES DON'T WORK IN VERSION 3.2

The System Software 6.0 documentation (as of this writing, the GS/OS ERS) refers to a new Console Driver feature. The Console Driver now has the capability to return direct character-in and character-out vectors for improved throughput (gained by bypassing most of GS/OS's overhead). The vectors are obtained through new DStatus device-specific call $8007, GetVectors.

Unfortunately, in version 3.2 of the Console Driver (which ships with System Software 6.0), this call returns addresses which are almost the correct ones (in other words, they're wrong). If DInfo says the Console Driver is version 3.2 or earlier, don't try to use the GetVectors feature.

NO-WAIT MODE AND USER INPUT MODE CONFLICT

When you read from a GS/OS driver in no-wait mode, the driver is supposed to return as quickly as possible, reading as much information as possible and returning as soon as the request is filled or no more information is instantly available. This is the opposite of wait mode, where the driver waits until the read can be finished even if it takes forever.

This philosophy directly conflicts with the Console Driver's user input routine (UIR) mode, where standard human interface editing functions are available. For example, if you want to read seven characters from the Console Driver in UIR mode, the user should be able to type four characters and hit three backspaces and not worry that the read request will end since he pressed seven keys. The entire concept of UIR mode is that the user can take his time and edit his input until he's happy with it, then press a terminator key to end editing.

This is how the Console Driver works, in fact, even in no-wait mode. If you ask for even one character in UIR mode and no-wait mode, the Console Driver will let the user edit the one character until he presses a terminator.

If you want instant feedback, you must use raw input mode.

Further Reference

- GS/OS Reference
- System 6.0 Documentation for GS/OS

### END OF FILE TN.GSOS.014

```
########################################################################
### FILE: TN.HCGS.001
########################################################################
```

Apple II
Technical Notes

Developer Technical Support

HyperCard IIGS #1: Corrections to the Script Language Guide

Written by: Dan Strnad & Matt Deatherage

March 1991

This Technical Note corrects the HyperCard IIgs Script Language Guide from Addison-Wesley.

Appendix A: External Commands and Functions

Page 317: ReturnStat

Developers who worked with the beta version of HyperCard IIgs on Volume V of the Developer CD (or volume 4 of Developer Essentials) should pay special attention to the use of the returnStat parameter documented on page 317 ofthe manual, as this method for using HyperCard's error-reporting facilities wasnot present in beta versions of HyperCard.

Page 318: HyperCard IIgs callbacks

Before describing the callbacks, the Script Language Guide says that thefirst parameter to each callback is the parameter block pointer that HyperCard IIgs passes to the XCMD or XFCN. This is not correct; the XCMD/XFCN parameterblock is not passed to callback routines. Each callback uses only the parameters supplied with its description.

Pages 318-324: Callback descriptions

The numbers listed for each callback are actually decimal numbers, not hexadecimal. There should not be a "$" in front of each number.

Pages 325-330: Beep, an example XCMD

Although there are "beep" sample XCMDs provided with the HyperCard IIgsScript Language Guide, they do not necessarily build and execute unmodified. Specifically, depending on your compiler, there could be a linking problemwith the Pascal and C XCMDs as given in the manual.

XCMDs and XFCNs are code resources, and are therefore subject to the limitations listed in Apple IIgs Technical Note #86, Risking ResourcefulCode. The specific problem here is that most Pascal and C compilers will create at least three segments: ~globals, ~arrays, and main. An XCMD or XFCN can only have one segment and the entry point must come first. Not only must you link all the object segments into one segment, but you must specifically extractthe entry point and link it first. HyperCard will pass control to the first byte of the loaded XCMD or XFCN, and therefore this must be the entry point. The samples in Appendix A point this out in the code.

Actual buildable sample source for the "beep" XCMDs is available in APW andMPW IIgs format on Volume VI or later of the Developer CD Series (or volume 5 or later of Developer Essentials). A complete APW C sample is included below.

An APW Sample XCMD: "CBeep"

CBeep.c

```
/*------------------------------------------------------------------------

  file CBeep.c

  This XCMD has the following syntax:

    CBeep       beep once
    CBeep ##    beep n times
    CBeep ?     display usage information
    CBeep !     display version information

  Copyright Apple Computer, Inc.  1989-1991
  All Rights Reserved.

------------------------------------------------------------------------*/

#include <types.h>
#include <MiscTool.h>
#include <GSOS.h>
#include <HyperXCMD.h>

/*
    Globals
*/

int _toolErr;
XCMDPtr gParamPtr;


/*
    Forwards
*/
pascal void CBeep();



/*  We place the entry point function in its own segment, so the linker can
    extract it and ensure that it's first in the load file. */

segment "EntrySeg"

/*
   This is the entry point to the program.  Make sure this procedure
   comes first in the final OMF resource because this is where HyperTalk
   will be jumping in.

   For a really simple XCMD you could just put the code all in here, but
   for cleanliness' sake this example calls another routine from here.


*/
pascal void EntryPoint(paramPtr)
XCMDPtr paramPtr;
{
  CBeep(paramPtr);
}

/*  All other code & data is placed in the "Main" segment   */

segment "Main"



/*  The actual CBeep function.  Interpret parameters and beep the speaker
*/

pascal void CBeep(paramPtr)
XCMDPtr paramPtr;
{
  short     beepCount;
  short     counter;
  Str255    str;

  char  *formStr    = "\pAnswer \"FORM: CBeep {count}\"";
  char  *versionStr = "\pAnswer \"CBeep XCMD v1.0\" & return & \"(c) 1991
Apple Computer, Inc.\"";

  gParamPtr = paramPtr;     /* put in a global for easy access in other funcs
 */

  if (paramPtr->paramCount > 0) {
    ZeroToPas(*(paramPtr->params[0]), &str);

    beepCount = 0;

    if (str.text[0] == '?')         /* test for special characters  */
      SendCardMessage(formStr);
    else if (str.text[0] == '!')
      SendCardMessage(versionStr);

    else beepCount = StrToNum(&str);        /* not a special - take as # of
beeps */
  }
  else beepCount = 1;   /* no count, assume one */

  beepCount = (beepCount <= 15) ? beepCount : 15;   /* limit 15 beeps   */

  for (counter = 0; counter < beepCount; counter++) SysBeep();
}

CBeep.r
```

```
/*******************************************************************/
/*
/* CBeep.r
/*
/* Copyright (C) 1991
/* Apple Computer, Inc.
/* All Rights Reserved
/*
/* Rez source for building XCMDs.
/*
/*******************************************************************/

#include "types.rez"

read $801E (1, convert) "CBeep.omf";

resource rResName ($0001801E) {
            1,
            { 1, "CBeep";
            }
};
```

Make file

```
* ------------------------------------------------------------------
*
*  This makefile will build C XCMDs for HyperTalk
*
*  Copyright Apple Computer, Inc.  1991
*  All Rights Reserved.
*
*  Builds:  CBeep
*  This makefile depends on a .r file called CBeep.r to act
*  as a source for the resource compiler.

compile +t +e CBeep.c keep=CBeep

* ------------------------------------------------------------------
*
*  The compilers will output 3 or more segments:  main, containing code;
*  and ~globals and ~arrays containing data.  This line ensures that
*  everything gets put back into the main segment.
*
*  In addition, it specifically links the EntryPoint procedure FIRST,
*  ahead of any globals or data structures.

* The linker line is very long - make sure you use all of it

linkiigs -x -lseg main CBeep.root(@EntrySeg) CBeep.root(@Main)
CBeep.root(@~arrays) CBeep.root(@~globals) 2/CLib -lib 2/CLib -o CBeep.omf

compile CBeep.r keep=CBeep.rsrc

* now use your favorite resource utility to copy the XCMD from CBeep.rsrc
* into your stack
```

Further Reference

  o HyperCard IIgs Script Language Guide
  o Apple IIgs Technical Note #86, Risking Resourceful Code
  o HyperCard IIgs Technical Note #2, Known HyperCard Bugs

### END OF FILE TN.HCGS.001

```
#####################################################################
### FILE: TN.HCGS.002
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

HyperCard IIGS
#2: Known HyperCard Bugs

Revised by: Matt Deatherage May 1992
Written by: Dan Strnad & Matt Deatherage March 1991

This Technical Note documents known bugs in the released version of HyperCard IIgs that may affect developers.

CHANGES SINCE MARCH 1991: Revised to list version 1.1 bugs (sigh) as well as version 1.0 bugs.

HYPERCARD EXTERNALS AND NAMED RESOURCES

HyperCard's XCMD and XFCN callbacks documented in Appendix A of the HyperCard IIgs Script Language Guide include callbacks that find named resources. In versions 1.0 and 1.1, these routines don't compare the lengths of the resource name strings, which makes HyperCard return the wrong named resource from time to time.

A more precise description of this problem is in Apple IIgs Technical Note #83, "Resource Manager Stuff." Note that HyperCard IIgs does not use the Resource Manager's named resource routines, but the code in the Resource Manager suffers from the same problem the HyperCard code has.

PREVIOUS BUGS FIXED

The two bugs previously listed in this Note--improper handling of desk accessories and crashing when using objects or properties of different stacks to externals--are both fixed in HyperCard IIgs version 1.1.

Further Reference

- o HyperCard IIgs Script Language Guide
- o HyperCard IIgs standard documentation (included with HyperCard IIgs)

### END OF FILE TN.HCGS.002

```
########################################################################
### FILE: TN.HCGS.003
########################################################################
```

Apple II
Technical Notes

Developer Technical Support

HyperCard IIGS #3: Pitching Sampled Sounds

Written by: Mark Cecys & Matt Deatherage

March 1991

This Technical Note describes the "relative pitch" field used in sound resources played by HyperCard (and sound scraps that HyperCard doesn't use)--what it does and what to put in it.

What is this relative pitch thing?

There are basically two ways to use a sound sample, in HyperCard or anywhere else: as a sample of a wave of definite pitch, or as a miniature "tape recording" of some sound that is not intended to be used as a sample of indefinite pitch.

Definite Pitch

To play a sample at the correct pitch, HyperCard assumes two things about the sample: it was sampled at a rate of 26.32 KHz, and the associated wave was playing a pitch of 261.63 Hz, when it was sampled.

In the real world, where most of us live, this is not very practical. To help compensate for reality, the sample sound format includes a "relative pitch" field, which can tell HyperCard (or anyone else playing the sound) how to compensate for the difference in pitch between the sample's actual pitch and a pitch of 261.63 Hz.

Follow these steps to calculate the relative pitch parameter for a given sampled sound resource. If the wave is of definite pitch, you must know the frequency of the source wave and the sampling rate for the sample in question.

1. Calculate the difference ratio r. In the equation below, Fw is the frequency of the sample (in Hz) and Fs is the sampling rate for the sample.

$$r = \frac{261.63}{Fw} \times \frac{Fs}{26,320}$$

2. Extract an offset to the pitch:

$$offset = 3072 \times \log (r)$$

(Remember that you can substitute (ln(r)/ln(2)) if your calculator doesn't provide the log in base 2.)

3. If offset is negative, make it positive and set bit 15 to tell sound players to lower the pitch instead of raise it. If offset is negative:

   relative = |offset| + $8000

   If offset is positive:

   relative = offset

That's all. Store the value of tuning in the sampled sound for the "relative pitch" field and HyperCard will take care of the rest.

Indefinite pitch

Sounds which are not samples of definite pitch (for example, a thunder clap or the sound of your mother saying "hello") should not need to be made to match pitch. Only sounds produced using optional parameters of HyperCard's Play command need to go through the same process outlined for "Definite pitch". In these cases, however, you don't need to worry about the frequency of the sample. Instead of using the equation provided in step 1 above, use this instead:

$$r = \frac{Fs}{26,320}$$

(or just use 261.63 for Fw.) Take the value of r and use it for steps two and three above.

A HyperTalk sample

The following simple button script will calculate the correct value of relativefor you, given the other values in card fields named Fw, Fs and card fields named offset and relativeto use as containers:

```
on mouseUp
lock screen
set numberFormat to "0"
put the value of card field Fs * 261.63 into r
put the value of card field Fw into denominator -- the bottom of the fraction
multiply denominator by 26320
divide r by denominator

put log2(r) into card field offset
multiply card field offset by 3072

if card field offset <0 then
    put abs(the value of card field offset) into card field tuning
    add 32768 to card field relative
end if

unlock screen
```

```
end mouseUp
```

Further Reference

- HyperCard IIgs Script Language Guide
- Apple IIgs Technical Note #76, Miscellaneous Resource Formats
- Apple IIgs Technical Note #99, Supplemental Scrap Types

### END OF FILE TN.HCGS.003

##
### 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:

- o saving all machine state information in locations that the debugger can access
- o setting the machine to a known state
- o passing control to an arbitrary debugger
- o restoring the remembered machine state upon regaining control from the debugger
- o 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: Dave Lyons January 1991
Written by: Dan Oliver October 1986

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

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

Information Bar Initialization

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

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

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

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

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

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

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

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

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


; --- Dereference menu bar handle
----------------------------------------------------------
;
        ldy    #2
        lda    [menuBar],y
        tay
        lda    [menuBar]
        sta    <menuBar                 Now menuBar is the pointer to the Menu
Bar.
        sty    <menuBar+2
;
;
; --- Set size of menu bar
-----------------------------------------------------------------
;
;
        lda    <tempRect+y1
        dec    a                        Overlap top side.
        ldy    #CtlRect+y1
        sta    [menuBar],y
;
        lda    <tempRect+x1
        dec    a                        Overlap left side.
        ldy    #CtlRect+x1
        sta    [menuBar],y
;
        lda    <rect+y2
        inc    a                        Overlap bottom side.
        ldy    #CtlRect+y2
        sta    [menuBar],y
;
;
; --- Set flag to tell Menu Manager to draw menu in current port
---------------------------
;
        ldy    #CtlOwner+2              Set high bit in CtlOwner.
        lda    [menuBar],y
        ora    #$8000
        sta    [menuBar],y
;
;
; --- Create the menus and add them to the window's menu bar
-------------------------------
```

```
;
        lda    #4
loop    pha                             Save index into menu list.
        tay                             Switch index to Y.
;
        pha                             Space for return value.
        pha
        lda    menu_list+2,y            Pass address of menu/item lines.
        pha
        lda    menu_list,y
        pha
        _NewMenu
;                                       Menu handle already on stack.
        pea    0                        Insert menu list at front of list.
        _InsertMenu                     Add my menus to the system menu bar.
;
        pla
        sec
        sbc    #4
        bpl    loop
;
;
; --- Initialize the size of the menu bar and menus
----------------------------------------
;
        pha                             Space for returned bar height.
        _FixMenuBar                     Fix up positions in the menu bar.
        pla                             Discard height of menu bar.
;
;
; --- Restore the system menu bar as the current menu
--------------------------------------
;
        pea    0                        Pass flag for system menu bar.
        pea    0
        _SetMenuBar                     Make system menu bar current.
```

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

Information Bar Definition Procedure (InfoDefProc)

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

The Window Manager passes your InfoDefProc the following information:

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, save in Y.
;
        tsc                     Move direct page point to stack.
        tcd
        lda    2,s              Move return address 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 informationbar.

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

Other Information Bar Uses

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

- o Display text information, as in Macintosh Finder windows.
- o Split window. Like the content region, the information bar could be large enough to hold data.
- o 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

---

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

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

Why Change Resolution?

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

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

How To Change Modes

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

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

```
;
```

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

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

Windows with vertical scroll bars in the window frame are the same width when you change modes, so switching from 320 mode to 640 mode results in anarrower 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 tothe current mode. If, as suggested above, you resize all the windows after the mode change and before calling RefreshDesktop, you should be in good shape. If you choose not the follow this recommendation, you should call SizeWindow for every window with scroll bars and change the size of each window at least one pixel since SizeWindow does not do anything if the passed size is not different than the current size.

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

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

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

Reinstalling Large Fonts

After restarting QuickDraw II, you should call InstallFont again on the fontsyour application is using. This causes the Font Manager to callInflateTextBuffer so that QuickDraw can draw text correctly in large fontsizes.

Repositioning and Resizing Windows in the New Mode

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

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

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

```
             _GetNextWindow     ;Get the pointer to the next window.
;
enter        pla                ;Remember the pointer to this window.
             sta <window
             pla
             sta <window+2
;
             ora <window        ;Are there any more windows?
             bne loop
;
```

WindNewRes

Generally, WindNewRes does the following:

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

MenuNewRes

Generally, MenuNewRes does the following:

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

CtlNewRes

Generally, CtlNewRes does the following:

- o chooses the proper icon font for radio button, check box, grow box and scroll bar arrows
- o 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 1990
Written by: Dan Oliver October 1986

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

Strings used for window titles should always have a space as the first and last characters. This spacing is especially important for windows that use a lined window title bar since, without the beginning and ending space, theline 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 withspaces 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 ahole in the title bar pattern than to compute the rectangle, fill it, and draw the text.

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

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

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

### 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

- Apple IIGS Toolbox Reference, Volume 2
- 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
o 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
o   Apple IIGS Toolbox Reference, Volume 2
o   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 & Dave Lyons — May 1992
Written by: Jim Merritt — April 1987

This Technical Note lists all known interdependencies between system tool sets on the Apple IIgs.

CHANGES SINCE JANUARY 1990: Added new and changed dependencies for System Software 6.0.

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

TOOL SET INTERDEPENDENCIES

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

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

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

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

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

Event Manager                                   Tool #6 ($06)
    Tool Locator        (#1)
    Memory Manager      (#2)
    Miscellaneous Tools (#3)

Scheduler                                       Tool #7 ($07)
    Tool Locator        (#1)
    Memory Manager      (#2)
    Miscellaneous Tools (#3)

Sound Tools Set                                 Tool #8 ($08)
    Tool Locator        (#1)
    Memory Manager      (#2)
    Miscellaneous Tools (#3)

Apple Desktop Bus (ADB)                         Tool #9 ($09)
    Tool Locator        (#1)

SANE (Standard Apple Numeric Environment)       Tool #10 ($0A)
    Tool Locator        (#1)
    Memory Manager      (#2)

Integer Math Tools                              Tool #11 ($0B)
    Tool Locator        (#1)

Text Tools                                      Tool #12 ($0C)
    Tool Locator        (#1)

Window Manager                                  Tool #14 ($0E)
    Tool Locator        (#1)
    Memory Manager      (#2)
    Miscellaneous Tools (#3)
    QuickDraw II        (#4)
    Event Manager       (#6)
  * QuickDraw Aux       (#18)                   Required in 6.0 and later,
                                                and the window manager loads
                                                and starts it for you.
    Control Manager     (#16)
    Menu Manager        (#15)
    * Line Edit         (#20)                   For AlertWindow call only
    * Font Manager      (#27)                   For AlertWindow call only
    * 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)
    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)
  * QuickDraw Auxiliary (#18)                   Required in 6.0 and later,
                                                and the Window Manager loads
                                                and starts it for you.
    Line Edit           (#20)
    Dialog Manager      (#21)
  * List Manager        (#28)
  * Resource Manager    (#30)                   For using resources in
                                                Standard File Operations
                                                calls.

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

Note Synthesizer                                Tool #25 ($19)
    Tool Locator        (#1)
    Memory Manager      (#2)
    Sound Tools         (#8)

Note Sequencer                                Tool #26 ($1A)
    Tool Locator        (#1)
    Memory Manager      (#2)
    Sound Tools         (#8)
    Note Synthesizer    (#25)

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

Font Manager                                  Tool #27 ($1B)
    Tool Locator        (#1)
    Memory Manager      (#2)
  * Miscellaneous Tools (#3)                  For ChooseFont call only.
    QuickDraw II        (#4)
  * Integer Math Tools  (#11)                 For ChooseFont call only.
  * Window Manager      (#14)                 For ChooseFont call only.
  * Control Manager     (#16)                 For ChooseFont call only.
  * Menu Manager        (#15)                 For FixFontMenu call only.
  * List Manager        (#28)                 For FixFontMenu
                                              and ChooseFont calls.
  * Line Edit           (#20)                 For ChooseFont call only.
  * Dialog Manager      (#21)                 For ChooseFont call only.

List Manager                                  Tool #28 ($1C)
    Tool Locator        (#1)
    Memory Manager      (#2)
    Miscellaneous Tools (#3)
    QuickDraw II        (#4)
    Event Manager       (#6)
    Window Manager      (#14)
    Control Manager     (#16)
    Menu Manager        (#15)

Audio Compression and Expansion (ACE)         Tool #29 ($1D)
    Tool Locator        (#1)
    Memory Manager      (#2)

Resource Manager                              Tool #30 ($1E)
    Tool Locator        (#1)
    Memory Manager      (#2)


MIDI Tools                                    Tool #32 ($20)
    Tool Locator        (#1)
    Memory Manager      (#2)
    Miscellaneous Tools (#3)
    Sound Manager       (#8)
  * Note Synthesizer  (#25)


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

Text Edit                                         Tool #34 ($22)
    Tool Locator        (#1)
    Memory Manager      (#2)
    Miscellaneous Tools (#3)
    QuickDraw II        (#4)
    Event Manager       (#6)
    Window Manager      (#14)
    Menu Manager        (#15)
    Control Manager     (#16)
    QuickDraw Auxiliary (#18)
    Scrap Manager       (#22)
    Font Manager        (#27)
  * Resource Manager    (#30)                     For using resources in Text
                                                  Edit calls.

MIDI Synth                                        Tool #35 ($23)
    Tool Locator        (#1)
    Memory Manager      (#2)
    Miscellaneous Tools (#3)
    Sound Tools         (#8)

Media Control Tool                                Tool #38 ($26)
    Tool Locator        (#1)
    Memory Manager      (#2)
    Miscellaneous Tools (#3)
    Integer Math        (#11)
    Resource Manager    (#30)
```

Recommended Start Order

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

```
    Tool Locator        (#1)
                        ADB Tools                 (#9)
                        Integer Math Tools        (#11)
                        Text Tools                (#12)
    Memory Manager      (#2)
                        SANE                      (#10)
                        ACE                       (#29)
    Resource Manager    (#30)
    Miscellaneous Tools (#3)
                        Scheduler                 (#7)
                        System Loader             (#17)     LoaderStartup does
                                                            nothing.
```

```
                    Media Control           (#38)
QuickDraw II        (#4)
                    QuickDraw II Auxiliary   (#18)
Event Manager       (#6)
Window Manager      (#14)
Control Manager     (#16)
Menu Manager        (#15)
Line Edit           (#20)
Dialog Manager      (#21)
either
        Sound Tools then     (#8)
        Note Synthesizer     (#25)
or
        Note Sequencer       (#26)
        MIDI Tools           (#32)
        MIDI Synth           (#35)
Standard File       (#23)
Scrap Manager       (#22)
List Manager        (#28)
Font Manager        (#27)
Print Manager       (#19)
Text Edit           (#34)
Desk Manager        (#5)
```

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

Further Reference

- o Apple IIgs Toolbox Reference

### 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 Screwiness

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

This Technical Note describes known anomalies in Standard File.

CHANGES SINCE DECEMBER 1991: Updated for System 6.0. Problems with the infinite loop and SFMultiGet2 reply record are fixed.

PREFIX CHECK IS CASE SENSITIVE

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

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

The following two problems are fixed in System 6.0:

INFINITE LOOP WITH EMPTY PREFIXES

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

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

SFMultiGet2 (AND SFPMultiGet2) REPLY RECORD

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

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

Further Reference

- Apple IIgs Toolbox Reference, Volumes 2 & 3

### 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

- 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 Access

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

This Technical Note describes how to install and remove a interrupt handler routine for the Z8530 Serial Communications Controller (SCC) on the Apple IIgs without breaking other parts of the system. This Note includes many suggestions that, if unheeded, could come back to haunt you in the form of bug fixes to your program.
Changes since March 1990: Added a method for finding which serial port AppleTalk is using under GS/OS.

Free Serial Routines Inside

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

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

Don't Eat Your Serial with Your Mouth Open

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

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

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

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

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

WhichPort           start

IDROUTINE           equ $FE1F           returns system ID information

                    stz ApTalkPort      default to not AppleTalk

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

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

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

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

                    sec                 emulation mode
                    xce
                    longa off
                    longi off

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

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

ItsPort2            inc ApTalkPort      port 2 is AppleTalk

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

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

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

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

                    sec                 emulation mode
                    xce
                    longa off
                    longi off

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

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

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

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

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

```
;
```

```
; This routine checks to see which serial port, if any, AppleTalk is using.
; The routine sets a flag byte, ApTalkPort, and the accumulator to indicate
; which port (if any) AppleTalk is using.
;    $0000 = AppleTalk is not using a serial port
;    $0001 = AppleTalk is using serial port 1 (printer port)
;    $0002 = AppleTalk is using serial port 2 (modem port)
; Note:  This method should be used under GS/OS only.
;
; Enter routine in native 16 bit mode
;
                    longa on
                    longi on
                    mcopy 2/AInclude/M16.GSOS

CheckPort           Start

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

                    phb                 save data bank
                    phk                 data bank = code bank
                    plb

                    lda #$0001          start with device #1
                    sta DIdevNum

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

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

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

NotFound            stz ApTalkPort      neither port is in use
                    bra Exit

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

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

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

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

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

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

                    end
```

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

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

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

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

The Z8530 SCC has four registers which are shared by both channels (ports). Of those four, only two are commonly used in the Apple IIgs, RR3 and WR9.

RR3, which only exists in channel A, lets you check the interrupt pending bits for both SCC channels. WR9 is the Master Interrupt Control register for both SCC channels and contains the Reset command bits.

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

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

Hints for the Serial Adventure

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

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

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

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

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

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

```
                    longa off           must be using 8-bit accumulator
                    longi off             and index registers
;
; Write to a SCC command register - channel A or B.
; Input:  A = value to store
;         X = SCC register number ($0-$F)
;         Y = $01  channel A
;             $00  channel B
;
WriteSCC            php                 save the current interrupt status
```

```
                    sei                 disable interrupts
                    pha                 save value to write
                    txa                 get SCC register number from X
                    sta $C038,y         set the register number
                    pla                 restore value to write
                    sta $C038,y         write the value
                    plp                 restore the interrupt status
                    rts

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

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

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

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

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

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

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

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

Don't be a Serial Killer

How to Install and Remove your SCC Interrupt Handler

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

The steps for installing your SCC interrupt handler are:

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

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

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

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

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

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

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

1. Disable the SCC Master Interrupt Enable (WR9, bit 3) briefly while performing the next six steps. The value you should write to WR9 is 00000010.

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

Further Reference

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

### 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 March 1990
Written by: Jim Merritt August 1987

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

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

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

Further Reference

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

### 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 Use of Dynamic Segments

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

This Technical Note discusses strategies that applications can use to deal with dynamic segments.
Changes since November 1988: Rewrote from scratch to address current problems.

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

Loading Dynamic Segments

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

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

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

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

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

Unloading Dynamic Segments

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

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

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

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

Dynamic Segment Interdependencies: Just Say No

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

```
            Main Program
    Dispatcher & User Interface  <-- static

         /       |       \
        /        |        \

     Mode 1    Mode 2    Mode 3  <-- dynamic
      Code      Code      Code

        \        |       /
         \       |      /
        Shared Utility Code      <-- static

    Figure 1-Sample Program Layout
```

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

A Final Warning:

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

Further Reference

- GS/OS Reference
- Apple IIgs Programmer's Workshop Assembler Reference

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

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

CHANGES SINCE DECEMBER 1991: Added corrections to Dialog Manager, Menu Manager, Tool Locator, Window Manager, and Appendix E.

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

CORRECTIONS TO VOLUME 1

DESK MANAGER--FIXAPPLEMENU CAN DIE WITH ERROR $0512

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

DIALOG MANAGER--EDITLINE ITEM VALUE

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

THE LIST MANAGER WANTS THE PORT SET PROPERLY

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

DELETEMITEM OPERATES ON THE CURRENT MENU BAR

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

ERROR $0F02 FROM GETMITEM

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

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

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

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

SETVECTOR REFERENCE NUMBERS

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

GETTING A CLEAN MOUSE MODE FROM READMOUSE

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

READASCIITIME RESULT BUFFER

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

SYSTEMEVENT IS ALL BACKWARDS

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

CORRECTIONS TO VOLUME 2

QUICKDRAW AUXILIARY ERROR CODES

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

```
    $1210: picEmpty
    $1211: picAlreadyOpen
    $1212: pictureError
    $1221: badRect
    $1222: badMode
```

FRAMERGN DOES NOT CONTRIBUTE TO AN OPEN REGION

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

TOOL LOCATOR, TLMOUNTVOLUME

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

TOOL LOCATOR, SETTSPTR

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

WINDOW MANAGER, "DRAW INFORMATION BAR ROUTINE"

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

WINDOW MANAGER, INVALRECT

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

WINDOW MANAGER, PINRECT

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

WINDOW MANAGER, SETZOOMRECT

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

WINDOW RECORD OFFSETS

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

APPENDIX A, "WRITING YOUR OWN TOOL SETS"

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

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

CORRECTIONS TO VOLUME 3

CONTROL MANAGER: MENU EVENTS

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

CONTROL MANAGER: DIMMED CUSTOM CONTROLS

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

CONTROL MANAGER: SIZE BOX CONTROLS

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

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

DESK MANAGER: ERRORS FROM ADDTORUNQ AND REMOVEFROMRUNQ

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

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

EVENT MANAGER: WHAT SETAUTOKEYLIMIT REALLY DOES

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

LIST MANAGER

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

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

MEMORY MANAGER

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

MENU MANAGER

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

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

MISCELLANEOUS TOOLS: INTERRUPT STATE RECORD NOT ALWAYS COMPLETE

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

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

Table 1--Validity of Interrupt Record

STANDARD FILE

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

TOOL LOCATOR: NOTES ON STARTUPTOOLS

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

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

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

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

WINDOW MANAGER:NEWWINDOW2 PARAMETERS OVERRIDE TEMPLATE EVEN WHEN YOU PASS NIL

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

APPENDIX E: RTEXTFORLETEXTBOX2 RESOURCES

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

APPENDIX E: RTWORECTS RESOURCES

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

Further Reference:

- o   Apple IIgs Toolbox Reference, Volumes 1-3
- o   Apple IIgs Technical Note #101, Patching the Toolbox

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

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

CHANGES SINCE SEPTEMBER 1990: Added a reference to Apple IIgs Technical Note #102 for TOBRAMSETUP.

CONTENTS

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

CHAPTER 2: NOTES FOR PROGRAMMERS

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

CHAPTER 3: SYSTEM MONITOR FIRMWARE

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

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

Page 43, Back to BASIC: The last paragraph should read: "If you are using DOS 3.3 or ProDOS(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 4: VIDEO FIRMWARE

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

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

CHAPTER 5: SERIAL-PORT FIRMWARE

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

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

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

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

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

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

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

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

Page 105, GetIntInfo: The command list should read:

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

The following should be added after the command list: "Note: The

Parameter count of $03 is correct even though there are four parameters."

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

Page 106, SetIntInfo: The command list should read:

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

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

CHAPTER 7: SMARTPORT FIRMWARE

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

This is an example of a standard SmartPort call:

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

This is an example of an extended SmartPort call:

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

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

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

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

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

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

Page 125, Possible errors: Add the following:

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

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

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

	Standard call	Extended call
CMDNUM	$06	$46

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

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

```
----------------------------------------------------------------------
|13      |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 141, SetInterleave: The Control code and Control list should be:

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

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

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

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

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

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

Page 166, Table 7-8, Standard command packet contents":

Byte 3 descriptions should read "Byte 2 of param list."
Byte 4 descriptions should read "Byte 3 of param list."
Byte 5 descriptions should read "Byte 4 of param list."
Byte 6 descriptions should read "Byte 5 of param list."
Byte 7 descriptions should read "Byte 6 of param list."
Byte 8 descriptions should read "Byte 7 of param list."
Byte 9 descriptions should read "Byte 8 of param list."

CHAPTER 8: INTERRUPT-HANDLER FIRMWARE

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

CHAPTER 9: APPLE DESKTOP BUS MICROCONTROLLER

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

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

CHAPTER 10: MOUSE FIRMWARE

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

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."

APPENDIX D: VECTORS

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

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

Further Reference:

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

### 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

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

AppleTalk

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

Control Panel CDA

- o The original Options menu is now the Keyboard menu and does not contain mouse parameters.
- o 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.
- o 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.
- o If slot 7 is set to AppleTalk, the Control Panel displays a warning if neither slot 1 nor slot 2 is similarly set.
- o The Printer Port and Modem Port menus now display only those parameters that may be changed if AppleTalk is the selection for those ports.
- o The RAM disk no longer has minimum and maximum settings, but rather one RAM disk size setting.

Monitor

- o Enhanced memory searching commands to automatically cross bank boundaries.
- o Added Step and Trace debugging functions.
- o 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.
- o Now resize the RAM disk when the system is rebooted with the Control-Open Apple-Shift-Reset key combination.
- o Handle text page 2 shadowing and power-up bits in the new CYA chip.
- o Flash the border if the sound volume is set to zero and a beep is necessary.
- o 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: Jim Luther — September 1990
Written by: Rilla Reynolds & Jim Luther — October 1987

This Technical Note includes updates to the Apple IIgs Hardware Reference, published by Addison-Wesley. 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 these manuals.
Changes since July 1990: Changed the description in "Signals at the Serial Ports and the Serial Communications Controller" to correctly note that the SCC can support a maximum asynchronous transmission rate of 57,600 bits per second (bps) in X16 clock mode.

There are two editions of the Apple IIgs Hardware Reference, the first edition (July 1987) which covers the original Apple IIgs only, and the second edition (1989) which covers both original Apple IIgs and the 1 MB Apple IIgs. Because page numbers have changed between the two editions and because an update to one edition may not be needed in both editions, this Note organizes corrections by chapter, always noting corrections to the Second Edition followed by corrections to the First Edition.

Chapter 3: Memory

Second Edition--Page 40, Table 3-2, Bits in the State register
First Edition--Page 36, Table 3-2, Bits in the State register

Switch the given values and descriptions for bits 7 and 2 as follows:

Bit	Value	Description
7	1	ALTZP: If this bit is 1, then bank-switched memory, stack, and direct page are in auxiliary memory.
	0	If this bit is 0, then bank-switched memory, stack, and direct page are in main memory.
2	1	LCBNK2: If this bit is 1, language-card RAM bank 2 is selected.
	0	If this bit is 0, language-card RAM bank 1 is selected.

Chapter 6: The Apple Desktop Bus

Second Edition--Page 148, after final paragraph

Add a new heading and description:

Control Panel Control Jumper

The ADB microcontroller provided with the 1 MB Apple IIgs includes an input that disables the text Control Panel (normally available via the Classic Desk Accessory menu). This feature allows the system parameters to be set and then protected from changes made via the text Control Panel. A jumper across the pins of connector S1 removes the text Control Panel from the Classic Desk Accessory menu. All other installed classic desk accessories are still available in the Classic Desk Accessory menu when the S1 jumper is installed. The S1 connector is located near the ADB microcontroller at motherboard location F12.

Note: The S1 jumper does not prevent the system parameters from being changed with the graphic Control Panel (a new desk accessory normally available from the Apple menu of the Finder or of any other application that includes the Apple menu).

First Edition--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

```
    A3   A2   A1   A0   0    0    0    0    Device Reset
```

First Edition--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."

First Edition--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

Second Edition--Page 154, Table 7-3, Disk-port soft switches
First Edition--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."

Second Edition--Page 155, Table 7-4, IWM states
First Edition--Page 146, Table 7-4, IWM states

Change the table to the following:

Q7	Q6	Drive	Operation
0	0	enabled	Read Data register
0	1	-	Read Status register
1	0	-	Read Handshake register
1	1	disabled	Write Mode register
1	1	enabled	Write Data register

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

First Edition--Page 146, after Table 7-4, IWM states

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	-	-	-	0	0
-	1	1	-	1	0
-	1	-	1	0	1

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

First Edition--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.

Second Edition--Pages 156-7, Table 7-5, Bits in the Mode register
First Edition--Pages 147-8, Table 7-5, Bits in the Mode register

For Second Edition, change the description for bit 2, value 0 as shown. For First Edition, switch the given values and descriptions for bits 1, 2, and 4 as shown.

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

disk drives.

Second Edition--Page 159, The serial ports
First Edition--Page 150, The serial ports

The first sentence should read: "The Apple IIgs has two serial ports located at the back of the computer, which may provide synchronous and asynchronous serial communications."

Second Edition--Page 160, Table 7-9, Pins on a serial-port connector
First Edition--Page 151, Table 7-8, Pins on a serial-port connector

Replace the table title and table with this table title, table and note:

Table 7-x   Signal assignments for the mini 8-pin serial port connectors

Pin Number	Signal name	Signal Description
1	HSKo	Handshake output. Driven uninverted from the SCC's /DTR output. Voh = 3.6V; Vol = -3.6V; Rl - 450 ohms
2	HSKi	Handshake input or external clock. Received inverted at SCC's /CTS and /TRxC inputs. Vih = 0.2V; Vil = -0.2V; Ri = 12K ohms
3	TxD-	Transmit data (inverted). Driven inverted from SCC's TxD output; tri-stated when SCC's /RTS is not asserted. Voh = 3.6V; Vol = -3.6V; Rl = 450 ohms
4	GND	Signal ground. Connected to logic and chassis ground.
5	RxD-	Receive data (inverted). Received inverted at SCC's RxD input. Vih = 0.2V; Vil = -0.2V; Ri = 12K ohms
6	TxD+	Transmit data. Driven uninverted from SCC's TxD output; tri-stated when SCC's /RTS is not asserted. Voh = 3.6V; Vol = -3.6V; Rl = 450 ohms
7	GPi	General-purpose input. Received inverted at SCC's /DCD inputs. Vih = 0.2V; Vil = -0.2V; Ri = 12K ohms
8	RxD+	Receive data. Received uninverted at SCC's RxD input. Vih = 0.2V; Vil = -0.2V; Ri = 12K ohms

Note: Absolute values of specified voltages are minimums;
Ri is a minimum, Rl is a maximum.

Second Edition--Page 164, after Figure 7-9
First Edition--Page 155, after Figure 7-9

Add a new heading and description:

Signals at the Serial Ports and the Serial Communications Controller

The Apple IIgs has two serial ports which are compatible with most RS-232-C devices. This section describes the input and output signals provided at the serial ports. This section also discusses some input signals to the 8530 Serial Communications Controller (SCC) chip that are not described in the Apple IIgs Hardware Reference.

The transmit-data and receive-data lines of the Apple IIgs serial interface conform to the EIA standard RS-422, which differs from the more commonly used RS-232-C standard in that, whereas an RS-232-C transmitter modulates a signal with respect to a common ground, an RS-422 transmitter modulates the signal against an inverted copy of the same signal (to generate a differential signal). The RS-232-C receiver senses whether the received signal is sufficiently negative with respect to ground to be logical 1, whereas the RS-422 receiver simply senses which line is more negative than the other. An RS-422 signal is therefore more immune to noise and interference, and degrades less over distance, than an RS-232-C signal. If you ground the positive side of each RS-422 receiver and leave unconnected the positive side of each transmitter, you have essentially converted to EIA standard RS-423, which can be used to communicate with most RS-232-C devices over distances up to fifty feet, as illustrated in Figures 7-x1 and 7-x2.

```
................................................
.    8530   26LS32 Receivers  IIGS Mini 8-pin  .      RS-232-C DTE Device
.    SCC    & 26LS30 Drivers  Serial Connector .        DB-25 Connector
.  _______                                     .         ____________
. |       |        /|             ______       .        |            |
. |       |       / |            |  8   |      .        |            |
. |       |      / +|____________| RxD+ |________       |            |
. |   RxD |_____/   |            |______|      . |      |            |
. |       |     \   |            |  5   |      . |      |            |
. |       |      \ -|____________| RxD- |________|______| TxD pin 2  |
. |       |   |\  \ |            |______|      . |      |            |
. |       |   | \  \|            |  6   |      . |      |            |
. |       |   |  \_______________| TxD+ |________|_ NC  |            |
. |   TxD |___|   \              |______|      . |      |            |
. |       |   |   /              |  3   |      . |      |            |
. |       |   |  /O______________| TxD- |________|______| RxD pin 3  |
. |       |   | /  /|            |______|      . |      |            |
. |       |   |/  / |            |  4   |      . |      |            |
. |       |      / +|_______o____| GND  |________o______| GND pin 7  |
. |  /CTS |__o__/   |       |    |______|      .        |            |
. |       |  |  \   |       |    |  2   |      .        |            |
. | /TRxC |__|   \ -|_______|____| HSKi |________________| DTR pin 20 |
. |       |   |\  \ |       |    |______|      .        |            |
. |       |   | \  \|       |    |  1   |      .        |            |
. |       |   |  \__________|____| HSKo |________________| DSR pin 6  |
. |  /DTR |___|   \         |    |______|      .        |            |
. |       |   |   /         |    |  7   |      .        |            |
. |       |   |  /O      ___|____| GPi  |____________o__| RTS pin 4  |
. |       |   | /  /|   |   |    |______|      .     |  |            |
. |       |   |/  / |   |   |                  .     |  |            |
. |       |      / +|___|___o                  .     |__| CTS pin 5  |
. |  /DCD |_____/   |   |   |                  .        |            |
. |       |     \   |   |   |                  .        |            |
. |       |      \ -|___|   |                  .        |            |
. |       |       \ |     signal               .        |            |
. |_______|        \|     ground               .        |____________|
................................................
```

Figure 7-x1-Apple IIgs Connection to an RS-232-C DTE Device

```
................................................
.    8530   26LS32 Receivers  IIGS Mini 8-pin  .      RS-232-C DCE Device
.    SCC    & 26LS30 Drivers  Serial Connector .        DB-25 Connector
.  _______                                     .         ____________
. |       |        /|             ______       .        |            |
. |       |       / |            |  8   |      .        |            |
. |       |      / +|____________| RxD+ |_________      |            |
. |  RxD  |_____/   |            |______|      . |      |            |
. |       |     \   |            |  5   |      . |      |            |
. |       |      \ -|____________| RxD- |________|______| RxD pin 3  |
. |       |   |\  \ |            |______|      . |      |            |
. |       |   | \  \|            |  6   |      . |      |            |
. |       |   |  \_______________| TxD+ |________|_ NC  |            |
. |  TxD  |___|   \              |______|      . |      |            |
. |       |   |   /              |  3   |      . |      |            |
. |       |   |  /O______________| TxD- |________|______| TxD pin 2  |
. |       |   | /  /|            |______|      . |      |            |
. |       |   |/  / |            |  4   |      . |      |            |
. |       |      / +|_______o____| GND  |________o______| GND pin 7  |
. |  /CTS |__o__/   |       |    |______|      .        |            |
. |       |  |  \   |       |    |  2   |      .        |            |
. | /TRxC |__|   \ -|_______|____| HSKi |_______________| DSR pin 6  |
. |       |   |\  \ |       |    |______|      .        |            |
. |       |   | \  \|       |    |  1   |      .        |            |
. |       |   |  \__________|____| HSKo |_______________| DTR pin 20 |
. |  /DTR |___|   \         |    |______|      .        |            |
. |       |   |   /         |    |  7   |      .        |            |
. |       |   |  /O      ___|____| GPi  |_______________| DCD pin 8  |
. |       |   | /  /|   |   |    |______|      .        |            |
. |       |   |/  / |___|   |                  .        |            |
. |       |      / +|___|___o                  .        |            |
. |  /DCD |_____/   |   |   |                  .        |            |
. |       |     \   |   |   |                  .        |            |
. |       |      \ -|___|   |                  .        |            |
. |       |       \ |     signal               .        |            |
. |_______|        \|     ground               .        |____________|
................................................
```

Figure 7-x2-Apple IIgs Connection to an RS-232-C DCE Device

The serial inputs and outputs of the SCC are connected to the external connectors through differential line drivers (26LS30) and receivers (26LS32). The output line drivers are tri-state devices and can be put in the high-impedance mode between transmissions to allow other devices (i.e., AppleTalk devices) to transmit over those lines. A line driver is activated by lowering the SCC's Request To Send (/RTS) output for that port.

The Handshake Output signal (HSKo, pin 1) for each Apple IIgs serial port originates at the SCC's /DTR output for that port and is driven uninverted by an RS-422 line driver (26LS30). Each port's Handshake Input signal (HSKi, pin 2) is received and inverted through a differential receiver (26LS32). The output of the differential receiver is connected to the SCC's Clear To Send (/CTS) and Transmit/Receive Clock (/TRxC) inputs for that port. HSKi is designed to accept an external device's Data Terminal Ready (DTR) handshake signal through the /CTS input. The /CTS input to the SCC can be polled by software or can be used to generate an interrupt. The

HSKi line is connected to the SCC's Transmit/Receive Clock (/TRxC) input for that port, so that an external device can perform high-speed synchronous data exchange. Note that you can't use the HSKi line for receiving DTR if you're using it to receive a high-speed data clock.

Each port's General-Purpose input (GPi, pin 7) is received and inverted through a differential receiver (26LS32). The output of the differential receiver is connected to the SCC's Data Carrier Detect (/DCD) input for that port. This input can be used to provide a handshake signal from an external device to the computer. The /DCD input to the SCC can be polled by software or can be used to generate an interrupt.

Note: Because a 26LS32 differential receiver is used for the external handshake or clock signals to the SCC, the signals must be bipolar, alternating between a positive voltage and a negative voltage with respect to the internally grounded input. If a device uses ground (0 volts) as one of its handshake logic levels, the receiver interprets that level as an indeterminate state, with unpredictable results.

The SCC's Receive/Transmit Clock (/RTxC) inputs for both ports are driven by a single crystal oscillator circuit. This is accomplished by connecting a 3.6864 MHz crystal between the /RTxC and Synchronization (/SYNC) input of port A. Port B's /RTxC pin is connected to port A's /SYNC pin to drive port B's clocks from port A's oscillator circuit. Because of this single circuit, Write Register 11 (WR11) bit 7 must be set to 1 for SCC port A and must be set to 0 for SCC port B. The SCC itself is clocked at 3.58 MHz by the Apple IIgs' Color-Reference clock (CREF) at the SCC's PCLK clock input. The maximum asynchronous transmission rate supported by the SCC is 57,600 bits per second (bps) in X16 clock mode (WR4=01xxxxxx).

The SCC's Interrupt Enable In (IEI) and Interrupt Acknowledge (/INTACK) inputs are both tied to logical high in the Apple IIgs. Keeping the SCC's IEI input high enables the SCC to always generate interrupts if interrupt modes are enabled through software. Keeping the SCC's /INTACK input high leaves the SCC in Interrupt Without Acknowledge interrupt mode.

Chapter 8: I/O Expansion Slots

First Edition--Page 167, Direct memory access

DMA bank register location is $C037.

Further Reference:

---

o Apple IIgs Hardware Reference, both editions

### 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:                    MPW IIGS Assembler version:
        MACRO                                        MACRO
&lab        ERRORDEATH &text                         ErrorDeath &text
&lab        bcc end&syscnt                           bcc @EDeathEnd
            pha                                      pha
            pea x&syscnt|-16                         pea @Message>>16
            pea x&syscnt                             pea @Message
            ldx #$1503                               ldx #$1503
            jsl $E10000                              jsl $E10000
x&syscnt    dc i1'end&syscnt-x&syscnt-1'  @Message   dc.B @EDeathEnd-@Message-1
            dc c"&text"                              dc.B  &text
            dc i1'13',i1'13'                         dc.B  13
            dc c'Error was $'                        dc.B  'Error Was $'
end&syscnt  anop                          @EDeathEnd
        MEND                                         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: Dave "Evad Snoyl" Lyons, Keith Rollin,
Steven Glass, Matt Deatherage & Eric Soldan — January 1991
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 1990: Added a Note on custom bottleneck procedures and updated information on ShieldCursor and UnShieldCursor.

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 bottleneckprocedures of QuickDraw II. You may modify this record by replacing those addresseswith 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.)

Note: A custom bottleneck procedure must not begin at the first byte of a segment. If it does, then the segment could load at the beginning of a bank, and the address minus one would be in the wrong bank and RTL would transfer control to the wrong location. (See Apple IIgs Technical Note #90, 65816 Tips and Pitfalls.)

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 dispatcherand 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 simpletrick to access them. For example, to access TheFillPat ($10E) without changingthe 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 leavethe other conic pointers alone (as shown in the code fragment above), QuickDrawII will not do all of the normal setups when calling the standard conicroutines. To deal with this bug of QuickDraw II, you must patch out the additional bottleneck procedures and set up those direct pages locations yourself, orthe 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 movementmay 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 or $012000, 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-screenGrafPort in this case, even though you switched to a GrafPort on the screen. Therefore, when you call ShieldCursor, you have to make sure that thesevalues 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. Save the existing values of MinRect, ImageRef, and BoundsRect.
3. Set MinRect, ImageRef, and BoundsRect.
4. Let QuickDraw II know you've changed the contents of its direct page by clearing the "dirty" flags bits 14 to 0:

```
    DirtyFlags      equ     $EC

                    ldx     #$200           ;index to QD's third page of work
    lda     DirtyFlags,x    ;space
                    and     #$8000
                    sta     DirtyFlags,x
```

5. JSL to ShieldCursor.
6. Restore the previous values of MinRect, ImageRef, and BoundsRect.

Note: Saving and restoring these values was not previously mentioned in this Note and in most circumstances it is not necessary. Saving and restoring is now recommended. In particular, if ShieldCursor is called inside a QuickDraw II bottleneck procedure, the system can crash if you fail to restore the contents of direct page.

Calling UnshieldCursor:
1. Set direct page for QuickDraw II.
2. JSL to 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 September 1990
Written by: Dan Hitchens, Matt Deatherage & Suki Lee 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 use this Note 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 refer to these for a complete picture.
Changed since March 1990: Added corrections and further descriptions.

Printing Modes

There are two printing modes: immediate and deferred.

o In immediate mode, pages are printed as they are drawn into the printing grafPort. As the application makes QuickDraw II calls, the printer driver immediately generates commands, transferring ink to page when the page is closed. This is the fastest form of printing, but only produces high-quality images on printers that can translate QuickDraw II commands to other graphic commands. For example, the LaserWriter driver translates the QuickDraw II calls into PostScript(R) calls which can produce high-quality images.

o In deferred mode (sometimes referred to as spool mode), pages are captured to memory or disk and printed after all pages have been defined. Most printer drivers use deferred mode to create high-quality images. Since most drivers cannot obtain enough memory to image an entire page at once, they redraw page in several pieces, or bands. The printer driver creates a grafPort whose boundsRect, portRect, clipRgn, and visRgn correspond to the band and plays the picture back, thus causing the saved commands to draw only the images which fall within the band. Once the pixel image for the band is created, the printer driver converts the image to printer codes and sends the codes to the printer through the port driver.

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 pixel map
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
PrGetPgOrientation	$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 PrGetPgOrientation, it must also support PrGetPrinterSpecs).
o The Print Manager places an index to the correct routine in the X register (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

```
StartOfNewDriver    START

                    dc i2'0'                          ; new style driver
                    dc i2'(ListEnd-PrDriverList)/4'    ; count

                    jmp (PrDriverList,x)


PrDriverList        dc a4'PrDefault'
                    dc a4'PrValidate'
                    dc a4'PrStlDialog'
                    dc a4'PrJobDialog'
                    dc a4'PrDriverVer'
                    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'PrPixelMap'
                    dc a4'PrGetPrinterSpecs'
                    dc a4'PrGetPgOrientation'
ListEnd             anop
```

In previous versions of this Note, the PrPixelMap and PrDriverVer entries were reversed.

Note that when using the above technique, you're using a 16-bit jump into a table of 24-bit addresses. If all your entry points are in the same segment, this is not a problem.

If your routines' entry points are not all in the same segment, you need a dispatching routine like the following:

```
StartOfNewDriver    START

                    dc i2 '0'                       ; new style driver
                    dc i2 '(ListEnd-PrDriverList)/4'    ; count

                    lda PrDriverList+2,x
                    sep #$20
                    pha                             ; push high byte of
;                                                     address
                    rep #$20
                    lda PrDriverList,x
                    dec a                           ; decrement low 2
;                                                     bytes only
                    pha                             ; push modified low
;                                                     word of address
                    rtl                             ; transfer to the
;                                                     routine
```

See Apple IIgs Technical Note #90, 65816 Tips and Pitfalls, for a discussion of dispatching with RTL.

Print Driver Exit

When one of your routines is ready to exit, it needs to remove the input parameters from the stack, leaving the result space (if any) and the two RTL addresses. Set the accumulator and the carry flag to reflect any error you are returning, then perform an RTL.

Example

If there are N bytes of input parameters to remove, use something like the following. This code assumes that the error code is in the accumulator.

```
                    tay                             ; keep error code in Y
;                                                     temporarily
                    lda 5,s
                    sta N+5,s
                    lda 3,s
                    sta N+3,s
                    lda 1,s
                    sta N+1,s
                    tsc
                    clc
                    adc #N
                    tcs
                    tya                             ; get error code
                    cmp #1                          ; set carry if error
;                                                     is not zero
                    rtl
```

Figure 1 diagrams the stack just before exiting 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 are 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 are 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 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 Volume 3 of the Toolbox Reference. The Print Manager returns 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.

For alternate driver identification, Developer Technical Support assigns new iDev values if you feel it is absolutely necessary for your driver. Please keep in mind, however, that no application knows how to interpret style records for non-standard iDev values, and that Apple does not publish such interpretations.

Print Driver Calls

Your printer driver handles the following 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.

Note: When calling other routines in your printer driver (like PrValidate), be sure to do so through the Tool Dispatcher ($E10000 or $E10004) so any necessary patches have an opportunity to execute.

```
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.

Note: When calling other routines in your printer driver (like PrValidate), be sure to do so through the Tool Dispatcher ($E10000 or $E10004) so any necessary patches have an opportunity to execute.

```
PrPixelMap                           ($0D13)

Description:
    Prints all or part of the specified pixel map.

Passed:
    srcLocPtr                        LONG    Pointer to the source LocInfo
                                             which contains the pointer to
                                             the pixel map.
    srcRectPtr                       LONG    Pointer to the rectangle which
                                             encloses the pixel map 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 grafPort.
- 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).

You can choose to print the pixel map in any convenient fashion; one convenient way is to allocate a new print record and call your normal printing routines. This method is outlined below.

- o Gets a new handle for a print record and set it to the defaults by calling PrDefault.
- o If colorFlag is set, change the style subrecord of the print record to reflect color printing.
- 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 pixel map'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 (or restore the previous cursor using SetCursor).
- o Restore the grafPort 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 needs 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 recording pages. You can pick an arbitrary number to start with (like 20). This assumes spooling to memory; spooling to disk will obviously be different.
- o Do other initialization that might be needed to start a print job.

Possible errors:

portNotOn	$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 recording of the document 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 to close the printing grafPort.
- o If the driver allocated a grafPort in PrOpenDoc, disposes of it.
- o If in immediate mode, does what is needed to shut things down.
- o Takes down the information dialog box from PrOpenDoc.

Possible errors:

portNotOn	$1302	Indicates Port Not On
prBozo	$13FF	Someone unloaded the driver in the middle of the print loop

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 occurs.
- 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 causes QuickDraw II calls to call your routines instead of drawing to the pixel map associated with 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.

Possible errors:
    portNotOn                        $1302    Indicates Port Not On
    prBozo                           $13FF    Someone unloaded the driver in
                                              the middle of the print loop

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  Restore the grafPort to the port saved in PrOpenPage.
o  Do a ClosePicture to close the picture.

Possible errors:
    portNotOn                        $1302    Indicates Port Not On
    prBozo                           $13FF    Someone unloaded the driver in
                                              the middle of the print loop

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 all the recorded
      page images.
   o  Put up an information dialog indicating that printing is occurring.
   o  Display a watch cursor (saving the current cursor first if you like).
   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.  This is to simplify your
      code; if the StatusRecPtr is NIL, you can reasonably choose not to use a
      status record at all, but this requires an extra code path.
   o  Initialize the status record and the number of copies counter.
   o  If the idle procedure pointer in the print record is NIL, point to an
      internal one.  Again, as with the StatusRecPtr, you can choose to ignore
      idle procedures if no pointer is provided, but this requires an extra
      code path.
   o  Do The Following For Each Copy:
      o  Calculate the number of bands 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 procedure 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 a vertical 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  Restore the cursor.
```

Possible errors:

portNotOn	$1302	Indicates Port Not On

```
    prBozo                          $13FF    Someone unloaded the driver in
                                             the middle of the print loop

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
    prBozo                          $13FF    Someone unloaded the driver in
                                             the middle of the print loop

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)
(also known as PrChanged)

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 work over AppleTalk.  The name
   passed as the parameter to PrDevPrChanged should be what AppleTalk uses in
   an NBPLookup situation; for AppleTalk, such a name should follow Name
   Binding Protocol conventions.

   This routine will be called by the Print Manager when your driver is first
   loaded so a network port driver can find devices of your type.
   Applications should not make this call.  When this routine will be called
   is not guaranteed; you can't use this as a substitute for a startup call.

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.
```

The value returned for PrinterType should be the driver's iDev value.

PrGetPgOrientation ($3813)

Description:
    Returns the page orientation from a print record.

Passed:

Wordspace	WORD	Space for result
PrintRecordHandle	LONG	Handle to the print record

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 cause printing when you make QuickDraw II calls (as noted in PrOpenPage). This section describes the structure and parameters for these routines.

The basic idea is that your driver replaces low-level QuickDraw II routines with pointers to your own routines. For example, when someone wants QuickDraw II to draw some text (say with DrawString), QuickDraw II calls your low-level routine to draw the text. You can then print the text instead.

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 replacement procedures into it and call SetGrafProcs. Installing your procedure addresses causes the appropriate QuickDraw II calls to call your procedures, which, in turn, perform the actual printing.

The routines that need to be written are known as QuickDraw II "bottleneck procedures." For most dot-matrix printer drivers, the one of most concern when writing immediate mode procedures is StdText. If your target device has an alternate page imaging language, you may wish to print entirely in immediate mode. In this case, you want to intercept most of the bottleneck procedures. Apple IIgs Technical Note #34, Low-Level QuickDraw II Routines, contains information on how to install these procedures. The sample code which follows shows how to replace StdPixels and StdText.

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 Pixel maps)
;
;-------------------------------------------------------------------
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

                pushlong #PixelLoc              ;push pointer to LocInfo
                pushlong #PixelRect             ;push pointer to rectangle


;++++++++++++++++++++++
; Insert code here to print a pixel map
;    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                             ;return with 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.

               pushlong #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 1-3
- Apple IIgs Technical Note #36, Port Driver Specifications
- Apple IIgs Technical Note #90, 65816 Tips and Pitfalls

PostScript is a registered trademark of Adobe Systems Incorporated.

### 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

- o Apple IIGS Toolbox Reference, Volumes 1 & 2
- o 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: C.K. Haun   September 1990
Written by: Keith Rollin, Dave Lyons & Eric Soldan   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 March 1990: Changed input parameter definition for myFilterProc from long pointer to word pointer.

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 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.

Note that only one list control is allowed in a modal dialog. If you need more than one, the Dialog Manager cannot help you--create a normal window instead.

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.

   Note: After you add the list control, you must not add any more dialog items.

   ```
   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 is in effect) when you call ModalDialog.

```
    pascal Word myFilterProc(theDialog, theEvent, theItem)
        GrafPortPtr    theDialog;
        EventRecord    *theEvent;
        word           *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;
       word           *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

- 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: Matt Deatherage & Keith Rollin November 1990
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.
Changes since November 1988: Added information about the font family numbering convention used by those who assign 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 thescreen fonts New York, Geneva, and Monaco respectively.

Prior to System Software 3.2, all non-LaserWriter fonts (except New York, Geneva, and Shaston) were converted to Courier when printing. With System Software 3.2 and later, the LaserWriter driver prints 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. With System Software 5.0 and later, the LaserWriter driver uses fonts previously downloaded, although it does not download PostScript fonts itself.

Font Family Number Conventions

By convention, font family numbers that have the high bit set are designed for the 5:12 aspect ratio of the Apple IIgs computer. Font family numbers with the high bit clear are designed for computers with a 1:1 pixel aspect ratio, such as the Macintosh. Fonts designed for a 1:1 pixel aspect ratio appear "tall and skinny" when displayed on an Apple IIgs.

Some third-party font packages were released before this convention was defined; therefore, font family numbers between 1000 and 1200 (decimal) do not adhere to this convention.

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 tothe IIGS, the font family number is likely to be arbitrary, as is the font family number of any user-created fonts.

Further Reference

- 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

Revised by: Matt Deatherage March 1991
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.

Changes since November 1988: This Note is now obsolete.

This Note formerly described how CalcMenuSize behaves when menu widths and heights are stored as $0000 or $FFFF. This behavior is now documented in Volume 3 of the Apple IIgs Toolbox Reference on page 37-3.

Further Reference

- o Apple IIgs Toolbox Reference, Volume 3

### 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

Revised by: Matt Deatherage March 1991
Written by: Keith Rollin November 1988

This Technical Note corrects an error in the record used for GetPenState and SetPenState.

Changes since November 1988: This note is now obsolete.

This Note formerly described an error in the pen state record in Volume 1 of the Apple IIgs Toolbox Reference. This error is corrected on page 43-2 of Volume 3 of the Toolbox Reference.

Further Reference

- Apple IIgs Toolbox Reference, Volume 3

### 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

o 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

- o 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.
- o 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.
- o 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.
- o 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

- o 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 July 1989: The information on AlertWindow formerly found in this Note has been updated and is now included in the Apple IIgs Toolbox Reference, Volume 3.

The information on AlertWindow formerly found in this Note has been updated and is now included in the Apple IIgs Toolbox Reference, Volume 3. This Window Manager call was first introduced in System Software 3.2.

```
### 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

o   Apple IIGS Firmware Reference
o   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: How to Avoid Running Out of Memory

Revised by: Dave Lyons May 1992
Written by: Eric Soldan January 1989

This Technical Note discusses handling nearly-out-of-memory situations when working with the IIgs tools.

CHANGES SINCE SEPTEMBER 1990: Added discussion of an Out-of-memory routine problem fixed in System 6.0.

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.

Low-level toolbox functions (for example, QuickDraw II calls) require that a 16K block of memory be allocatable, while high-level routines (for example, the Window Manager) require that a 32K block of memory be allocatable. Apple does not guarantee that toolbox functions behave reasonably if there is less memory available, and the tools are not stress-tested with less than the minimum required memory available.

Since the toolbox assumes reasonable memory-allocation requests succeed, just waiting for an out-of-memory error is not adequate memory management. To make your application work reliably in low-memory situations, you need a method of ensuring that the toolbox gets memory when it needs it. This Note describes two approaches.

HOW MUCH MEMORY CAN BE ALLOCATED

There's no way to tell how much memory can be allocated without actually trying to allocate it.

MaxBlock tells you the size of the largest single free block, but this doesn't take into account purgeable blocks, compaction, and out-of-memory routines (see Apple IIgs Toolbox Reference, volume 3). FreeMem and RealFreeMem cannot tell you how badly fragmented the memory is, and they do not take into account out-of-memory routines.

A SUGGESTED METHOD

A method of checking for a nearly-out-of-memory condition is to have your own purgeable handle just for this task. If the handle has not been purged, then you have plenty of memory for the toolbox, and in the worst case, the toolbox purges your handle if it needs the RAM.

The less often your purgeable handle gets purged, the better performance you get in nearly-out-of-memory situations. Therefore, you should arrange for other purgeable memory, not necessarily belonging to your application, to be purged before your handle. For example, you want dormant applications to be purged, rather than having your handle get repeatedly purged and reallocated. So the purge level of this handle should be one.

The check to see if a handle has been purged is very fast. If it has been purged, you have to try to reallocate it. Reallocating a handle is not a fast process, so the fewer times the handle is purged, the faster the check is and the better your performance. Unless you are in a nearly-out-of-memory situation, the handle should not be purged at all, and you should have virtually no overhead for this process.

This technique can 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, keeping in mind that high-level toolbox routines require a 32K block.

AN ALTERNATIVE

For better control over when your handle is purged or disposed, you can write an out-of-memory routine as described in the Memory Manager chapter of Apple IIgs Toolbox Reference, volume 3. Out-of-memory routines have the opportunity to free up memory before or after the Memory Manager attempts to purge purgeable handles, and this manual contains a sample of such a routine.

NOTE : If your Out-of-memory routine frees up memory on the second pass, there is a problem with the Memory Manager in System Software 5.0 through 5.0.4 that may affect you. If your routine frees enough bytes on the second pass, but the Memory Manager still cannot complete the request it is working on, it can hang for a couple of minutes and then crash. This is fixed in System 6.0.

Further Reference:

- o Apple IIgs Toolbox Reference, Volumes 1-3

### 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: Dave Lyons May 1992
Written by: Eric Soldan January 1989

This Technical Note discusses strategies for preventing applications from loading into special memory.

CHANGES SINCE JULY 1989: Noted that the System 6 Loader always tries non-special memory before special memory.

The System 6.0 Loader always tries to load segments into non-special memory before allowing them to load into special memory. The rest of this Note is useful if your application does not require System 6, or if you need an example of an initialization segment.

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 (in System Software earlier than 6.0) 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 and 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 userI
               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 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:

- o GS/OS Reference, Volume I
- o MPW IIgs Tools Reference
- o 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

Revised by: Dave "Out of Phase" Lyons May 1992
Written by: Matt Deatherage, Jim Mensch, & Dave Lyons March 1989

This Technical Note discusses compatibility issues that can arise between desk accessories and applications. Where possible, it presents solutions.

CHANGES SINCE MARCH 1991: Updated information about QuickDraw Auxiliary and StartUpTools for System 6.0.

This Note presents guidelines to help applications and desk accessories work together smoothly.

TOOL SETS

The greatest conflict between applications and desk accessories, especially NDAs, is the use of system tool sets. The Apple IIgs Toolbox Reference, Volume 1, defines the minimum collection of tools sets available to an NDA. The Desk Manager requires that an application start the following tool sets before calling DeskStartUp:

- Tool Locator (#1)
- Memory Manager (#2)
- Miscellaneous Tools (#3)
- QuickDraw II (#4)
- Event Manager (#6)
- Window Manager (#14)
- Menu Manager (#15)
- Control Manager (#16)
- LineEdit (#20)
- Dialog Manager (#21)
- Scrap Manager (#22)

NDAs may assume that these tools are all present and running, so they do not need to check for their presence. NDAs can also use the following tool sets without special consideration for starting them up: Desk Manager, Scheduler, Apple Desktop Bus, and Integer Math.

In addition to the tool sets applications must start to support NDAs, Apple recommends that applications start the following tools:

QuickDraw Auxiliary (#18)	(see discussion under QuickDraw Auxiliary)
Font Manager (#27)	

These two additional tools are so widely used by desk accessories that they should be present. NDAs may not assume their presence, but it is reasonable to write an NDA that checks for them, with the assumption that they usually turn out to be available.

NDA GUIDELINES

WHICH TOOL SETS CAN AN NDA USE?

- In general, NDAs can use the tool sets which have already been started up by the host application, even tools that are not guaranteed to be started up. Using other tool sets is trickier (see below).

- In general, NDAs should not start up tools that are already started up. (The Resource Manager is an exception.)

- The Resource Manager must be started separately by each client. See Apple IIgs Technical Note #71 for detailed information on using the Resource Manager from an NDA.

- Sound tools are an exception to the rule of freely using a tool which is already started. See the section "Sound Tools" sections later in this note.

- Some tool sets are easily started up each time they are needed, if they are not already present.

  Standard File is an excellent example. If an NDA needs to use 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 allocate a page of direct-page space and start the tool before using it. When finished with the tool, the NDA must shut it down, deallocate the direct-page space, and unload it with UnloadOneTool. (A tool is already running if its xxxStatus function returns TRUE and does not return an error.)

  The important thing here is that the NDA shuts down Standard File imediately after using it, if it was not already started. This does not cause conflicts with the host application or with other NDAs.

  Note that by pre-initializing the result space of an xxxStatus call to zero, you can avoid caring whether the tool is present but not started or simply not present.

  ```
  pea $0000
  _SFStatus
  pla                  ;A is nonzero if Standard File is started
  ```

  From a high-level language, you may not be able to pre-initialize the result space. Instead, you need something like the C statement:

  ```
  StdFileActive = ( SFStatus() && !_toolErr);
  ```

  or the Pascal statement:

```
StdFileActive := (SFStatus<>0) AND (ToolErrorNum=0);
```

o It is impractical or impossible to start up certain tool sets each time they are needed. These include the Font Manager, Scrap Manager, and Text Edit.

  If an NDA needs to start up a tool and keep it started while letting the application continue to run, things get interesting. (There is a risk that the host application will later try to start up the tool set itself and not be able to deal with the tool already being started.)

  In practice, the safest thing you can do for a tool you need to leave running is:

  --When your NDA is opened, check the tool set's status. If it is not available, use LoadOneTool, allocate any needed direct-page space, start up the tool set, and set a flag indicating that your NDA started the tool set.

  --When your NDA's Init routine is called at DeskShutDown time (Accumulator equal to zero), check the flag set above. If your NDA started a tool set, shut it down, dispose of any direct-page space you allocated for it, and call UnloadOneTool.

  (Keep in mind that your NDA can be opened and closed many times before DeskShutDown is called when the application finally quits. If you have started a tool and set a flag on an open, be sure not to disturb the flag on a future open, when the tool is already available because you started it! You still need to shut it down at DeskShutDown time.)

  --Do not shut down tool sets when your NDA is closed. To see why, consider what would happen if two NDAs just like yours were used at the same time. If the NDAs were closed in any other than the exact opposite order they were opened, some NDAs would have tool sets shut down from underneath them.

STARTUPTOOLS

o StartUpTools in System Software 5.0.4 and earlier is designed to be called only by an application, not a desk accessory. Unexpected things happen if your NDA calls StartUpTools (for example, you may get a second copy of the application's resource fork open in your NDA's private resource search path; this wastes RAM and can interfere with an application's attempt to write to its own resource fork).

o See the System 6.0 Toolbox documentation for information on using StartUpTools from an NDA. There are new flag bits you need to know about.

TLSTARTUP AND TLSHUTDOWN

o Do not call TLStartUp or TLShutDown from a desk accessory.

o You may call MMStartUp at any time to get your desk accessory's own memory ID. This does not allocate a new ID; it just tells you what ID you already have (it returns the memory ID of the block the MMStartUp call is made from).

USER TOOL SETS BELONG TO THE APPLICATION

o A desk accessory must not install user tool sets, because there is no arbitration of user tool set numbers. User tool sets are the sole property of the current application.

  A desk accessory should not call user tool sets even if it determines that the host application has installed a certain tool set, because that limits future system software options. For example, consider a hypothetical multiple-application environment. If DAs call user tool sets and the system automatically switches between separate collections of user tool sets, there would be no way for the system to know which set to switch in before giving control to a desk accessory.

BANK ZERO MEMORY AND ERROR $0201

o If you get error $0201 (unable to allocate memory block) while trying to launch a ProDOS 8 application, it is probably because your NDA allocated some memory in bank 0 or bank 1 and failed to dispose of it at DeskShutDown time (when the NDA's Init routine is called with the accumulator equal to zero). GS/OS needs to allocate all of this memory for ProDOS 8 to use.

QUICKDRAW AUXILIARY

o In System 6.0 and later, QuickDraw Auxiliary is always available to an NDA, because the Window Manager automatically loads and starts QuickDraw Auxiliary (because it's needed for AlertWindow, for example). To prevent problems, duplicate QDAuxStartUp and QDAuxShutDown calls are tolerated, and QDShutDown automatically calls QDAuxShutDown.

o Before System 6.0, starting QuickDraw Auxiliary when the application has not started it can be a problem. An application that correctly implements switching between 320 and 640 mode calls QDShutDown and QDStartUp. QuickDraw Auxiliary depends heavily on QuickDraw, and restarting QuickDraw while QuickDraw Auxiliary is active will fry big-time.

SOUND TOOLS

o A desk accessory cannot use any of the sound tools if they are already started. This is contrary to the rule for other tool sets, but it is required because there is no memory management of the sound RAM (or "DOC 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 could 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 during 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.

APPLICATION GUIDELINES

For best compatibility with NDAs, applications should follow the following guidelines.

- Be careful about when your application starts and shuts down tools. A highly compatible approach is to start tools at the beginning of your application and leave them started. For certain tools, like Standard File, it is reasonable to load and start the tool set each time it's needed (you may want to check whether it's already started, in case some impolite NDA started Standard File and left it started).

  Note that UnloadOneTool followed later by LoadOneTool does not necessarily cause disk access or ask the user to insert the boot disk. UnloadOneTool calls UserShutDown to put the tool set into "zombie" state, where it can be restarted from memory if none of its segments have been purged. Unloading tools while they aren't in use is a Good Thing--if the user has plenty of RAM, there's no noticeable performance hit, but if RAM space is tight then doing extra disk access still is preferable to actually running out of memory.

  For maximum compatibility, an application should not shut down any tools which were ever active when it called SystemTask or TaskMaster (until quitting time, of course, when it shuts down everything, starting with the Desk Manager). The application can start more tools, but it should not shut down those which are already active.

  If your application is going to start a tool and not keep it started, use it and then shut it down immediately, without allowing desk accessories to be opened during that time.

- Don't just start the Scrap Manager--use it! Many desk accessories support cutting and pasting to exchange text and pictures with your application, but they can do it only if you use the Scrap Manager. If you have a need for your own private scrap internally, you should still also use the Scrap Manager to exchange text and pictures with other applications and DAs.

- Provide an Edit menu, and when an NDA window comes to the front enable the menu and the Undo, Cut, Copy, Paste, and Clear items.

- Applications should never make a Close call with reference number zero at file level zero. (If you need to use Close with reference number zero, use GetLevel and SetLevel to avoid closing files you did not open.)

  DAs written recently can open their files at an internal file level, as documented in GS/OS Technical Note #13, but applications still need to avoid closing all files at level zero for compatibility with older desk accessories.

- An application with some memory to spare can save NDAs time by providing them the additional tools which they are most likely to use.

  The most common tools which desk accessories require besides those available in the standard Desk Manager set are QuickDraw Auxiliary (#18), the Print Manager (#19), Standard File (#23), the Font Manager (#27), and the List Manager (#28).

- When you call TaskMaster or GetNextEvent, or EventAvail, be sure bit 10 is turned on in the event mask, to enable "desk accessory" events. If you turn this bit off, users will not be able to get to the Classic Desk Accessory menu by pressing Apple-Ctrl-ESC.

CDA GUIDELINES

- CDAs are nearly always modal, but by using the HeartBeat interrupt queue or other mechanisms, they can get control when the user is no longer "in" the CDA. The list of guaranteed tools for NDAs does not apply to CDAs, and CDAs must be prepared to deal with the ProDOS 8 environment as well as GS/OS.

- Under ProDOS 8, a CDA will not be able to allocate any bank 0 space through the Memory Manager; it can only use page 0 and page 1 safely (the stack is in page 1).

- Do not call TLStartUp or TLShutDown from a desk accessory.

- You may call MMStartUp at any time to get your DA's own memory ID. This does not allocate a new ID; it just tells you what ID you already have (it returns the memory ID of the block the MMStartUp call is made from).

Further Reference:

- Apple IIgs Toolbox Reference
- Programmer's Introduction to the Apple IIgs
- Apple IIgs Technical Note #71, Desk Accessory Tips and Techniques
- Apple IIgs Technical Note #83, Resource Manager Stuff

### END OF FILE TN.IIGS.053

```
######################################################################
### FILE: TN.IIGS.054
######################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs #54: MIDI Drivers

Revised by: Matt Deatherage November 1990
Written by: Jim Mensch May 1989

This Technical Note describes how to write a driver for use with the Apple IIgs MIDI tools.
Changes since May 1989: Noted that MIDI drivers also work with the MIDI Synth tool.

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 have to write your own. This Note includes all the information you need to create a MIDI driver. Note that the same drivers that work with MIDI Tools (Tool #32) also work with the MIDI Synth (Tool #35). This Note collectively refers to MIDI Tools and MIDI Synth as the "MIDI tools."

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' or MIDI Synth's 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 Eror 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.

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.

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.

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

Revised by: Matt Deatherage — November 1990
Written by: Eric Soldan — July 1989

This Technical Note discusses application difficulties when transferring control to dynamic segments during low-memory conditions.
Changes since July 1989: The information formerly covered in this Note is now discussed in greater detail in Apple IIgs Technical Note #22, Proper Use of Dynamic Segments.

This Note formerly warned of the dangers of using dynamic segments-if memory is not available for the dynamic segment, the system crashes. Apple IIgs Technical Note #22, Proper Use of Dynamic Segments, covers this problem and strategies for working around it.

### 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

Revised by: Dave "nocturnal" Lyons   December 1991
Written by: Dave Lyons   July 1989

This Technical Note discusses how you can use the Memory Manager from interrupt routines and documents a flag byte that debugging utilities can use to temporarily prevent the Memory Manager from moving or purging memory. Changes since July 1989: Expanded and retitled Note to discuss safe use of the Memory Manager at interrupt time.

The Memory Manager does not disable interrupts while it's busy. Instead, it increments the system BUSY flag when it's in the middle of something important.

Can interrupt routines call the Memory Manager?

If you write code that executes at interrupt time, you must check the BUSY flag (the byte at $E100FF) before making any Memory Manager calls. If the BUSY flag is zero, it's safe to call the Memory Manager. If the BUSY flag is nonzero, the Memory Manager may be in the middle of a call, so it is not safe to call it.

What routines must check the BUSY flag?

Classic desk accessory main routines and shutdown routines do not need to check the BUSY flag. If the Event Manager is active, the CDA gets control during GetNextEvent, not at interrupt time. If the Event Manager is not active, the CDA gets control only when the BUSY flag reaches zero.

GS/OS signal handlers do not need to check the BUSY flag, because the system dispatches signals only when the BUSY flag is zero.

Run Queue tasks do not need to check the BUSY flag before calling the Memory Manager. The system dispatches Run Queue tasks at SystemTask time--the BUSY flag may not be zero, but no Memory Manager call will be in progress.

Heartbeat interrupt tasks and other interrupt handlers do need to check the BUSY flag before calling the Memory Manager.

Interrupt-time use of moveable memory blocks

If an interrupt-time routine needs access to an unlocked, non-fixed memory block, you must check the BUSY flag. It is not sufficient to lock the block, use it, and then unlock it (even if you twiddle the handle's access word directly). If the BUSY flag is non-zero, the Memory Manager could be in the middle of compacting memory, which means your block could be "in transit" from one address to another (some bytes copied, some not).

To use already-allocated memory at interrupt time, either keep the block locked or fixed, or check that the BUSY flag is zero before using the memory at interrupt time.

What if BUSY is nonzero?

If the BUSY flag is nonzero, you may want to (depending on your application) exit the interrupt routine and hope the BUSY flag is zero the next time, or call SchAddTask in the Scheduler to make the system call your routine when the BUSY flag next returns to zero. Keep in mind, though, that only four scheduled tasks can be pending at a time.

Interrupt-time flag byte

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.

It is not useful to check the value of the $E100CB flag. It is always set during interrupt handling whether any non-reentrant system component is busy or not.

### 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

```
Revised by:    Dave Lyons                                January 1991
Written by:    Dave Lyons                                   July 1989
```

This Technical Note described a problem with zero-length text scraps.
Changed since July 1989: This Note is obsolete beginning with System Software 5.0.3. There is no longer a problem with creating a text scrap of length zero.

In System Software 5.0.3 and later, LEFromScrap no longer trashes memory if you create a text scrap (scrap type 0) with length zero.

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:   Menu Manager Memorabilia

Revised by: Matt Deatherage, Dave Lyons, & Tim Swihart   November 1990
Written by: Dave Lyons   July 1989

This Technical Note discusses the Menu Manager, specifically a few anomalies and some tips for making menus your friends.
Changes since May 1990: Noted that System Software 5.0.3 fixes a bug in NewMenuBar2.

The Menu Manager Is Your Friend

In general, this is the truth. You can do all kinds of nifty things with menus, especially in System Software 5.0 and later. However, there are a few things you should know unless you generally are fond of pain in your life.

Disabling Menus Gracefully

As documented, SetMenuFlag can be used to disable and enable entire menus. When a menu is disabled, the menu title and all items within the menu are disabled. You may pull down a disabled menu, but you may not select any item within it (unless the routine MenuGlobal has been used to allow inactive menu items to be selected).

Volume 1 of the Apple IIgs Toolbox Reference says you should call DrawMenuBar if you change the appearance of a menu title with SetMenuFlag. You can do this; this is fine. It may, however, induce dizziness if used often.

A more graceful way to dim menus is to follow SetMenuFlag with HiliteMenu. Calling HiliteMenu causes the menu title to be redrawn to reflect the current (or new) highlighting and menu flags. Using HiliteMenu instead of DrawMenuBar allows you to disable and enable menus gracefully, without noticeable flicker or threat of nasty patent infringement lawsuits from strobe light manufacturers.

"System" Bars Versus "Window" Bars

As far as the Menu Manager is concerned, there are only two kinds of menu bars. One kind is in a window and the other kind is not. The former are called "window" menu bars and the latter are generally referred to as "system" menu bars.

Most people think of the System bar as the big menu bar across the top of the screen. This is encouraged by calls like SetSysBar, which takes a menu bar handle and sets the menu bar across the top of the screen to that menu bar. Trying to rename one or the other of these two concepts at this point is probably useless; instead, this Note refers to the bar across the top of the screen as the "System" bar (with a capital S), and menu bars not in windows as "system" bars (with a lowercase s).

When you start the Menu Manager, it creates an empty System bar for you. Before System Software 5.0, most people simply called NewMenu and InsertMenu to insert menus into that System bar. All was well in the world.

When 5.0 was released, it became very easy to create a new menu bar and all the menus within it using the NewMenuBar2 call. This avoids a lot of code, and many new people use it. The problem comes with DrawMenuBar. If you simply call NewMenuBar2 to obtain your menu bar and menus from resources, then call DrawMenuBar to make them visible, you usually get an empty menu bar. Why? The windowPtr parameter passed to NewMenuBar2 determines whether or not the new menu bar created is a system bar or a window bar-it does not force the new bar to be the System (note the capital 'S') bar. So when DrawMenuBar draws the current System bar, it hasn't changed from the empty default one created by MenuStartUp.

This is why Volume 3 of Apple IIgs Toolbox Reference recommends code similar to the following:

```
       menuHandle := NewMenuBar2(refDesc,menuBarTRef,NIL);
       SetSysBar(menuHandle);
       SetMenuBar(NIL);       {NIL makes the System bar the current menu bar}
```

if you want your menu bar to be the one across the top of the screen.

A Bug in NewMenuBar2

NewMenuBar2 is a handy thing to have around, but it does have a problem in 5.0.2 and earlier. When the Menu Manager is done with resources, it tries to use the internal toolbox call CMReleaseResource to free them in memory. However, it passes the wrong resource ID, and CMReleaseResource calls SysFailMgr if it encounters any errors at all (such as Specified resource not found).

What NewMenuBar2 does improperly is push the high word of the resource ID onto the stack twice, instead of the high word followed by the low word. Because of the way the Resource Manager operates, CMReleaseResource returns with no error if the ID passed is NIL, but the resource is not released (another good reason not to try to use the illegal value NIL as a resource ID).

If the high word of the menu bar resource is $0000, NewMenuBar2 passes a resource ID of NIL to CMReleaseResource, which then doesn't quite release the resource, but returns no error. The menu bar resource hangs around in memory until ResourceShutDown. It's usually fairly small, so this is no loss. It still takes up less room than menu strings, which had to stay in memory until MenuShutDown.

If the high word of the menu bar resource is not zero, the bug causes CMReleaseResource to bring down the system. When using System Software 5.0.2 or earlier, make sure all menu bar resource IDs have a high word of $0000.

System Software 5.0.3 fixes this bug.

Menu and Menu Title ID Numbers

Table 13-4 in Volume 1 of Apple IIgs Toolbox Reference gives a listing of menu and menu item ID numbers. In both lists, $0000 and $FFFF are "reserved for internal use" and noted that $0000 usually indicates the first menu in the bar (or first item in the menu) and $FFFF usually indicates the last menu in the bar (or last item in the menu). Some developers have taken this to mean that they should give their first menu an ID of $0000 and their last one an ID of $FFFF.

This assumption is incorrect.. The Menu Manager may change these values internally to reflect such IDs, but they must not be assigned that way by an application. Some applications that use IDs of $0000 or $FFFF break under System Software 5.0 and later. Note that $0000 can be used as the insertAfter parameter to InsertMenu to insert a menu at the left of a menu bar, but $FFFF is not a valid insertAfter value.

Desk Accessories and Menus

Some desk accessory developers would like to have their NDAs insert a menu in the System menu bar. While the menu itself can be inserted, the NDA cannot detect that a user has selected an item within that menu. The application gets the event and does not know what to do with it. NDAs that need a menu can put a menu bar in their own window. Since the mouseDown event then happens within the NDA's window, the NDA gets the event and can handle it normally. Be sure to make the NDA's menu bar the current menu bar before calling MenuSelect from within your NDA (to avoid possible conflicts between NDA menu item IDs and application menu item IDs). Restore the current menu bar to the application's menu bar before returning control to the application. Failure to do so prevents the application from finding its menus. Apple IIgs Technical Note #3, Window Information Bar Use documents how to put a menu in a window's information bar.

Documentation Error in MenuSelect

Volume 1 of Apple IIgs Toolbox Reference states that MenuSelect returns the menu ID and the item ID of the selected item in the when field of the event record. This is incorrect. MenuSelect actually returns the information in the wmTaskData field of the task record (and this, in fact, is why you pass a task record and not just an event record to MenuSelect).

Menu Strings and Bank Boundaries

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 do 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 noCrossBank 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.

Note that this restriction applies only to menu strings, not the menu templates that can be used with NewMenu2.

Return Values From GetMenuTitle and GetMItem

Starting with System Software 5.0, GetMenuTitle and GetMItem can return handles and resource IDs, not just pointers. The type of data returned depends on how the menu or item was created, so existing applications are not affected. For more information, see Apple IIgs Toolbox Reference, Volume 3, Chapter 37, "New Features of the Menu Manager."

Further Reference

- Apple IIgs Toolbox Reference, Volumes 1 & 3
- Apple IIgs Technical Note #3, Window Information Bar Use

### 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

- 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

o 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

Revised by: Dave Lyons May 1991
Written by: Jim Luther July 1989

This Technical Note documents master color values used for the Apple IIgs text, text background, and border colors.

Changes since July 1989: Added information on the standard QuickDraw II 640-mode color table and provided a 320-mode color table that produces similar colors.

Border Color Values

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.

Standard 640-mode Color Table

The description of dithering on pages 16-35 and 16-36 of Apple IIgs Toolbox Reference, Volume 2 is correct, but some of the color values in Table 16-5 are incorrect. Table 2 lists the standard QuickDraw II 640-mode color table:

Color Table Offset	Color Name	Master Color Value
$0	Black	$0000
$1	Red	$0F00
$2	Green	$00F0
$3	White	$0FFF
$4	Black	$0000
$5	Blue	$000F
$6	Yellow-green	$0FF0
$7	White	$0FFF
$8	Black	$0000
$9	Red	$0F00
$A	Green	$00F0
$B	White	$0FFF
$C	Black	$0000
$D	Blue	$000F
$E	Yellow-green	$0FF0
$F	White	$0FFF

Table 2-Standard 640-mode Color Table

Table 3 shows Master Color values you can use in 320-mode to get close approximations of the sixteen standard 640-mode "solid" (really dithered) 640-mode colors.

Color Table Offset	Color Name	Master Color Value
$0	Black	$0000
$1	Deep Blue	$0008
$2	Yellow-brown	$0880
$3	Gray	$0888
$4	Red	$0800
$5	Purple	$0808
$6	Orange	$0F80
$7	Pink	$0F88
$8	Dark Green	$0080
$9	Aqua	$0088
$A	Bright Green	$08F0
$B	Pale Green	$08F8
$C	Gray	$0888
$D	Periwinkle Blue	$088F
$E	Yellow	$0FF8
$F	White	$0FFF

Table 3-Standard 640-mode Color Table

Further Reference

- o Apple IIgs Hardware Reference, pp. 58, 76-78
- o Apple IIgs Toolbox Reference, Volume 2, p. 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

- o 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

- 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

Revised by: Matt Deatherage May 1992
Written by: Matt Deatherage September 1989

This Technical Note discusses the ExpressLoad feature and how it relates to the standard Loader and your application.

CHANGES SINCE SEPTEMBER 1990: Clarified some changes now that ExpressLoad and the System Loader are combined to be "Loader 4.0" in System Software 6.0. Completely removed the note about not calling Close(0) since it's not relevant.

SPEEDY THE LOADER HELPER

ExpressLoad is a GS/OS feature which is usually present with System Software 5.0 (if the ExpressLoad file is present and there's more than 512K of RAM), and always on System Software 5.0.4 and later. In fact, ExpressLoad is no longer a separate file in System Software 6.0; it's included in the System Loader version 4.0. Even though ExpressLoad is part of the Loader, we refer to its functionality separately to distinguish how the Loader takes special advantage of "expressed" files.

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 which allows the Loader to load these files more quickly, 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 LOADER COMPONENTS, 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 rest of 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 regular System Loader loads it instead. ExpressLoad also gets first shot at other loader calls.

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.

- o 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.

- o 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. This is fixed in 6.0, but is still a good thing to avoid for compatibility with System Software 5.0 through 5.0.4.

- o 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.

- o Avoid using GetLoadSegInfo before System Software 6.0. This call returns System Loader data structures which are not supported by ExpressLoad previous to 6.0. In System Software 6.0 and later, the combined Loaders return correct information for GetLoadSegInfo regardless of whether the load file is expressed or not.

- o 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.

- o Don't have segments that link to other files. ExpressLoad does not support this type of link.

Further Reference

- 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

Revised by: Matt Deatherage May 1992
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.

CHANGES SINCE NOVEMBER 1989: Corrected some typographical errors and added Carta and Sonata, two fonts the LaserWriter driver knows about but aren't built into any LaserWriter.

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 and later 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.

There are no similar resources on the Apple IIgs, which means the Apple IIgs LaserWriter driver has no way to match PostScript fonts to Apple IIgs fonts. Instead, 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) plus Carta and Sonata (two graphical fonts used in map and music programs) 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, -Italic, and -BoldItalic. 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.

All LaserWriters	LaserWriter Plus and LaserWriter II	
Courier	AvantGarde	Palatino
Carta	Bookman	Symbol
Helvetica	Courier	Times
Sonata	Helvetica	ZapfChancery
Symbol	Helvetica-Narrow	ZapfDingbats
Times	NewCenturySchlbk	

Table 2-Built-in LaserWriter Fonts

Further Reference

- o Apple IIgs Toolbox Reference, Volumes 1 & 2
- o 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

- o Apple IIGS Hardware Reference
- o Apple IIGS Firmware Reference
- o Apple IIGS Technical Note #28, Interface Card Design Guidelines
- o 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

Revised by: Matt Deatherage — May 1990
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.
Changes since September 1989: Removed the section which stated that this Note showed how to switch slots in a way that does not interfere with slot arbitration and replaced it with the proper description, which is how to search a 14-slot system for peripherals and their identification bytes.

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 search a 14-slot system for peripherals and their identification bytes.

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.IIGS.071
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#71: DA Tips and Techniques

Revised by: Dave "Mr. Tangent" Lyons May 1992
Written by: Dave Lyons November 1989

This Technical Note presents tips and techniques for writing Desk Accessories.

CHANGES SINCE DECEMBER 1991: Reworked discussion of NDAs and Command-keystrokes. Marked obsolete steps in "NDAs Can Have Resource Forks."

CLASSIC DESK ACCESSORY TIPS AND TECHNIQUES

READING THE KEYBOARD

For a CDA that runs only under GS/OS, the Console Driver is the best choice for reading from the keyboard. Other CDAs have two cases to deal with: the Event Manager may or may not be started. The Text Tools can read the keyboard in either case, but you should avoid using the Text Tools whenever possible (see Apple IIgs Technical Note #69, The Ins and Outs of Slot Arbitration).

You can call EMStatus to determine whether the Event Manager is started. When it is, you can read keypresses by calling GetNextEvent. When the Event Manager is not started, you can read keys directly from the keyboard hardware by waiting for bit 7 of location $E0C000 to turn on. When it does, the lower seven bits represent the key pressed. Once you've detected a keypress, you need to write to location $E0C010 to remove the keypress from the buffer.

Alternately, you can use IntSource (in the Miscellaneous Tools) to temporarily disable keyboard interrupts and then read the keyboard hardware directly. Be sure to reactivate keyboard interrupts if, and only if, they were previously enabled.

JUST ONE PAGE OF STACK SPACE

CDAs normally have only a single page of stack space available to them (256 bytes at $00/01xx). Your CDA may or may not be able to allocate additional stack space from bank 0 during execution. The following code (written for the MPW IIgs cross-assembler) shows a safe way to try to allocate more stack space and to switch between stacks when the space is available.

If ProDOS 8 is active, your CDA cannot allocate additional space (and there is no completely safe way to "borrow" bank 0 space from the ProDOS 8 application).

```
HowMuchStack   gequ    $1000          ;try for 4K of stack space
```

```
start           phd
                phb
                phk
                plb
                pha                    ;Space for result
                pha
                PushLong #HowMuchStack
                pha
                _MMStartUp
                pla
                ora    #$0f00          ;OR in an arbitrary auxiliary ID
                pha
                PushWord #$C001        ;fixed, locked, use specified bank
                PushLong #0            ;(specify bank 0)
                _NewHandle
                tsc
                sta    theOldStack
                bcs    NoStackSpace    ;still set from _NewHandle
                tcd
                lda    [1]
                tcd
;               clc                    ;carry is already clear
                adc    #HowMuchStack-1
NoStackSpace    pha
                ldx    #$fe
keepStack       lda    >$000100,x
                sta    stackImage,x
                dex
                dex
                bpl    keepStack
                pla
                tcs
                jsl    RealCDAentry     ;carry is clear if large stack
                                        ;available
                php
                php
                pla
                sta    pRegister
                sei
                ldx    #$fe
restoreStack    lda    stackImage,x
                sta    >$000100,x
                dex
                dex
                bpl    restoreStack
                lda    theOldStack
                tcs
                lda    pRegister
                pha
                plp
                plp
                lda    1,s
                ora    3,s
                beq    noDispose
                _DisposeHandle
                bra    Exit
noDispose       pla
                pla
```

```
Exit           plb
               pld
               rtl
pRegister     ds 2
theOldStack   ds 2
stackImage    ds.b 256
```

When this routine calls RealCDAentry, the carry flag is set if no extra stack space is available. If the carry is clear, the additional stack space was available and the direct-page register points to the bottom of that space.

```
RealCDAentry   bcs    smallStack           ;if c set, only 1 page of stack
                                           ;is available
               ...                         ; put something interesting here
               rtl

smallStack     _SysBeep
               rtl
```

Note that interrupts are disabled while the page-one stack is being restored; they are reenabled (if they were originally enabled) only after the stack pointer is safely back in page one.

INTERRUPTS, EVENT MANAGER, MEMORY, AND CDAS

Whether the Event Manager is active or not, the user hits Apple-Ctrl-Esc and usually gets to the CDA menu. It looks the same, but what happens internally is different affects what happens when your CDA allocates memory.

When the Event Manager is active (as it normally is while the user is running a Desktop application), hitting Apple-Ctrl-Esc posts a deskAcc event to the event queue. The CDA menu appears only when the application calls GetNextEvent or EventAvail with the deskAcc bit enabled in the event mask.

So with the Event Manager active, the CDA menu and individual CDAs are running in the "foreground"--no processor interrupt is being serviced, and the foreground application is stuck inside the GetNextEvent or EventAvail call. The Memory Manager knows that no interrupt is in progress, so it will happily compact and purge memory if necessary to carry out a memory allocation request from your CDA. This is just fine, since the foreground application made a toolbox call--unlocked memory blocks are not guaranteed to stay put.

When the Event Manager is not active, hitting Apple-Ctrl-Esc either enters the CDA menu immediately (if the system Busy Flag is zero) or calls SchAddTask so that the CDA menu appears during a the next DECBUSYFLG call that brings the system Busy Flag down to zero. If the CDA menu appears during a DECBUSYFLG , normal memory compaction and purging are possible, just like when the Event Manager is active.

But if the Busy Flag was zero when the user hit Apple-Ctrl-Esc, then the CDA menu appears inside of the interrupt, and the foreground application is at an unknown point where it may justifiably expect that unlocked memory blocks will not move or be purged (see Apple IIgs Toolbox Reference, Volume 1, page 12-5). (Note that the Desk Manager does a tricky dance to allow additional interrupts to occur, even though the Apple-Ctrl-Esc interrupt will not return until the user chooses Quit from the CDA menu. Normally interrupts cannot be nested; the Desk Manager and AppleTalk are exceptions.)

The Memory Manager knows an interrupt is in progress, so CompactMem takes no action and memory allocation requests do not cause unlocked memory blocks to move and do not attempt to purge purgeable blocks to make room. Memory allocation requests will still normally succeed, but you will not be able to allocate a block larger than the value returned by MaxBlock.
New Desk Accessory Tips and Techniques

AN NDA CAN FIND ITS MENU ITEM ID

After the application has called FixAppleMenu, an NDA can look at its menu item template (after the "\H" in the NDA header) to determine the menu ID corresponding to the NDA's name in the Apple menu. This is sometimes useful to pass to OpenNDA (if the NDA has some way to open itself), or to pass to a Menu Manager call.

Finding the menu item ID in the NDA's header is easy if the NDA is written in assembly. In a high-level language it may be harder (if you don't have direct access to your NDA's header, you need to find it on the fly and scan for the "\H").

NDAS AND COMMAND- KEYSTROKES

To give the user a consistent way to close NDA windows, System 6.0 handles Command-W automatically when a system window is in front. It calls CloseNDAbyWinPtr without letting the NDA or the application see the Command-W.

However, there is a special action code (optionalCloseAction) that an NDA can accept to handle the Close request itself. This way the NDA can offer the user a chance to cancel the Close, which is impossible when the system calls the NDA's main Close routine, as CloseNDAByWinPtr does. (See the System 6.0 Toolbox documentation for details.)

There is no way for an NDA to accept some keystrokes and pass others along to applications, but if your NDA does not want any keystroke events, turn off the corresponding eventMask bits in the NDA header (this allows the application to receive keystrokes while your NDA window is in front).

CALLING INSTALLNDA FROM WITHIN AN NDA

It is possible to write an NDA that installs other NDAs. However, with System Software 5.0 and later, InstallNDA returns an error when called from an NDA. When your NDA has control because the Desk Manager called one of your NDA's entry points, the Desk Manager's data structures are already in use, so InstallNDA is unable to modify them.

The solution is to use SchAddTask in the Scheduler to postpone the InstallNDA call until the system is not busy. Remember that the Bank and Direct Page registers are not defined when your scheduled task is executed.

PROCESSING MOUSEUP EVENTS

When an NDA's action routine receives a mouseUp event, it is not always safe for the NDA to draw in its window.

For example, when the user drags an NDA window, the NDA receives the mouseUp before the window is actually moved, and before DragWindow erases the outline of the new window position, which may overlap the window's content. In addition, when the user chooses a menu item, the front NDA receives the mouseUp before the menu's image is removed, and the image may overlap the NDA's window. In either case, drawing in the window makes a mess.

The solution is to avoid drawing in direct response to a mouseUp. Instead, invalidate part of the window to force an update event to happen later.
NDAs Can Have Resource Forks

Following is the recommended way for a New Desk Accessory to use its file's resource fork.

In the NDA's Open routine, do the following. Steps that are obsolete (and safely omitted) with System Software 6.0 and later are marked with an asterisk (*):

1. Call GetCurResourceApp and keep the result.
2. If the NDA does not already know its Memory Manager user ID, call MMStartUp to get it.
3. Call ResourceStartUp using the NDA's user ID.
4. Call the Loader function LGetPathname2 with the NDA's user ID (and a fileNumber of $0001) to get a pointer to the NDA's pathname. (The result is a pointer to a class-one GS/OS string.)

*5. Use GetLevel to get the current file level, then use SetLevel to set it to zero. This helps protect your resource fork from being closed accidentally.

6. Use GetSysPrefs to get the current OS preferences, then use SetSysPrefs to ensure that the user is prompted, if necessary, to insert the disk containing your resource fork. (To compute the new preferences word, take the current one, AND it with $1FFF, and ORA it with $8000. This tells GS/OS to deal with volume-not-found conditions by putting up a please-insert-disk dialog with an OK button and a Cancel button.)
7. Call OpenResourceFile using the result from LGetPathname2. Save the returned fileID--you need it when closing the file. (Be prepared to deal with an error, such as $0045, Volume Not Found.)
8. Use SetSysPrefs to restore the OS preferences saved in step six.

*9. Use SetLevel to restore the file level to its old value (saved in step five).

10. Call SetCurResourceApp with the old value saved in step one.

In the NDA's action routine, no special calls are necessary--the Desk Manager calls SetCurResourceApp automatically before calling your action routine, so your NDA's own resource search path is already in effect.

Run queue routines and NDA installs with AddToRunQ are treated the same way--the NDA's resource search path is automatically in effect when the run queue routine is called.

In the NDA's Close routine, do the following:

1. Call CloseResourceFile with the fileID that was returned when you opened it.
2. Call ResourceShutDown with no parameters.

NDAS MUST BE CAREFUL HANDLING MODAL WINDOWS

If your NDA uses its resource fork and calls TaskMaster with a restricted wmTaskMask to produce a modal window, you must be careful not to allow TaskMaster to update the contents of any application windows that happen to need updating.

The problem is that an application window's wContDraw routine can reasonably assume that the current Resource Manager search path is the application's, but TaskMaster does not take any special steps to set it. When the content-draw routine draws controls which were created from resources which are not presently in the resource search path, the system may crash.

If your NDA does not start up the Resource Manager, the Desk Manager is unable to SetCurResourceApp to your NDA, so the application's search path is still in effect--no problem. But if your NDA does start the Resource Manager, you have to be careful not to cause application routines to be called.

AVOID HARD-CODING YOUR PATHNAME

If your NDA needs to know its own pathname or the pathname of the directory it's in, call LGetPathname or LGetPathname2 using your User ID. This is a better method than hard-coding "*:System:Desk.Accs:MyDAName" because the user may change your DA's file name or use a utility to install it from some non-standard directory.

AVOID EXTRA GETNEWID CALLS

Normally there is no reason for a Desk Accessory to call GetNewID. When you can, just call MMStartUp to find your own User ID, and use that. You can freely use all the auxiliary IDs derived from your main ID (MMStartUp+$0100, MMStartUp+$0200, ..., MMStartUp+$0F00).

By not calling GetNewID, you conserve the limited supply of IDs (255 of in the $50xx range for Desk Accessories), and you make life easier for people trying to debug their systems, since all your allocated memory can be readily identified.

OPEN IS NOT CALLED IF NDA IS ALREADY OPEN

Your NDA's Open routine does not get called if the user chooses the NDA from the Apple menu while the NDA is already open. In this case, the Desk Manager simply calls SelectWindow on your existing window.

There is no need to include code in your Open routine to check if your window is already open, and to call SelectWindow if it is.

Further Reference

- Apple IIgs Toolbox Reference, Volumes 1-3
- GS/OS Reference
- Apple IIgs Hardware Reference
- Apple IIgs Technical Note #53, Desk Accessories and Tools
- Apple IIgs Technical Note #57, The Memory Manager and Interrupts
- Apple IIgs Technical Note #69, The Ins and Outs of Slot Arbitration

### END OF FILE TN.IIGS.071

###
### FILE: TN.IIGS.072
###

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#72: QuickDraw II Quirks

Revised by: Dave Lyons May 1992
Written by: Dave Lyons & C. K. Haun <TR> November 1989

This Technical Note points out some things things you need to watch out for when using QuickDraw II, especially with FastPort-aware and Shadowing modes.

CHANGES SINCE NOVEMBER 1990: Removed some obsolete information on ScrollRect and on shadowing. Noted that DrawPicture in 6.0 is now compatible with FastPort mode. Added a warning about making QuickDraw II calls while QuickDraw II is not started.

DON'T CALL QUICKDRAW II WHILE IT'S INACTIVE

Most QuickDraw II functions behave unpredictably if you call them while QuickDraw II is inactive, so watch it! Don't make QuickDraw II calls while QuickDraw II isn't started, except as documented. GrafOn and GrafOff are okay. (And so are QDStartUp, QDVersion, and QDStatus.)

FASTPORT-AWARE ANOMALY

Before System 6.0, when the FastPort-aware bit is turned on in the MasterSCB parameter to QDStartUp, DrawPicture did not notice changes in the pen pattern. If your application does not require 6.0 and uses pictures, either directly or indirectly (i.e., by printing to the ImageWriter driver), you may need to leave FastPort-aware mode turned off to get the expected behavior.

FASTFONT AND LARGE PIXEL MAPS

FastFont does not work correctly when drawing past the first 64K of a pixel map. If you are drawing text that uses FastFont (i.e., Shaston 8), you can avoid this problem by using a non-rectangular clipRgn.

DON'T SHOWPEN WHILE COLLECTING POLYGONS, REGIONS, OR PICTURES

The Macintosh QuickDraw documentation permits calling ShowPen after an OpenPoly, OpenRgn, or OpenPicture call to cause drawing calls to contribute to a polygon, region, or picture AND draw to a pixel map at the same time.

The Apple IIgs QuickDraw II documentation does not say you can do that. In some cases, it works, but it works "by accident" and it's not one of the things Apple tests or guarantees in QuickDraw II.

YOU MAY NEED SETBUFDIMS!

The call description for SetBufDims on page 16-215 of Volume 2 of theToolbox Reference is misleading. The note in the description states, "You only need to make this call if your application is going to use, or allow the user to choose, fonts that have unusually large values of chExtra and spExtra." This is not true; you need to call SetBufDims to adjust the clipping buffers for your application if you plan to use a clipRgn that has a greater width than the width you passed at QDStartUp.

SetBufDims sets the clipping buffer width as well as that of the text buffer, so if you plan to use a clipping region larger than the startup port width you must use SetBufDims.

Be aware that this call may be necessary even if your application does not ever set a clipping region or rectangle. Some toolbox calls assume that the clipping buffer size is correct based on the parameters passed to that routine. For example, if the locInfo you pass to CopyPixels has a width parameter that is wider than the width you passed at QDStartUp, CopyPixels may fail. A safe rule of thumb is to make sure (possibly by setting) that the width parameter in the buffer dimensions is the same or greater than the widest width in the locInfo structures passed to routines that use them.

Further Reference

- Apple IIgs Toolbox Reference, Volumes 1 and 3

### END OF FILE TN.IIGS.072

```
#####################################################################
### FILE: TN.IIGS.073
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#73: Using User Tool Sets

Revised by: Dave "Flag Bits" Lyons — July 1991
Written by: Dave Lyons — November 1989

This Technical Note explains how to write a user tool set and why writing a user tool set is better than stealing a system tool set number.

Changes since January 1991: Expanded recommendation on where to keep user tool set files on disk and clarified SetTSPtr information.

The Apple IIgs Toolbox Reference describes system tool sets, which are usually called through the system tool dispatcher vectors 1 ($E10000) and 2 ($E10004).

There are 255 possible system tool set numbers (1 through 255). All of these are reserved for definition by the system. If your program is "borrowing" a system tool set number, please feel guilty and switch over to the user tool set numbers. There are 255 of them too, and they're called through user tool dispatcher vectors 1 ($E10008) and 2 ($E1000C). All 255 user tool set numbers are available for the current application to use as it chooses. (Desk accessories are forbidden to use user tool sets.)

Of the four tool dispatcher vectors, only the first one ($E10000) has received a lot of publicity. $E10008 works just like $E10000, except that it passes control to a user tool set instead of a system tool set.

The second vector of each pair ($E10004 and $E1000C) works just like the first, except that one extra RTL address must be pushed onto the stack after any parameters are pushed. This way you can have a subroutine to do some or all of your toolbox dispatching, and that subroutine can do extra processing before or after the tool call, or both.

How Can I Write a User Tool Set?

Appendix A of Toolbox Reference, Volume 2, shows how to write a user tool set. Your tool set's Work Area Pointer is a four-byte value you can set with SetWAP and get with GetWAP. The WAP value is already loaded into the Y and A registers every time one of your tool set's functions gets control. The traditional use for the WAP is to keep track of an area of memory owned by your tool set.

If you do use the WAP in a conventional way, your xxxStatus function should return TRUE if the WAP is nonzero; your xxxStartUp function should set the WAP to a non-zero value pointing to some memory space you own (provided by the caller, or allocated with NewHandle using a memory ID provided by the caller); and your xxxShutDown function should set the WAP back to zero.

Since the X register contains the tool set and function number when one of your functions gets control, it is not necessary for a tool set to be written to be used as a predetermined user tool set number. At execution time, your tool set can compute the proper error codes and values to send to GetWAP and SetWAP.

Note: At the bottom of page A-8 of the Apple IIgs Toolbox Reference, Volume 2, "lda #$90" should read "lda #$8100" for version 1.0 prototype. On page A-10, the figure should show two RTL addresses (6 bytes) on the stack.

ToStrip and ToBusyStrip Vectors

These two vectors are for tool sets to jump to when a function exits.

```
ToBusyStrip     $E10180
ToStrip         $E10184

Inputs:         X = error code (0 if no error)
                Y = number of bytes of input parameters to strip
```

When your function is ready to exit, set up the registers and jump to ToStrip. It shifts the six bytes of RTL addresses up by Y bytes, sets up A and the carry flag appropriately, and returns to whomever called the tool.

If the system busy flag needs to be decremented, jump to ToBusyStrip instead of ToStrip.

How Can I Load My Tool Set From Disk?

One way to load your tool set from disk is to use InitialLoad or InitialLoad2, supplying a pathname like "9:MyToolset" (prefix 9 is initially set to the directory containing your application; prefix 1 also works, but its length is limited to 64 characters). You can then use SetTSPtr to tell the Tool Locator about your tool set, as shown in Appendix A.

Note that SetTSPtr calls your xxxBootInit function. Even if there is no useful work to be done at BootInit time, you still need to have a BootInit function (function number 1) that returns $0000 in the Accumulator and the carry flag cleared..

When you're done with your tool set, call UserShutdown on the memory ID returned by InitialLoad, so the memory it's using is disposed of or made purgeable. (You can shut it down and allow it to remain in memory in a purgeable state; if you do this, you should try to revive your tool set with Restart before you try InitialLoad or InitialLoad2.)

To allow several applications to share one copy of a user tool set file, you may want to keep your user tool set in the user's *:System:Tools folder. To avoid duplicate file names, leave the ToolXXX names for System tool sets, and give your user tool set a descriptive name.

If your tool set is not found in the *:System:Tools folder, you can then check the 9: folder. This way users do not need to burden their *:System:Tools folders if few of their applications use a particular user tool set or if space on their boot volume is limited.

When your application quits and calls TLShutDown, the system disconnects your tool set from the user tool set TPT. If the UserShutDown is not followed immediately by the TLShutDown, you may wish to use SetTSPtr to cleanly remove your tool set from the system (set the tool set pointer so that it points at a zero word).

Note: Because of the way the tool dispatcher transfers control to toolbox functions, a function's entry point must not be at the first byte of a bank ($xx0000). This is normally not an issue, since it's common to put the actual code right after the function pointer table, all in one load segment. Just make sure no function begins at the first byte of a load segment, and you're safe.

Further Reference

- Apple IIgs Toolbox Reference, Volume 2
- GS/OS Reference

### END OF FILE TN.IIGS.073

```
#####################################################################
### FILE: TN.IIGS.074
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#74: Top Ten List Manager Things

Revised by: Dave Lyons May 1992
Written by: Jim Mensch November 1989

This Technical Note presents a method for speeding up custom List Draw routines, with sample source code for the APW assembler.

CHANGES SINCE NOVEMBER 1989: Added information on memFlag and on shared rListRef resources, and noted that System 6.0 already checks the clip region and calls your listDraw routine only when needed.

TEN--MORE MEMFLAG BITS

In each member record, bits 0 and 1 of memFlag indicate whether memPtr is a pointer, handle, or resource ID. You don't normally have to worry about that--a custom listDraw routine is one place that you do. The complete definition of memFlag is as follows:

Bit	Description
7	memSelected
6	memDisabled
5	memNever (Inactive)
4-2	reserved--set to zero
1-0	00 = memPtr is a pointer
	01 = memPtr is a handle
	10 = memPtr is a resource ID (type is rPString or rCString)
	11 = reserved

NINE--SHARING RLISTREF RESOURCES

When listRef is a resource ID, the List Manager calls LoadResource every time it needs your rListRef resource. If two or more lists share the same rListRef, they will get the same handle from LoadResource and will interfere with each other.

To give each list its own copy of your the rListRef resource, load the resource yourself and use DetachResource. Then feed the listRef to the List Manager as a handle. Repeat the process for each list.

EIGHT--CUSTOM LISTDRAW ROUTINES AND THE CLIP REGION

The custom listDraw routine below speeds up your list when running System

Software earlier than 6.0. The System 6.0 List Manager already calls your listDraw routine only for members that will not be completely clipped (but this is still a good starting point if you're writing a custom listDraw routine for some other reason).

To scroll text, the List Manager calls ScrollRect to scroll the list--then 6.0 redraws the newly-exposed members, and older versions redraw all the visible members. On small lists this is fine, but on larger lists it can cause the redrawing of much data that is already on the screen, which can take time. If your application does not require 6.0, you may want to use a custom listDraw routine like this one.

First, we check the current clipRgn (which the List Manager was kind enough to shrink down to include only the portion of the list that needs redrawing) against the passed item rectangle. If the rectangle is in any way enclosed in the clipRgn, then the member is redrawn; otherwise the routine simply returns to the List Manager without drawing. This sample routine is designed to work only with Pascal-style strings, but it can be easily modified to use any other type of string you choose.

```
MyListDraw     Start
;
; This routine draws a list member if any part of the member's
; rectangle is inside the current clipRgn.
;
; Note that the Data Bank register is not defined on entry
; to this routine.  If you use any absolute addressing, you
; must set B yourself and restore its value before exiting.
;
top           equ  0
left          equ  top+2
bottom        equ  left+2
right         equ  bottom+2
rgnBounds     equ  2
;
oldDPage      equ  1
theRTL        equ  oldDPage+2
listHand      equ  theRTL+3
memPtr        equ  listHand+4
theRect       equ  memPtr+4
              using globals

              phd
              tsc
              tcd

              pha
              pha
              _GetClipHandle
              PullLong listHand

              ldy #2
              lda [listhand],y
              tax
              lda [listhand]
              sta listhand
              stx listhand+2
              lda [therect]                ; now test the top
              dec a                        ; adjust and give a little slack
              ldy #rgnbounds+bottom
              cmp [listhand],y             ; rgnRectBottom>=top?
              blt skip2
              brl NoDraw                   ; if not don't draw..
Skip2         ldy #bottom                  ; now see if the bottom is higher
than the top
              inc a                        ; give a little slack
              lda [therect],y
              ldy #rgnBounds+top
              cmp [listhand],y
              blt NoDraw
NoTest        ANOP

              PushLong theRect
              _EraseRect             ; erase the old rectangle

              ldy #left
              lda [theRect],y
              tax
              ldy #bottom
              lda [theRect],y
              dec a
              phx
              pha
              _MoveTo
              ldy #2
              lda [memptr],y
              pha
              lda [memptr]
              pha
              _DrawString

              ldy #4
              lda [memPtr],y
              and #$00C0             ; strip to the 6 and 7 bits
              beq memDrawn           ; if they are both 0 the member is drawn
              cmp #$0080             ; member selected?
              bne noSelect           ; member not selectable
              PushLong theRect
              _InvertRect
              bra memDrawn
; if we get here the member is disabled
noSelect      PushLong #DimMask
              _SetPenMask
              PushLong theRect
              _EraseRect
              PushLong #NorMask
              _SetPenMask
memDrawn      ANOP


; exit here
              pld
              sep #$20
              longa off
              pla
              ply

              plx
              plx
              plx
              plx
              plx
              plx
              phy
              pha
              rep #$20
              longa on
              rtl

DimMask       dc  i1'$55,$AA,$55,$AA,$55,$AA,$55,$AA'
NorMask       dc  i1'$FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF'
              end
```

SEVEN THROUGH ONE--RESERVED FOR FUTURE EXPANSION

Further Reference

- Apple IIgs Toolbox Reference, Volumes 1 and 3

### END OF FILE TN.IIGS.074

```
#####################################################################
### FILE: TN.IIGS.075
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#75: BeginUpdate Anomaly

Revised by: Dave Lyons May 1992
Written by: Eric Soldan January 1990

This Technical Note discusses a Window Manager anomaly with the handling of the visRgn and the updateRgn between BeginUpdate and EndUpdate calls.

CHANGES SINCE JANUARY 1990: Updated for System 6.0. CopyPixels is in a static segment, and GS/OS automatically prompts for disks on the text screen when necessary to avoid interfering with a window update in progress.

If an application calls BeginUpdate, it needs to be fully aware of what is going on behind the scenes in terms of its visRgn and updateRgn. Typically an application has TaskMaster handle the update events. TaskMaster calls BeginUpdate, the application update procedure, then EndUpdate. So any application that uses TaskMaster to handle updates, whether or not it makes any BeginUpdate calls directly, needs to be aware of problem described in this Note.

BeginUpdate is responsible for intersecting the visRgn and the updateRgn and making the intersection of these two regions the temporary visRgn. (EndUpdate undoes this effect.) Following are the steps BeginUpdate takes to do this:

1. Localize the updateRgn. (All grafPort regions are local, therefore the visRgn is local. All window regions are global, therefore the updateRgn is global. One of them has to change if they are to be intersected correctly.)
2. Intersect the visRgn and localized updateRgn, then place the result in the updateRgn.
3. Swap the visRgn and updateRgn handles.

   The handle swapping has two effects:

   - Makes the intersection region the current visRgn.
   - Saves the real visRgn as the updateRgn. (Saving the real visRgn is necessary because everything has to be restored to normal by EndUpdate.)

EndUpdate restores things to normal after an update procedure is finished. When an application calls EndUpdate, it swaps back the handles and sets the updateRgn to empty.

SO WHAT'S THE PROBLEM?

The problem is that the updateRgn is not a very good place to save the visRgn. Since InvalRect and InvalRgn modify the updateRgn, if either of these two calls is made between a BeginUpdate and EndUpdate, they modify the saved visRgn. When the update is finished, EndUpdate restores the modified visRgn instead of the original.

The solution to this problem seems simple enough: don't call InvalRect or InvalRgn between BeginUpdate and EndUpdate. Unfortunately, there are other calls which can call BeginUpdate, EndUpdate, InvalRect, and InvalRgn, so an application might inadvertently call one of these routines.

If this situation isn't bad enough already, you could really mess things up by opening another window between BeginUpdate and EndUpdate calls. Opening a window at this time may seem like a perfectly normal thing (i.e., to display an alert); however, opening a window forces the recalculation of the visRgn for any windows obscured by the new window. If the window being updated has its visRgn recalculated, the application obviously loses the visRgn that BeginUpdate created. This doesn't seem too serious since the visRgn is restored to the entire visible part of the window when the new window is closed; however, it does mean that the application would have to update the entire window instead of the original updateRgn.

Unfortunately, the Window Manager also posts update events for the portion of the window that was obscured, and it does this by changing the updateRgn. Of course the updateRgn for the window being updated is really the visRgn that is being "safely" preserved until the EndUpdate call. So, there are some really good reasons why this can't be done.

Okay, so along with not making calls to InvalRect and InvalRgn between BeginUpdate and EndUpdate, an application cannot open any other windows either. Good.

NOW TO MAKE THINGS EVEN WORSE.

Starting with System 5.0, some toolbox functions are stored on disk in dynamic segments and loaded when they are first called. For example, CopyPixels is in a dynamic segment in System versions 5.0 through 5.0.3. If the startup disk is not available and the system prompts for it between BeginUpdate and EndUpdate by calling AlertWindow, the bad things discussed above happen.

Starting with System 6.0, the system is smart enough not to prompt for a disk using AlertWindow if a window update is in progress. (Internally, GS/OS calls WindStatus to see if it can prompt on the graphics screen. If BeginUpdate has been called more times than EndUpdate, WindStatus fibs by returning with the carry set. GS/OS takes the hint and prompts for the disk with a text dialog instead.)
But I Have to Do...

If you absolutely must do some of the things previously discussed, there is a way to accomplish it. It is not simple, but it can be done.

Assuming that BeginUpdate has been called, and an application is in its update procedure:

1. Create a new region and copy the visRgn into it. Doing this allows the application to restore the visRgn to just the area to be updated that BeginUpdate calculated. This needs to be done for any other windows which obscure a part the the window being

updated. Again, these are not windows that an application would open directly. CopyPixels may open a window, since it is a dynamic segment and may need to get loaded from a disk that is off-line.

2. Create a new region, then swap its handle with the updateRgn handle. This protects the real visRgn and lets an application call InvalRect and InvalRgn at any time if necessary. It also means the application doesn't need to worry about the system making these calls either. The updateRgn is also an empty region after the swap, so any contributions to it constitute a valid update event that needs to be handled.
3. Do the update part of the update procedure. In this part, if the application has any calls to CopyPixels, or any other QuickDraw Auxiliary dynamic segment functions, after the call is completed, copy the saved visRgn back to the visRgn of the grafPort. The closing of the dynamic segment alert window recalculates the visRgn, and copying it undoes this effect. Do not do the same for the updateRgn. Leave the updateRgn alone. We are accumulating an actual updateRgn, and the closing of the alert window for the dynamic segment may have contributed to this region.

There are two methods for leaving the update procedure. Although the second method works whether or not an application uses TaskMaster, if an application does not use TaskMaster, then the first method is simpler.

The procedure without using TaskMaster (i.e., you made the BeginUpdate call, and you will make the EndUpdate call) is as follows:

A. Dispose of the region created in Step 1. This region was only needed to restore the partial visRgn that BeginUpdate calculated after a window was opened.
B. Swap the updateRgn handle with the region handle created in Step 2.
C. Make the EndUpdate call.
D. If the region created in Step 2 is not empty, copy this region into the updateRgn for the window with CopyRgn. You can't just do an InvalRgn with it because InvalRgn globalizes the region and the region is already global. You want to copy the region since this generates a valid update event. You can use CopyRgn instead of UnionRgn because the update region is empty.
E. Dispose of the region created in Step 2.

With TaskMaster, things are a little messier. Since TaskMaster makes the EndUpdate call, you have less control over the situation. It is important to do the EndUpdate before generating the update event. Posting the update event has to happen outside the update procedure, since you have to leave the update procedure for TaskMaster to do the EndUpdate. So it follows that you do Steps A and B, post an application event to handle the rest externally, and when the application event is handled, do Steps D and E.
Some consideration was given to posting an application event via the PostEvent call. Unfortunately, there is a possibility that this application event will drop out of the queue not handled. When the queue is full, the oldest event is dropped, and this could occur to application events, which would be very bad in this case. Due to this possibility, posting an application event refers to setting a global variable that is checked before the TaskMaster call in the main event loop. This can be considered equivalent to posting an event via the PostEvent call.

So, the TaskMaster case would be as follows:

A. Dispose of the region created in Step 1.
B. Swap the updateRgn handle with the region handle created in Step 2.
C. Store the handle of the region created in Step 2 in a global variable named eventUpdateRgn. Store the current window port in a global variable named eventWindowPort.
D. Return to TaskMaster, which returns to the main event loop.
E. Immediately after the TaskMaster call in the main event loop, check the global variable eventUpdateRgn. If it is not NULL then:
   a. Copy the region into the updateRgn of the window eventWindowPort. Using CopyRgn is the easiest way to copy the region. (Copying the region posts an update event if the event UpdateRgn is not NULL.
   b. Dispose of the region eventUpdateRgn, then set the variable eventUpdateRgn to NULL, so that this "event" won't be handled again.

Of course, the simplest way to handle all of this is to avoid situations where you have to take the steps described above. If things like opening a window (or allowing the system to open one) and InvalRect and InvalRgn can be avoided between calls to BeginUpdate and EndUpdate, so can all of this ugliness.

Further Reference

- Apple IIgs Toolbox Reference, Volume 2

### END OF FILE TN.IIGS.075

```
#####################################################################
### FILE: TN.IIGS.076
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#76: Miscellaneous Resource Formats

Revised by: Matt Deatherage — May 1992
Written by: Matt Deatherage, C.K. Haun, Llew Roberts & Dave Lyons — January 1990

This Technical Note describes resource structure formats for previously-unpublished types.

CHANGES SINCE DECEMBER 1991: Added information on rFont resources. Clarified the note about rVersion resources to note that version numbers must increase with subsequent releases for the Finder's benefit.

The format used to describe the resources is similar to that used in File Type Notes, where the offsets, given in the form (+xxx), determine the offset from the beginning of the resource.

SAMPLED SOUND RESOURCE (TYPE: $8024, RSOUNDSAMPLE)

The following describes the Sampled Sound resource format. It consists of a ten-byte header followed by the sample data bytes.

Format	(+000)	Word	This must always be zero.
Wave Size	(+002)	Word	Sample size in pages (256 bytes per page). For example, an 8K sample takes 32 pages; a 128K sample requires $200 pages.
Rel Pitch	(+004)	Word	The high byte of this word is a semitone value; the low byte is a fractional semitone. These values are used to tune the sample to correct pitch. See HyperCard IIgs Technical Note #3, Pitching Sampled Sound.
Stereo	(+006)	Word	The output channel for this sound is in the low nibble of this word.
Sample rate	(+008)	Word	The sampling rate of the sound, in Hertz (Hz).
Sound	(+010)	Bytes	The sampled sound data. The bytes are all 8-bit samples. The sample starts here and continues until the end of the resource.

The resource compiler template follows:

```
#define rSoundSample     $8024

/*--------------------- rSoundSample -------------------*/
type rSoundSample {
     integer;            /* format */
     integer;            /* wave size */
     hex integer;        /* rel pitch */
     integer;            /* stereo channel */
     unsigned integer;   /* sample rate */
     hex string;         /* raw 8 bit sound data */
};
```

CURSOR RESOURCE (TYPE: $8027, RCURSOR)

The following describes the Cursor resource format:

```
   height   (+000)   Word  The height of the cursor, in pixels.
   width    (+002)   Word  The width of the cursor, in Words.
   image    (+004)   Bytes The image of the cursor. There are height*width
                     Words in the cursor, or twice that many Bytes.

   We will call the first byte beyond the image offset "ei" for "end of
   image."

   mask     (+ei)    Bytes The mask of the cursor. This is the same size as
                     the image.

   We will call the first byte beyond the mask offset "em" for "end of mask."

   hotSpotY (+em)    Word  The cursor's Y "hot spot."
   hotSpotX (+em+2)  Word  The cursor's X "hot spot."
   flags    (+em+4)  Flag  Cursor flags:
                           Bit 7: 1 = 640 Mode, 0 = 320 Mode
                           All other bits are reserved and must be zero.
   reserved (+em+6) 8 Bytes Reserved, must be zero.
```

The resource compiler template follows:

```
#define rCursor     $8027

/*--------------------- rCursor --------------------*/
type rCursor {
    height :
        hex integer;            /* height */
    width :
        hex integer;            /* width in words */
        hex string[2*$$Word(height)*$$Word(width)];    /* cursor image */
        hex string[2*$$Word(height)*$$Word(width)];    /* cursor mask */
        hex integer;            /* hotspot Y */
        hex integer;            /* hotspot X */
        hex integer;            /* flags */
        hex longint = 0;        /* reserved */
        hex longint = 0;        /* reserved */
    };
};
```

Following is a simple cursor example:

```
resource rCursor(1,fixed) {
        5,     /* height */
        2,     /* width */
        $"ffff0000"
        $"f00f0000"
        $"f00f0000"
        $"f00f0000"
        $"ffff0000",

        $"ffff0000"
        $"ffff0000"
        $"ffff0000"
        $"ffff0000"
        $"ffff0000",

        2,      /* hotspot Y */
        2,      /* hotspot X */
        $80     /* 640 mode */
};
```

Note that the resource is marked fixed so that its handle can be dereferenced and passed to SetCursor.

VERSION RESOURCE (TYPE: $8029, RVERSION)

Files may include a version resource with ID=1 for display by programs such as the Finder. All rVersion resource IDs other than 1 are reserved for future definition. The following describes the version resource format:

version	(+000)	Long	The application's version number, in Apple IIgs Long Version format. See Apple IIgs Technical Note #100, VersionVille, for more details.
country	(+004)	Word	An international country version code. Possible values are as follows:

verUS	0
verFrance	1
verBritain	2
verGermany	3
verItaly	4
verNetherlands	5
verBelgiumLux	6
verSweden	7
verSpain	8
verDenmark	9
verPortugal	10
verFrCanada	11
verNorway	12
verIsrael	13
verJapan	14
verAustralia	15
verArabia	16
verFinland	17
verFrSwiss	18
verGrSwiss	19
verGreece	20
verIceland	21
verMalta	22
verCyprus	23
verTurkey	24

```
                                                   verYugoslavia  25
                                                   verIreland     50
                                                   verKorea       51
                                                   verChina       52
                                                   verTaiwan      53
                                                   verThailand    54

   name          (+006)   String  Pascal string containing the desired name.
                                  May be the null string.
   moreInfo      (+xxx)   String  Additional information to be displayed,
                          such as a copyright notice. May be the null string.
                          Recommended maximum length is about two lines of 35
                          characters each. May contain a carriage return
                          (character $0D).
```

The resource compiler template follows:

```
#define rVersion    $8029

// Equates for the country code of an rVersion resource

#define Region \
   verUS, verFrance, verBritain, verGermany,
   verItaly, verNetherlands, verBelgiumLux,
   verFrBelgiumLux = 6, verSweden, verSpain,
   verDenmark, verPortugal, verFrCanada, verNorway,
   verIsrael, verJapan, verAustralia, verArabia,
   verArabic=16, verFinland, verFrSwiss, verGrSwiss,
   verGreece, verIceland, verMalta, verCyprus,
   verTurkey, verYugoslavia, verYugoCroatian = 25,
   verIndia = 33, verIndiaHindi = 33, verPakistan,
   verLithuania = 41, verPoland, verHungary,
   verEstonia, verLatvia, verLapland, verFaeroeIsl,
   verIran, verRussia, verIreland = 50, verKorea,
   verChina, verTaiwan, verThailand

/*------------------- rVersion ------------------*/
type rVersion {
     ReverseBytes {
          hex byte;                            // Major revision in BCD
          hex bitstring[4];                    // Minor vevision in BCD
          hex bitstring[4];                    // Bug version
          hex byte  development = 0x20,        // Release stage
                         alpha = 0x40,
                         beta = 0x60,
                         final = 0x80, /* or */ release = 0xA0;
          hex byte;                          // Non-final release #
     };
     integer   Region;                        // Region code
     pstring;                                 // Short version number
     pstring;                                 // Long version number
};
```

Following is a simple version example for "Super Graphics Destroyer", version 2.0:

```
resource rVersion(1) {
       { $02,$0,$0,release,$00 },       /* version 2.0 release */
```

```
     verUS,                        /* US version */
     "Super Graphics Destroyer",        /* our app's name */
     "(C) 1991 Pretty as a Picture, Inc."    /* the copyright notice */
};
```

   NOTE: For compatibility with the Finder, keep the name field identical across different versions of the same application, and make sure the version field increases on each later version released to your customers. The moreInfo field is not critical; if it changes between versions, it's no big deal.

COMMENT RESOURCE (TYPE: $802A, RCOMMENT)

Files may include a comment resource with ID=1 for display by programs such as the Finder. All rComment resource IDs other than 1 are reserved for future definition. The following describes the comment resource format:

text	(+000)	Bytes	The comment. This is unformatted, 8-bit text suitable for displaying by a desktop program. No length limit is imposed by this resource format, although a practical limit of a few hundred characters is recommended.

The resource compiler template follows:

```
#define rComment    $802A

/*--------------------- rComment -------------------*/
type rComment {
    string;
};
```

TAGGED STRINGS RESOURCE (TYPE: $802E, RTAGGEDSTRINGS)

A tagged strings resource lists pairs of Word values and Pascal strings.

count	(+000)	Word	Number of word/string pairs in this resource.
firstWord	(+002)	Word	Word value of first pair.
firstString	(+004)	String	Pascal string of first pair.
secondWord	(+xxx)	Word	Word value of second pair.
secondString	(+yyy)	String	Pascal string of second pair.
...			

The resource compiler template follows:

```
#define rTaggedStrings        $802E

/*------------------- rTaggedStrings --------------------*/

type rTaggedStrings {
          integer = $$Countof(StringArray);
          array StringArray {
                    hex integer;        /* Key integer */
                    pstring;            /* String */
          };
};
```

Following is a simple rTaggedStrings example:

```
resource rTaggedStrings(1) {{
     $0050, "red",
     $0033, "green",
     $0100, "blue"
}};
```

PATTERN LIST RESOURCE (TYPE: $802F, RPATTERNLIST)

A pattern list resource contains zero of more 32-byte QuickDraw II patterns. (This resource type exists for your convenience. The System Software contains no direct support for resources of this type.)

```
firstPattern   (+000)    32 Bytes  First QuickDraw II pattern structure.
secondPattern  (+032)    32 Bytes  Second QuickDraw II pattern structure.
...
```

The resource compiler template follows:

```
#define rPatternList          $802F

/*-------------------- rPatternList ---------------------*/

type rPatternList {
     array {
          array[32] {
               hex byte;
          };
     };
};
```

RECTANGLE LIST RESOURCE (TYPE: $C001, RRECTLIST)

The rectangle list (type rRectList) is provided to allow an extensible, easily modifiable collection of QuickDraw II rectangle structures. This capability can enhance a developer's ability to modify on-screen displays without recompiling an entire application. This resource also enables easier cross-development and parallel development for the Apple IIgs and the Macintosh.

The following describes the rectangle list resource format:

```
count          (+000)    Word      Number of rectangles in this resource.
firstRectangle (+002)    8 Bytes   First QuickDraw II rectangle structure.
                ...                Rectangles, eight bytes each.
lastRectangle  (+002+(8*(count-1)))
                         8 Bytes   Last QuickDraw II rectangle structure.
```

The resource compiler template follows:

```
#define rRectList    $C001
type rRectList {
        integer = $$Countof(RectArray);
        array RectArray {
            Rect;
            };
};
```

PRINT RECORD RESOURCE (TYPE: $C002, RPRINTRECORD)

As a convenience for applications, a print record may be included as a resource of type $C002 (rPrintRecord). If more than one of these resources is present, the one to use as the document's primary print record is the first one. You can get this resource's ID by calling GetIndResource with type rPrintRecord and index 1. Storing the primary print record with ID = 1 is a good way to start.

Since the print record is filled in and interpreted by the printer driver, you can't always programmatically set options that are driver-specific. For example, although the ImageWriter driver stores the color-vs.-black and white option in one place, not all color printers will do the same thing. If you want to use driver-specific options on many printers, you can use those printer drivers to create print record that reflect the options you want and store those records as resources. Then, if you need a pre-initialized print record with the options you want, you may already have one.

print record	(+000)	160 Bytes	The print record. The handle to this resource is suitable for passing to any Print Manager call that requires a print record handle.

Since applications shouldn't create print records from scratch, but rather allow printer drivers to fill them in with PrDefault and PrVerify, the resource compiler template that follows only allocates 160 bytes for storage.

```
#define rPrintRecord     $C002

/*-------------------- rPrintRecord ------------------*/
type rPrintRecord {
     array[160] {
          hex byte;
     };
};
```

FONT RESOURCE (TYPE: $C003, RFONT)

Some applications wish to keep fonts with the application itself instead of in a separate font file. This isn't always advisable--fonts not in font files can't be easily used in other applications, which may confuse users who, for example, use text-editing desk accessories and can't get at certain fonts except in certain applications. Also, fonts not in the Fonts directory must be completely memory-resident where normal Fonts are only loaded from disk when they're needed.

Nevertheless, in some cases keeping fonts inside an application file is necessary or desirable. In such cases, the rFont resource is a convenient way to keep a font around. The resource is a QuickDraw II Font, as defined in Apple IIgs Toolbox Reference, Volume 2. The font family name is the resource's name--the same Pascal string that normally precedes the QuickDraw II font record in the font file.

```
   font  (+000)  Bytes     The font record, without the font family name.
```

Since the font record is a bunch of variable-sized tables, and since you probably want to use a font editor (and not the resource compiler) to create fonts, the following resource compiler template isn't very revealing.

```
/*-------------------- rFont ------------------*/

type rFont {
     hex string;
};
```

Further Reference

- Apple IIgs Toolbox Reference, Volumes 1-3
- Apple IIgs Technical Note #100, VersionVille
- HyperCard IIgs Technical Note #3, Pitching Sampled Sounds

### END OF FILE TN.IIGS.076

```
#####################################################################
### FILE: TN.IIGS.077
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#77: Print Manager & AppleTalk Configuration Files

Written by: Jim Luther — January 1990

This Technical Note describes the Print Manager user configuration file Printer.Setup and the AppleTalk user configuration file ATInit. This Note also describes a limitation of the Print Manager call PrGetUserName, which is a result of the way configuration information is stored in the Printer.Setup file.

Printer.Setup and ATInit

What Are the Printer.Setup and ATInit Files?

The Print Manager user configuration file Printer.Setup, which is found in the System:Drivers directory of the Apple IIGS boot disk, is used by the Print Manager tool set to keep the name of the printer driver and port driver you've selected between system boots. In addition, if you've selected a network printer, Printer.Setup contains the selected printer's network address and your machine's User Name. The file format of Printer.Setup has not been published because revisions have been made, and may be made again, to the Apple IIGS System Software, which can change Printer.Setup's file format.

The AppleTalk user configuration file ATInit, which is found in the System:System.Setup directory of the Apple IIGS boot disk, is used to keep the default AppleShare startup application, the default AppleShare prefix, the default AppleTalk User Name, and the default AppleTalk printer entity name (the network printer entity used by AppleTalk's Remote Print Manager) between system boots. The file format of the ATInit file was published incorrectly in the AppleShare Programmer's Guide for the Apple IIGS. The correct file format for ATInit will be discussed later in this Note.

It is important to remember that the Print Manager tool set uses the information from the Printer.Setup file only, and that AppleTalk and AppleShare use the information contained in the ATInit file only. It is also important to note that the Print Manger tool set, which is used to print QuickDraw II graphics, and AppleTalk's Remote Print Manager (RPM), which is used to print ASCII data to network printers, are not the same thing even though both contain the words "Print Manager."

What Writes to the Printer.Setup and ATInit Files?

Before Apple IIGS System Software 5.0, Printer.Setup and ATInit were handled as completely separate configuration files. The Print Manager call

PrChoosePrinter allowed you to select the printer and port drivers the Print Manager would use and wrote the printer and port driver selections to the Printer.Setup file. The AppleTalk application Chooser.II let you select the printer AppleTalk's Remote Print Manager would use and wrote the printer entity selection to the ATInit file.

With System Software 5.0 all printer selections for both the Print Manager and AppleTalk are made by using one of the Control Panel NDA's printer CDevs. All printer CDevs (e.g., DirectConnect, ATIWriter, ATLQIWriter, and ATLWriter) write the new printer and port driver selections to the Printer.Setup file. However, if the printer selected uses the AppleTalk port (i.e., the selection is made with the ATIWriter, ATLQIWriter, or ATLWriter printer CDevs), then the selected printer's network address and your User Name are written to both the Printer.Setup and the ATInit files. The DirectConnect CDev does not write any information to the ATInit file. If AppleShare is installed, then the AppleShare CDev will also write your User Name to the ATInit file.

On AppleShare file servers with the Apple II Setup option installed, the ATInit file in User folders will also be written to by the AppleShare Admin application when the Apple II startup information is set.

When are the Printer.Setup and ATInit Files Read?

The Printer.Setup file is read by the Print Manager and by the printer CDevs. The Print Manager reads the information contained in the Printer.Setup file whenever the Print Manager needs to load a printer driver or a port driver into memory. A printer CDev reads the information contained in the Printer.Setup file when that CDev is selected so it can know the current printer and port selections.

Ways the printer driver and the port driver might be unloaded and need to be loaded (which will cause Printer.Setup to be read by the Print Manager) are as follows:

- The Print Manager is shut down.

- The current printer driver or port driver is changed with a Control Panel printer CDev. When a new printer or port is selected with a printer CDev, the current drivers are unloaded from memory so the Print Manager will be forced to read the new printer and port selections from Printer.Setup.

- Your application makes the PMUnloadDriver Print Manager call.

An application can load one or both of the drivers (which will cause Printer.Setup to be read by the Print Manager) by making the PMLoadDriver call. The AppleTalk user configuration information contained in the ATInit file is read during system startup as part of AppleTalk's initialization.

Network Booting and Printer.Setup

When Apple IIGS computers are booted over an AppleShare network, they all share a single copy of the Printer.Setup file. That means all machines must use the same printer and port driver selections that are stored in the Printer.Setup file. If all machines are expected to be able to print using the Print Manager tool set, then the printer and port selection stored in Printer.Setup must be something that all can use. The only two options are:

- o A single shared network printer for all machines (i.e., a LaserWriter, an AppleTalk ImageWriter, or an AppleTalk ImageWriter LQ). In situations where many machines are booted over a single file server, this may cause the workload on the single shared printer to be unacceptable.

- o A direct-connect printer on each machine. The limitations of this solution are that the printers must be of the same type (all ImageWriters, all ImageWriter LQs, or all Epsons) and all machines must use the same printer port (either printer or modem).

The server administrator should set the default printer selection, which will be used by all machines, by using one of the Control Panel NDA's printer CDevs. Then, the access privileges to the server's System:Drivers directory should be set to "Bulletin Board" (i.e., Everyone See Folders, Everyone See Files, Owner Make Changes) so other machines cannot change the printer and port selection.

Using User Names

The User Name We Use

You may have noticed that you see your AppleTalk User Name in the Control Panel's AppleShare and printer CDevs. AppleShare allows a machine's User Name to be up to 31 characters long. The CDevs read the User Name from the ATInit file. The AppleShare and printer CDevs also store the complete User Name back into the ATInit file.

PrGetUserName (Almost)

The Printer.Setup file sets aside 15 characters for the User Name so the printer CDevs store only the first 15 characters of the User Name in the Printer.Setup file. This limitation is leftover from early Print Manager implementations of the PrChoosePrinter call, which limited the User Name length to 15 characters.

Since the Print Manager gets the User Name it uses from the Printer.Setup file, the User Name returned by the Print Manager call PrGetUserName will be truncated to 15 characters if the complete AppleTalk User Name is 16 characters or longer.

Where to Get the Complete User Name

If your application needs the complete default AppleTalk User Name, it can be read from the ATInit file. When an Apple IIGS is booted from a local disk volume that has AppleShare or at least one of the AppleTalk network printers installed, ATInit will be found in the System:System.Setup directory of the local boot volume. When an Apple IIGS is booted over AppleTalk, ATInit will be found in the Users:YourName:Setup directory of the AppleShare boot volume (where YourName is the User Name used to log on to the boot server).

The ATInit File Format

The AppleShare Programmer's Guide for the Apple IIGS shows the file format of the ATInit file as it is stored on an AppleShare boot volume. However, the file format of ATInit is not always as shown in that manual. In all cases, ATInit will contain the three required data fields UserName, PrinterFlags, and PrinterTuple at the end of the file. Before those data fields, ATInit may also contain executable code or additional data fields. Since the three required data fields are directly before ATInit's end-of-file (EOF), you can find them relative to ATInit's EOF using the displacements listed in Table 1.

Displacement to ATInit EOF	Size	Field Name	Description
133	33 Bytes	UserName	A Pascal-type string containing the default User Name. It consists of a length byte followed by up to 31 bytes of ASCII data and a single, unused byte. This field is always 33 bytes long.
100	Byte	PrinterFlags	This is the Flags field used by the Remote Print Manager's default network printer.
99	99 Bytes	PrinterTuple	This field specifies the name of the default network printer used by the Remote Print Manager. The PrinterTuple field is in standard Name Binding Protocol (NBP) format. This field is always 99 bytes long.

Table 1-Offsets of Required Data Fields

If the ATInit file is on an AppleShare server, it will have 6 additional data fields (PathVolID, PathDirID, Path, PrefixVolID, PrefixDirID, and Prefix) directly before the three required data fields. These fields can also be found relative to ATInit's EOF using the displacements listed in Table 2.

Displacement to ATInit EOF	Size	Field Name	Description
275	Word	PathVolID	The Volume ID number of the user's AppleTalk startup application.
273	Long	PathDirID	The Directory ID number of the user's AppleTalk startup application.
269	65 Bytes	Path	The Pathname of the user's AppleTalk startup application.
204	Word	PrefixVolID	The Volume ID number of the user's AppleTalk default prefix.
202	Long	PrefixDirID	The Directory ID number of the user's AppleTalk default prefix.
198	65 Bytes	Prefix	The user's AppleTalk default prefix.

Table 2-Offsets of Optional Data Fields

The displacements in Tables 1 and 2 can be used with the GS/OS SetMark call to move the file mark to the beginning of any of the above fields. The SetMark call's base field should be set to $0001 so the mark will be set equal to EOF minus the displacement.

Further Reference

- o Apple IIGS Toolbox Reference
- o Inside AppleTalk
- o AppleShare Programmer's Guide for the Apple IIGS

### END OF FILE TN.IIGS.077

```
#####################################################################
### FILE: TN.IIGS.078
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#78: Bank Alignment and Memory Management

Revised by: Matt Deatherage May 1992
Written by: Matt Deatherage March 1990

This Technical Note discusses the way the Memory Manager deals with requests for memory that is already in use, and why this can be really annoying.

CHANGES SINCE MARCH 1990: Included new information about some smarter algorithms in System Software 6.0 and later which can avoid problems some of the time.

The Memory Manager is a sophisticated software module that provides the framework for the allocation, moving, management, and disposal of blocks of memory; however, it's not magic.

When you ask the Memory Manager for a block of memory and it's not immediately allocatable, the Memory Manager starts through the procedure for purging, compacting, and calling out-of-memory (OOM) queue routines until, at the end of its rope, it finally gives up and returns error $0201. The exact procedure is repeated below, taken from Volume 3 of the Apple IIgs Toolbox Reference. Note that each successive step is only taken if, after the previous step, the requested memory still isn't available.

1. Calls each OOM queue routine until either all routines have been called or until one OOM queue routine reports that it has freed enough memory to satisfy the request.
2. Compacts memory.
3. Purges all level 3 handles. If this frees enough memory, compaction occurs.
4. Purges all level 2 handles. If this frees enough memory, compaction occurs.
5. Purges all level 1 handles. If this frees enough memory, compaction occurs.
6. Calls each OOM queue routine again until all have been called or until one OOM queue routine reports that it has freed enough memory to satisfy the request.
7. Gives up and returns error $0201.

This strategy works pretty well--as long as the request is for a block of memory wherever it fits. If someone has asked the Memory Manager for memory at a specific address, things get stickier.

Suppose that you've asked the Memory Manager for a handle starting at the beginning of bank 2, and that something else (i.e., the ProDOS FST) is already using that memory. The Memory Manager notices that the handle isn't immediately available, so it starts going through the listed procedures. Since the handle for the ProDOS FST is neither purgeable nor movable and GS/OS isn't likely to give it up in an OOM queue routine, the request fails and the Memory Manager returns error $0201.

However, the Memory Manager went through all the steps listed to get to the seventh step, the error. The Memory Manager has no way to know that one of the OOM queue routines isn't going to give up that particular handle and allow the request to be fulfilled. The OOM queue routines cannot know themselves, since they are only told how much memory is needed, not where it has to be. Therefore, whenever the Memory Manager returns error $0201, all purgeable handles have been purged.

This is particularly annoying to loaders. OMF supports a "bank-aligned" attribute for load segments, and the loaders ensure that such segments are loaded at the beginning of some bank or another. The Memory Manager does not have a "bank-aligned" attribute for handles, so the loaders have to do these things themselves. They do this by asking for a handle of the appropriate size at the beginning of bank two. If this fails, the loaders try again with bank three, then bank four, and so on through the end of memory.

Since some part of GS/OS is almost always occupying the memory at the beginning of bank two, which is where the loader first attempts to load a bank-aligned segment, the presence of such a segment in a load file virtually guarantees that all purgeable handles are purged when the file is loaded. This kicks out dormant applications and zombie-state tool sets, among other things, requiring they be loaded from disk again when needed.

Starting with System Software 6.0, the Loader attempts an alternate strategy first--it tries to allocate an entire bank of memory that is page aligned and doesn't cross a bank boundary. This block, if available, will by definition be bank-aligned. Since code segments can't be larger than 64K, such a block can always hold a bank-aligned segment. The Loader now tries to allocate such a block and if it finds one, it immediately disposes of it and allocates a block of the right size at the same bank address.

If this strategy succeeds, it's a lot faster than the other method and may avoid purging all of memory. If it fails, though, the Memory Manager still goes through all seven steps before returning error $0201, so all of memory may still be purged. It's just less likely in System Software 6.0 and later.

It doesn't make sense to bank-align a small segment, and small segments fit better into fragmented memory. If you use large segments anyway, consider the trade-off: bank-aligning a segment may purge memory at load time, but your linker may be able to generate smaller OMF, decreasing disk size and load time.

SUMMARY

The general recommendation against asking for specific blocks of memory is well-known to most developers; the reasons outlined above simply add fuel to the fire against such programming practices. What isn't as widely known is that having a bank-aligned load segment in a load file may cause everything purgeable to be purged, and could also cause OOM queue routines to dispose of handles when there really isn't any kind of memory shortage.

Apple advises developers to carefully consider the advantages and disadvantages of bank-aligned segments before including one in a load file.

Further Reference

- o Apple IIgs Toolbox Reference, Volumes 1 and 3
- o GS/OS Reference

### END OF FILE TN.IIGS.078

###
### FILE: TN.IIGS.079
###

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#79: Integer Math Data Types

Revised by: Jim Luther May 1990
Written by: Dan Strnad March 1990

This Technical Note describes the format of Fixed and Frac data types used by the Integer Math tool set and operations performed on the Integer Math numerical data types.
Revised since March 1990: Fixed original date, bit numbering of diagrams, and a multiplication sign in the equation.

As stated in Volume 1 of the Apple IIgs Technical Reference, the Integer Math tool set provides the following numerical data types:

    Integers
    Longints
    Fixed
    Frac
    Extended

The precise format of the Fixed and Frac data types is not provided in the reference manual, so this Note details these formats.

The format for the Fixed data type is stated in the manual as being a 32-bit signed value with 16 bits of fraction. This means that the low-order 16 bits of the Fixed format data value are considered as a fraction of 2^16, which is the binary number represented by a one followed by 16 zeroes ($10000). In other words, a Fixed value is the same as a long integer value whose binary point has been moved to the left 16 places. In this representation, if the low-order part of the Fixed format data value were $8000, the fractional value would be equal to 1/2. A low-order part of $C000 would represent a fractional part equal to 3/4. Therefore the highest value that a Fixed can contain is 32,767 and 65,535/65,536; the least value is equal to -32768.

```
     31          30          29                      18          17          16
 ___________ ___________ ___________ ____________ ___________ ___________ ___________
|           |           |           |            |           |           |           |
|  -32768   |  16384    |   8192    |    ...     |     4     |     2     |     1     |
|           |           |           |            |           |           |           |
|___________|___________|___________|____________|___________|___________|___________|
                                      high-order word

     15          14          13                       2           1           0
 ___________ ___________ ___________ ____________ ___________ ___________ ___________
|     1     |     1     |     1     |            |     1     |     1     |     1     |
|     -     |     -     |     -     |    ...     |   -----   |   -----   |   -----   |
|    2      |    4      |    8      |           |  16384    |  32786    |  65536    |
|___________|___________|___________|___________|___________|___________|___________|
                                    low-order word
```

Figure 1-Fixed Data Type

The format for the Frac data type is stated in the manual as being a 32-bit signed value with 30 bits of fraction. This means that the low-order 30 bits of the Frac format data value are considered as a fraction of 2^30, which is the binary number represented by a one followed by 30 zeroes ($40000000). In other words, a Frac value is the same as a long integer value whose binary point has been moved to the left 30 places. The high-order 2 bits of the Frac format data value are treated as follows. The high bit has a value of -2 and the low bit has a value of 1. Therefore the highest value that a Frac can contain is 1 and ((2^30)-1)/2^30; the least value is equal to -2.

```
     31          30          29                       18          17          16
 ___________ ___________ ___________ ____________ ___________ ___________ ___________
|           |           |     1     |            |     1     |     1     |     1     |
|    -2     |     1     |     -     |    ...     |   ----    |   ----    |   -----   |
|           |           |     2     |            |   4096    |   8192    |  16384    |
|___________|___________|___________|____________|___________|___________|___________|
                                    high-order word

     15          14          13                       2           1           0
 ___________ ___________ ___________ ____________ ___________ ___________ ___________
|     1     |     1     |     1     |            |     1     |     1     |     1     |
|   -----   |   -----   |  ------   |    ...     |---------  |---------  |----------|
|   32768   |   65536   |  131072   |            |268435456  |536870912  |1073741824|
|___________|___________|___________|____________|___________|___________|___________|
                                    low-order word
```

Figure 2-Frac Data Type

Note that for Longints, Fixed, and Frac values, the hex representations of the largest and smallest data values are $7FFFFFFF and $80000000, respectively.

A property of the Fixed and Frac data types is that two Fixed or two Frac values may be added or subtracted just as if they were 32-bit integers. To demonstrate this, imagine scaling the numbers by a given factor to make them integers. After adding the numbers, the sum could be scaled back down by the same factor. This follows from the distributive property of multiplication over addition, which allows one to make the inference shown in the equations which follow. In these equations, V1 and V2 are both either Fixed or Frac values. The value for C being discussed, which illustrates the ability to scale Fixed and Frac values, is 2^16 for Fixed values of V1 and V2, or 2^30 for Frac values of V1 and V2.

$$\frac{(C * V1) + (C * V2)}{C} = \frac{C * (V1 + V2)}{C} = V1 + V2$$

Similarly, two Fixed or two Frac values may be compared, as Longints are compared, with one another. In general, the comparison, addition, and subtraction operations used for long integers may also be performed on any two Fixed or any two Frac values.

Further Reference

- o Apple IIGS Technical Reference Manual
- o Apple Numerics Manual, Second Edition

### END OF FILE TN.IIGS.079

```
#####################################################################
### FILE: TN.IIGS.080
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#80:   QuickDraw II Clipping

Written by: Eric Soldan   March 1990

This Technical Note explains a lot about QuickDraw II operation, specifically clipping.

Before Beginning

Before beginning this Note, some statements, disclaimers, and definitions:

1. This is not a substitute for the QuickDraw II introduction in the Apple IIGS Toolbox Reference, but rather a supplement.
2. A pixelmap is a series of bytes that hold pixel data whose rectangular shape is defined by a LocInfo structure.

This Note describes in great detail the way that QuickDraw II does things with pixelmaps. It begins with a description of the LocInfo structure, which is the most important thing to understand in terms of QuickDraw II pixelmap management. Once this is understood, this Note covers how it applies to using functions such as PPToPort, PaintPixels, and CopyPixels. And once this is understood, it then describes how LocInfo structures are used to control drawing into a grafPort. (PPToPort is used in this Note. PaintPixels and CopyPixels are very close in function to PaintPixels. The information and theory in this Note also apply to these calls.)

Understanding the material in this Note should help you better understand the entire toolbox. It is surprising how much can be accomplished with the toolbox without completely understanding these concepts; it is also surprising how much easier programming with the toolbox gets when these concepts are fully understood.

Note: Structures are written with C syntax in this Note. In addition, this Note uses the screen address 0xE12000L. The possibility of shadowing being active and the screen address being 0x12000L is ignored.

The Beginning

One must begin with the LocInfo structure, which is as follows:

```
struct LocInfo {
       Word          portSCB;              /* SCB in low byte */
       Pointer       ptrToPixImage;        /* ImageRef */
       Word          width;                /* Width */
       Rect          boundsRect;           /* BoundsRect */
};
```

For this Note, one can change this structure a little bit by calling the width element rowBytes. This convention is good because rowBytes is more descriptive than width (it indicates that one is measuring the width in bytes) and it allows one to use the word "width" elsewhere in this Note without confusion. So for the purposes of the Note, the new LocInfo structure definition is as follows:

```
struct LocInfo {
       Word          portSCB;              /* SCB in low byte */
       Pointer       ptrToPixImage;        /* ImageRef */
       Word          rowBytes;             /* Width in bytes*/
       Rect          boundsRect;           /* BoundsRect */
};
```

The ptrToPixImage field is a pointer to some block of bytes in memory. (This block of bytes is referred to as the pixImage from here on.) A pixImage doesn't have any inherent shape. QuickDraw II deals with it as a rectangle, and the LocInfo record defines the rectangularity of it.

When saving a 32,000 byte screen image, one doesn't save the number of bytes of which each row consists. One assumes that each row is 160 bytes by convention, and this is a safe assumption, since the IIGS video hardware expects 160 bytes. But the point is that in the 32,000 bytes of screen data, there is no indicator as to the specific size of a row. One must just know that it is 160 bytes per row. This size is fine for screen shots, but it is not fine when different pixelmaps can be different widths. If they can be different widths, then one also needs some information as to what those widths are, hence the portSCB, rowBytes, and boundsRect fields in a LocInfo structure.

The boundsRect and portSCB fields tell the shape of the pixelmap in pixels, the boundsRect tells how many pixels wide and tall the pixelmap is, and the portSCB tells how big those pixels are (320-mode pixels are four bits wide and 640-mode pixels are two bits wide). One would think that this would be enough information to determine the size of the pixImage, but it isn't. The rowBytes can be larger than the boundsRect/portSCB would indicate (see Figure 1). This situation is legal; it means that some bytes are being wasted, but it is legal.

```
 ___________________________ rowBytes ______________________
|                                                           |
|      boundsRect                                           |
 ___________________________________________ ................
|0,0 .......................................|................
|...........................................|................
|...........................................|................
|...........................................|................
|...........................................|................
|...........................................|................
|...........................................|................
|...........................................|................
|...........................................|................
|...........................................|................
|...........................................|................
|...........................................|................
```

```
|___________________________________________|................
.............................................313,97..........
.......................................................^.....
.......................................................|.....
                                                       |
_____________________________________________          |
 Block of bytes pointed to by ptrToPixImage. \_________|
_____________________________________________/
```

Figure 1-Sample LocInfo Structure

One simply has to know the size of the pixImage, since it cannot be determined by the LocInfo information. If the pixImage is the screen, then it is 32,000 bytes. If it is a fixed or locked handle, then one can do a FindHandle on the pointer followed by a GetHandleSize on the found handle.

Figure 1 represents a sample LocInfo structure. The portSCB (although not pictured) is also relevant, as it determines the size of the pixels. If the pixelmap is a 320-mode pixelmap, one could change it to a 640-mode pixelmap by changing the portSCB to 640 mode and doubling the width of the boundsRect. In doing this conversion, note that rowBytes is not affected and that the pixImage does not change size.

In the example illustrated in Figure 1, the pixImage is bigger than the boundsRect, but again, this is okay. However, this is not the case for the screen, where the rowBytes is 160 and the height of the boundsRect is 200 (the size of the screen is exactly equal to 160 * 200 = 32,000).

There are some rules to determining the rowBytes value. First, rowBytes must not be too small. This is obvious. Second, rowBytes must be evenly divisible by eight. This is not at all obvious, but it is very important. QuickDraw II makes some assumptions for speed, and one of them is that rowBytes is a multiple of eight.

So much for describing the LocInfo structure. Now for how to use it via PPToPort.

PPToPort accepts (among other things) a pointer to a source LocInfo record and a pointer to a source rectangle. PPToPort does not use the source rectangle directly; it first intersects it with the boundsRect in the LocInfo record, and it uses this intersection rectangle instead. This intersection rectangle guarantees that the area involved is completely enclosed by the boundsRect (and therefore within the pixImage). If the source rectangle is entirely outside the boundsRect, then the intersection of the source rectangle and the boundsRect is empty, thus nothing is drawn.

```
 ___________________________ rowBytes ______________________
|                                                           |
|      boundsRect                                           |
 ___________________________________________  ................
|0,0........................................|................
|.............intersection rectangle........|.........sourceRect
|.......____________________________________|_____________________________
|......|50,25 / / / / / / / / / / / / / / / |................             |
|......| / / / / / / / / / / / / / / / / / /|................             |
|......|/ / / / / / / / / / / / / / / / / / |................             |
|......| / / / / / / / / / / / / / / / / / /|................             |
|......|/ / / / / / / / / / / / / / / / / / |................             |
|......| / / / / / / / / / / / / / /  313,77|................             |
|......|____________________________________|_____________________________|
|...........................................|................        523,77
|...........................................|................
|___________________________________________|................
.........................................313,97..............
.......................................................^.....
.......................................................|.....
                                                       |
_____________________________________________          |
 Block of bytes pointed to by ptrToPixImage. \_________|
_____________________________________________/
```

Figure 2-Sample LocInfo Structure With sourceRect

Figure 2 contains a sourceRect which is not completely contained by the boundsRect; the sourceRect is so wide that it even goes beyond the edge of the pixImage. If the entire contents of this rectangle were drawn, the result would be quite a mess, since it extends beyond the boundary of the pixelmap. However, PPToPort first intersects the sourceRect and the boundsRect, and then uses the resulting intersection rectangle (illustrated with a thicker border in the figure). PPToPort uses only the contents of the intersection rectangle.

Up until now, the boundsRect upper-left corner has always been 0,0. This is an easy way to think of it, but it is not necessary. The important thing to remember about these rectangles is their relation to one another. If one were to offset both the boundsRect and sourceRect in this example, the values for the corners of the rectangles would change, but the relationship between the two rectangles would stay the same. Figure 3 illustrates the same example if one were to offset both rectangles by -60,-45.

```
 ___________________________ rowBytes ______________________
|                                                           |
|      boundsRect                                           |
 ___________________________________________ ................
|-60,-45....................................|................
|.............intersection rectangle........|.........sourceRect
|.......____________________________________|_____________________________
|......|-10,-20 / / / / / / / / / / / / / / |................             |
|......| / / / / / / / / / / / / / / / / / /|................             |
|......|/ / / / / / / / / / / / / / / / / / |................             |
|......| / / / / / / / / / / / / / / / / / /|................             |
|......|/ / / / / / / / / / / / / / / / / / |................             |
|......| / / / / / / / / / / / / / /  253,32|................             |
|......|____________________________________|_____________________________|
|...........................................|................        463,32
|...........................................|................
|___________________________________________|................
.........................................253,52..............
.......................................................^.....
.......................................................|.....
                                                       |
_____________________________________________          |
 Block of bytes pointed to by ptrToPixImage. \_________|
_____________________________________________/
```

Figure 3-Sample LocInfo Structure Offset by -60,-45

Notice that the same area of the pixImage is involved, even though the boundsRect and sourceRect are offset. When one offsets both the boundsRect and sourceRect by the same amount, the referenced part of the pixImage does not change--this is an important concept.

Time to ask a question that is answered shortly: "Why isn't the upper-left corner of the boundsRect always 0,0?" Because the LocInfo record isn't always a source LocInfo record. It can also be a destination LocInfo record, and the most common pixelmap to which a destination LocInfo record refers is the screen.

If you had not noticed, the discussion changes gears here--to discuss LocInfo records that indicate a destination pixelmap. Basically, everything is the same as has been described with two exceptions. First, destination pixelmaps do not have a sourceRect. Instead there is a rectangle that describes some portion of the destination pixelmap, and this rectangle is called the portRect. Second, the LocInfo record is part of a grafPort, and each grafPort has a LocInfo record as part of the grafPort data structure.

It is important to remember that a LocInfo record can be used as either a source or destination LocInfo. All a LocInfo record does is define some bytes in memory as a pixImage. Even the screen, which is usually used as a destination pixelmap, can be used as a source pixelmap. There could be situations where one might want to take part of the screen and copy it into some off-screen pixelmap, and in this case, the screen would be a source of pixel data, not a destination.

In the case of the screen pixelmap, there are no wasted bytes in the pixImage, as all of the screen bytes are enclosed by the boundsRect. The screen width of 160 is evenly divisible by eight, so there is no slop at the right edge, and there are no extra rows hanging off the bottom of the boundsRect.

Figure 4 shows a sample LocInfo and portRect (every grafPort has a LocInfo and a portRect).

```
 __________________ rowBytes _______________
|                                           |
|      boundsRect                           |
 ___________________________________________
|0,0........................................|
|.............intersection rectangle.......|         portRect
|.......____________________________________|_____________________________
|......|98,54 / / / / / / / / / / / / / /  |                              |
|......| / / / / / / / / / / / / / / / / / /|                              |
|......|/ / / / / / / / / / / / / / / / / / |                              |
|......| / / / / / / / / / / / / / / / / / /|                              |
|......|/ / / / / / / / / / / / / / / / / / |                              |
|......| / / / / / / / / / / / / / / 640,143|                              |
|......|____________________________________|______________________________|
|...........................................|                         917,143
|.......................................<__|______
|___________________________________________|      |
                                        640,200    |
                                                   |
 ___________________________________________       |
 Block of bytes pointed to by ptrToPixImage. \_____|
 In the case of the screen, this is $E12000. /
```

```
____________________________________________/
```

Figure 4-Sample LocInfo and portRect

Following are two important points to remember:

1. Every grafPort works in local (not global) coordinates (local coordinates are defined soon).
2. The origin of the grafPort is the upper-left corner of the portRect. There is no GetOrigin call; there is a SetOrigin call, but no GetOrigin. To get the origin of a grafPort, one needs to do a GetPortRect call, and then look at the upper-left corner to determine the current origin of the grafPort. This is the way to get the origin.

In the case of Figure 4, local and global coordinate systems are the same, as is always the case when the boundsRect has an upper-left corner of 0,0 (which it seldom does). So, for this exceptional case, one doesn't need a definition of local coordinates. In the global coordinate system, the upper-left corner of the screen is 0,0. In local coordinates, the upper-left corner of the screen is whatever the boundsRect says it is. So when the upper-left corner of the boundsRect is 0,0, the global and local coordinate systems are the same.

In Figure 4, if one tried to draw something to point 0,0, it would not draw--it would be clipped because it is outside the portRect. So even if one tried to draw there, it would not change point 0,0. If a user moved a mouse to that location and an application performed a GetMouse (which returns the mouse location in the local coordinates of the current grafPort), it would return 0,0 as the mouse location.

If one did a SetOrigin(0,0), then the boundsRect and portRect would be offset by the difference between the old and new origins. Both rectangles would be offset, so the relationship between them would remain the same, as Figure 5 illustrates.

```
 __________________ rowBytes _______________
|                                           |
|      boundsRect                           |
 ___________________________________________
|-98,-54....................................|
|............intersection rectangle........|        portRect
|.......____________________________________|_____________________________
|......|0,0 / / / / / / / / / / / / / / / / |                             |
|......| / / / / / / / / / / / / / / / / / /|                             |
|......|/ / / / / / / / / / / / / / / / / / |                             |
|......| / / / / / / / / / / / / / / / / / /|                             |
|......|/ / / / / / / / / / / / / / / / / / |                             |
|......| / / / / / / / / / / / / / /  542,89|                             |
|......|____________________________________|_____________________________|
|...........................................|                             819,89
|.......................................<__|______
|___________________________________________|      |
                                   542,146   |
                                             |
_____________________________________________       |
 Block of bytes pointed to by ptrToPixImage. \_____|
 In the case of the screen, this is $E12000. /
```

```
_____________________________________________/
```

Figure 5-Sample LocInfo and portRect, Both Offset

Now if a user moves a mouse to the upper-left corner of the screen, a call to GetMouse returns a value of -98,-54, as expected, and if a user moves the mouse to the upper-left corner of the portRect, a call to GetMouse returns 0,0, again as expected. This is how origins work and how the conceptual drawing space relates to the grafPort. The boundsRect of the grafPort (in the LocInfo record of the grafPort) and the portRect of the grafPort are offset when one calls SetOrigin. It is that simple.

Now that it is simple, time to complicate matters with one more player in the QuickDraw II clipping world: the visRgn.

The visRgn exists for one purpose: to cause more clipping. It never causes anything to be clipped less than the portRect does, and in the case of a top window that is completely visible, the visRgn and the portRect are exactly the same size. Even more than that, the enclosing rectangle for the visRgn (every region has an enclosing rectangle) is this case would be exactly the same as that of the portRect. This all makes sense when one looks at the purpose of a visRgn. Again, the visRgn can only cause more clipping. If the entire window is visible, one does not want more clipping, so a visRgn the same size as the portRect guarantees that it does not clip any more than the portRect, as it must clip the same amount.

The visRgn is a different size than the portRect when the window is not the top window and part of it is overlapped (or if part of the window is off the screen). The part that is overlapped is excluded from the visRgn, and this excluded part is clipped to protect the window above from being drawn upon. This is how window clipping works. This is all there is to it.

Figure 6 enhances Figure 5 by adding an overlapping window to demonstrate the visRgn.

```
 __________________ rowBytes ________________
|                                            |
|      boundsRect of current grafPort        |
 ____________________________________________
|-98,-54.....................................|
|.............intersection rectangle.........| portRect of current grafPort
|......._____________________________________|_____________________________
|......|0,0 / / / / / / / / / / / / / / / /  |                             |
|......| / / / / / / / / / / / / / / / / / /|                              |
|......|/ / / / / / / / / / / _______________|___________________          |
|......| / / / / / / / / / / |               |                   |         |
|......|/ / / / / / / / / / /|               |                   |         |
|......| /  ^  / / / / / / / |         542,89|                   |         |
|......|____|________________|_______________|___________________|_________|
|...........|................|               |                   |      819,89
|..^........|................|               |                   |
|__|________|________________|_______________|                   |
   |        |                |           542,146                 |
   |        |                |___________________________________|
   |        |                 portRect of some overlapping window
   |        |    _____________________________
   |        |___/ visRgn of current grafPort
   |            \_____________________________
```

```
    |             _____________________________________________
    |____________/ Block of bytes pointed to by ptrToPixImage.
                 \ In the case of the screen, this is $E12000.
                  \_____________________________________________
```

Figure 6-Sample LocInfo and portRect With Overlapping Window

What happens to the visRgn during a SetOrigin? Remember that the boundsRect and portRect get offset. The visRgn does too. Again, if all of these elements are offset together, then the relationship between them remains the same; they stay the same, relative to one another. (For more information, see Einstein's theory of general relativity.)

The final component for clipping is the clipRgn, which is the application's property and, therefore, the application's responsibility. The system sets the clipRgn about as big as it can get to start (much bigger than the portRect); this is often referred to as arbitrarily large, even though it isn't so arbitrary. The system creates all grafPort structures with a large clipRgn, and this can be a problem for certain types of QuickDraw II operations. Since the clipRgn already reaches to the borders of the conceptual drawing space, it cannot be offset; it is effectively stuck, due to its size. It is a good practice to make the clipRgn smaller than the system default.

SetOrigin does not offset the clipRgn. (This is why the size problem with a big clipRgn is not so apparent.) The clipRgn is the only clipping component that is not offset by SetOrigin, and one should consider this when using clipRgn for clipping effects, since an application must remember to offset it if it needs to be offset.

Now with all of the fundamentals out of the way, it is time to play some grafPort clipping games. As a refresher, there are four clipping components in a grafPort: the boundsRect, the portRect, the visRgn, and the clipRgn.

If an application creates its own off-screen grafPort structures, then it can do as it wishes with all four clipping components. After all, if it has the responsibility to set them up in the first place, it should have the right to change them. If, however, the Window Manager creates the grafPort structures, then an application should keeps its figurative hands off certain clipping components, namely the boundsRect and the visRgn. The clipRgn, by definition, is the application's to do with as it sees fit, and if careful, an application can also change the portRect. Changing the portRect can be very useful, but one needs to be careful and fully understand all of the ramifications.

So, why would one change the portRect, and how would one do it?

Another figure is in order.

```
 __________________ rowBytes _______________
|                                           |
|      boundsRect                           |
 ___________________________________________
|-98,-54....................................|
|.............intersection rectangle........|         portRect
|......_____________________________________|_____________________________
|......|0,0 / /|100,0 / / / / / / / / / / / |                             |
|......| / / / | / / / / / / / / / / / / / /|                             |
|......|/ / / /|/ / / / / / / / / / / / / / |                             |
|......| / / / | / / / / / / / / / / / / / / /|                             |
|......|/ / / /|/ / / / / / / / / / / / / / / |                             |
|......| / / / | / / / / / / / / / / /  542,89|                             |
|......|_______|_____________________________|_____________________________|
|............................................|                                  819,89
|.........................................<__|______
|____________________________________________|      |
                                          542,146   |
                                                    |
______________________________________________      |
 Block of bytes pointed to by ptrToPixImage. \_____|
 In the case of the screen, this is $E12000. /
____________________________________________/
```

Figure 7-Sample LocInfo and Modified portRect

One can use the GetPortRect call to get the portRect for the current grafPort. One can then modify it, and then use the SetPortRect call to inform the grafPort about the change. Why do this? In Figure 7, the dotted line represents the new left edge of the portRect after the modification (a simple modification of adding 100 to the old value of zero).

Note that changing the portRect in this way changes the relationship between the portRect and the boundsRect. Anything drawn from 0 to 99 (x coordinate) is clipped, since it is outside the new (modified) portRect. Before the modification, anything drawn from 0 to 99 would have affected the screen.

This modification may cause the portRect to be smaller than the visRgn. This is okay, since the visRgn can only cause more clipping, not less. So, all of this works just fine. Note that the origin changed when the left edge of the portRect changed. The upper-left corner of the portRect is always the origin, and an application changed it. The origin changed without a SetOrigin call. (Scary, huh?)

One could have done exactly the same thing by making a clipRgn to exclude the x coordinates from 0 to 99. However, here is something cool. After the modification, do a SetOrigin(0,0), which sets the upper-left corner of the shrunk portRect to 0,0. One cannot accomplish this sort of thing as simply by making a clipRgn. One can effectively move where an origin of 0,0 is the screen, and just building a clipRgn to exclude some part of the screen does not accomplish this.

Why would one want to change where 0,0 is on the screen? This sort of trick is very useful for adding rulers to a document window, for example. One of the problems with rulers is that they should not scroll with the rest of a document. Unfortunately, TaskMaster, if allowed to handle scrolling, doesn't know about a ruler at the top of a window and scrolls it with the rest of the window's content area. By changing the portRect so that the ruler is not inside of it, one can keep TaskMaster from scrolling it. In a draw procedure, when it is necessary to draw the ruler, grow the portRect, set the origin to 0,0, and then draw the ruler. Once it is drawn, set the portRect back to the smaller size to protect the ruler again.

Another reason one might want to do this is if an application uses a split window (where the top of the window may show a different part of the document than the bottom). Changing the portRect has the advantage that the upper-left corner of the portRect is always the origin, so it makes mapping document coordinates easier.

Another advantage to using the portRect in this way is that it keeps the clipRgn free for other purposes. Being able to separate types of clipping to either the portRect or the clipRgn keeps the clipRgn from being overused.

As a final note, it should be observed that the only clipping that is done is on a destination pixelmap. There is no clipping on a source pixelmap. There is no need. All the clipping needed is done at the destination end, so it would be wasteful to clip twice.

This finishes the discussion about QuickDraw II and how the boundsRect, portRect, visRgn, and clipRgn work together to accomplish clipping. Hopefully this Note answers more questions than it creates.

Further Reference

- Apple IIGS Toolbox Reference, Volume 2
- Relativity the Special and General Theory (1920)

### END OF FILE TN.IIGS.080

```
#####################################################################
### FILE: TN.IIGS.081
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#81: Extended Control Ecstasy

Revised by: Dave Lyons July 1991
Written by: C.K. Haun May 1990

This Technical Note discusses special features and concerns that should be considered when using the extended controls introduced in System Software 5.0.

Changes since November 1990: Added information on which fields the Control Manager automatically copies from a custom control's template to its control record. Corrected NewControl2 parameter order. Added a note about SetCtlTitle. Changed some "pea 0" instructions into "pha" when pushing space for results.

Introduction

The extended controls introduced in System Software 5.0 allow the application programmer a great deal more freedom in designing and controlling applications. The new features enhance the functionality of the controls and TaskMaster, but can cause confusion and consternation if you are careless with the new parameter block . This Note also includes a discussion of the multipart nature of many of the extended controls and some pointers for writing extended custom controls.

Counting The Costs

One of the major stumbling blocks seen when programming the new extended controls is bad parameter counts. Extended controls introduce parameter blocks and parameter counts to the Control Manager. You need to fully understand the parameters required and the resulting parameter count for each control you create, or you can experience program problems that may be very confusing and difficult to track down.

Remember also that the Control Manager does not understand "skipping parameters." If you are creating an extended radio button and you want key equivalents, but not a color table, you cannot ignore the color table parameter field. There is no way to tell the Control Manager to "skip" a field during control creation; make sure you initialize all the fields to values that are meaningful--either a real pointer, handle, resource ID, or zeroes. In this case, if you try to "skip" the color table and do not zero out the color table parameter field, the resulting radio button wears ugly colors you do not expect.

As you can imagine, miscounting parameters in other extended controls can produce confusing results, so the parameter count is the first place you should check when you're having difficulties creating extended controls.

For Rez users, the Types.Rez file contains extended control templates that automatically generate the correct count value, so programmers creating all their controls with the Resource Compiler should not have these problems.

Silly Little Bits

The other area of the new parameter block model that is giving folks trouble is the moreflags field. This field hones the definition given by the reference fields and can cause you much grief if misused. Make sure to set the reference bits to the values you require. The bit settings have been standardized across all the extended controls, with %00 indicating a pointer, %01 indicating a handle, and %10 indicating a resource. Remember also to set the bits for all the references you have--strings, color tables, or whatever else may be ambiguously referenced in the control.

If you accidentally use the wrong bit pattern, you can experience strange bugs ranging from garbled text to SysFailMgr caused by a nonexistent resource being referenced. Again, Rez users can use the equates for all the reference specifiers in the Types.Rez file to avoid confusion and evil bugs.

The Parts are Greater than the Whole

To create some new extended controls, like pop-up menu controls and LineEdit controls, functions of different tool sets were combined. Pop-up menu controls are a combination of the Control Manager and the Menu Manager, LineEdit controls are a blending of the Control Manager and the LineEdit tool set, and other new control types follow the same pattern.

This means that, at times, you have to go further into the documentation to find information. Getting the text out of an LineEdit extended control is a multistep process that is a good example of this type of problem.

```
MyLineEdit dc     i2'8'                   ; parameter count
           dc     i4'1'                   ; id number 1
           dc     i2'10,10,23,90'         ; control rectangle
           dc     i4'editLineControl'     ; process reference
           dc     i2'0'                   ; flags
           dc     i2'fCtlCanBeTarget+fCtlWantEvents+fCtlProcRefNotPtr'
                                          ; moreflags
           dc     i2'0'                   ; refcon
           dc     i2'15'                  ; maximum characters allowed
           dc     i4'0'                   ; no default text
MyLineEditHandle ds 4                     ; handle for the created control

           pha
           pha
           pushlong mywindowgrafport      ;window that control will residein
           pea    0                       ;verb, single Extended control
           pushlong #MyLineEdit
           _NewControl2
           pla
           sta    MyLineEditHandle        ; save the control handle
           pla
           sta    MyLineEditHandle+2
```

When you want to get the text back out of that control later, you begin to experience what it means to have a control that is an amalgam of various tools. You would start by using the control handle returned by NewControl2:

```
Scratch    equ    $0                   ; some scratch space
Scratch2   equ    $4

           lda    MyLineEditHandle     ; move the ControlHandle to
           sta    Scratch              ; some direct page space
           lda    MyLineEditHandle+2
           sta    Scratch+2
           lda    [Scratch]
           tax
           ldy    #2
           lda    [Scratch],y          ; and dereference it, putting it
                                       ; back in some dpage
           sta    Scratch+2            ; space to use it.
           stx    Scratch
```

That gives you the pointer to the control record. Stored in the control record is the handle of the LineEdit item that is actually controlling the text processing:

```
           pha                         ; make space for the text handle to
                                       ; be returned
           pha
           ldy    #octlData            ; offset to the ctlData section
                                       ; of the ControlRecord
           lda    [Scratch],y          ; where the handle for the actual
           tax                         ; LineEdit item was stored
           iny
           iny
           lda    [Scratch],y
           pha
           phx
           _LEGetTextHand              ; ask for the handle for the text
           pla                         ; in this LineEdit control
           sta    Scratch2             ; and now you have the handle to
                                       ; the text you want.
           pla
           sta    Scratch2+2
```

The main point is that when you are using extended controls, you often cannot use the Control Manager to do everything that needs to be done. You also need to understand and use the supplementary or "hidden" tool sets.

Here's another example, using a pop-up menu extended control, and in this case we define a font pop-up that contains all the font names currently available.

```
MyPopUpControl dc i2'9'                ; parameter count of 9
           dc    i4'1'                 ; control ID of 1
           dc    i2'2,2,0,0'           ; Position, upper left corner of
                                       ; the window, let
                                       ; Control Manager calculate full
                                       ; size
           dc    i4'popUpControl'         ; def proc for PopUp
           dc    i2'0'                    ; flags
           dc    i2'fCtlWantEvents+fCtlProcRefNotPtr'
                                          ; more flags
           dc    i4'0'                    ; ref con
           dc    i2'0'                    ; title width, will be calculated
           dc    i4'mymenu'               ; pointer to actual menu structure
           dc    i2'500'                  ; initial value, item number of
                                          ; item to be displayed in popup at
                                          ; creation


mymenu     dc    i2'0'                    ; version number, should be 0
           dc    i2'200'                  ; menu ID number
           dc    i2'0'                    ; menu flags
           dc    i4'mymenutitle'          ; pointer to menu title
           dc    i4'mymenuitem1'          ; first menu item
           dc    i4'mymenuitem2'          ; second menu item
           dc    i4'0'                    ; null terminator, end of menu
mymenutitle str 'Font'

mymenuitem1 dc i2'0'                      ; version number
           dc i2'500'                     ; item number
           dc i2'0'                       ; no hot keys
           dc i2'0'                       ; not checked
           dc i2'0'                       ; item flags, no special drawing
           dc i4'mymenuitem1title'
mymenuitem1title str 'Plain'

mymenuitem2 dc i2'0'                      ; version number
           dc i2'501'                     ; item number
           dc i2'0'                       ; no hot keys
           dc i2'0'                       ; not checked
           dc i2'1'                       ; item flags, bold face this one
           dc i4'mymenuitem2title'
mymenuitem2title str 'Bold'
```

Now create this control:

```
           pha
           pha
           pushlong mywindow              ; target window grafptr
           pea    0                       ; verb, single control pointer
           pushlong #MyPopUpMenu
           _NewControl2
           pulllong mypopuphandle         ; save the handle
```

This pop-up menu control created is not associated with the menu bar across the top of the desktop. You can consider each of your pop-up menu controls as separate menu bars, so if you want to perform Menu Manager calls on a pop-up menu control, you need to set the menu to point at your pop-up menu control. In this example, to add all the fonts available to the pop-up menu you would:

```
           pha
           pha                            ; space to hold current bar
           _GetMenuBar                    ; get the handle to the current
                                          ; menu bar
```

```
            pushlong mypopuphandle
            _SetMenuBar
            pea    200                     ; id number of this menu
            pea    502                     ; first font family ID number to
                                           ; use
            pea    0                       ; fontspecbits
            _FixFontMenu
            pea    0
            pea    0
            pea    200
            _CalcMenuSize                  ; re-size the popup menu
            _SetMenuBar                    ; restore the previous menu as the
                                           ; current menu
```

Controls That Are Not Controls

The new picture extended control is not a "full-fledged" control; it has been provided to simplify your programming tasks. The picture control does not support normal mouse hit testing and highlighting. Think of it as a built-in extension to your content drawing routine, and not as a control. It is provided to allow you to refresh your whole window with a single DrawControls call, instead of drawing the controls and then drawing pictures. The icon button extended control has been provided as the graphic full-function control. If you need or want a fully functional control that uses a picture, you should consider writing your own custom control procedure.

Custom Extended Controls

Custom controls can also benefit from all the advantages of extended controls. You can create a custom control that uses a template, can be a resource, has a definition procedure that is a resource, and responds to all the new control calls. If you write an extended custom control or upgrade a previously-written custom control, there are new messages and changes to existing messages of which you need to be aware. These changes are documented in volume 3 of the Apple IIgs Toolbox Reference.

The Control Manager copies the following fields from the control template to the control record before it sends your control the init message: ctlOwner, ctlID, ctlRect, ctlFlag, ctlHilite, ctlMoreFlags, ctlVersion, ctlRefCon, and ctlProc. The ctlNext field is owned by the Control Manager. If any additional fields need to be set up based on the control template (such as ctlValue, ctlData, ctlColor, and any custom fields), your init routine needs to take care of it.

Putting your custom control definition procedure in a resource can significantly enhance the functionality of the custom control. You may find it easier to add to all of your programs and you do not have to manage the code space required. If you do write a custom control definition procedure and want to store it as a resource, here are some hints for success.

First, the code you store in your resource fork must be fully compiled and linked code. The code resource converter uses the System Loader to load the code, so the code must be executable code, not object code.

Second, set the convert and locked bits of the resource attributes for your code resource. The convert bit must be set to tell the Resource Manager to call the code resource converter when it loads this resource. The resource type for control definition procedures is rCtlDefProc, $800C.

By setting locked but not fixed, memory fragmentation is reduced (because of how the code resource converter and Memory Manager work). Setting the locked attribute is also recommended for compatibility with future system software.

Third, keep in mind that this definition procedure may be purged and reloaded whenever the Memory Manager needs the space. This means that you cannot store any information in your definition procedure if you want to keep track of it between calls to the definition procedure. If you do, and your definition procedure gets purged and reloaded, you lose that data.

If you need data space for your custom control, use the control record as your stash. You can easily either use the fields already provided in the control record, or you can expand the control record to as much space as you need (within sensible limits) and store your data there.

Warning: Control definition procedures are initially loaded with purge level zero. When they are released, they are given purge level three. If they are then reloaded, the Resource Manager does not change the purge level back to zero--your definition procedure may then be purged (even while executing) unless its handle is locked. The solution is to lock your definition procedure handle within the procedure:

```
myPosition    pea    0                ; space for result
              pea    0
              pushLong #myPosition
              _FindHandle
              _HLock
```

and unlock your handle with HUnlock on exit. This keeps your procedure safe, while not creating "code islands," which clog up memory.

Changing a Control's Title

If you call SetCtlTitle to give a control a new title, everything is great if the new title is referenced the same way as the current title (by pointer, by handle, or by resource ID). If the new title is referenced differently, you must first call SetCtlMoreFlags on your control so that the SetCtlTitle value can be interpreted correctly.

Conclusion

The extended controls provided in System Software 5.0 and later are a great leap forward for programmers. They relieve the application of much of the tedious detail code that relates to housekeeping, not the guts of application programming. Used in combination with the enhanced TaskMaster, you can have an application's visual interface up and running a lot faster, leaving you more time to work on the heart of your application.

Further Reference

- Apple IIgs Toolbox Reference, Volumes 1-3.

### END OF FILE TN.IIGS.081

```
######################################################################
### FILE: TN.IIGS.082
######################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs #82: Controlling the Control Manager

Revised by: Matt Deatherage November 1990
Written by: Dave Lyons May 1990

This Technical Note describes an anomaly in the NewControl2 call in System Software 5.0.2 and provides a solution.
Changes since May 1990: Noted that System Software 5.0.3 fixes this anomaly.

This Note formerly advised of a problem with NewControl2-the current port was not set before adding the controls, which gave unpredictable results. System Software 5.0.3 and later fix this problem.

### END OF FILE TN.IIGS.082

```
#####################################################################
### FILE: TN.IIGS.083
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#83: Resource Manager Stuff

Revised by: Matt "Even less of a middle name" Deatherage May 1992
Written by: Dave Lyons May 1990

This Technical Note answers your miscellaneous Resource Manager questions.

CHANGES SINCE DECEMBER 1991: Added several notes pertaining to System Software 6.0 and a note about making Resource Manager calls from a resource converter. Added new discussion about how "changed" is really a resource attribute.

UNIQUERESOURCEID

In System Software 5.0.4 and earlier, calling UniqueResourceID with an IDRange value of $FFFF does not work reliably. It sometimes returns a system-range ID ($07FFxxxx) if there are already system-range resources of the specified type present in the current search path.

If you are using a development utility that generates resource IDs using UniqueResourceID, check the results to make sure no system-range resource IDs are being used by accident. This problem is fixed in System Software 6.0.

WHAT SETCURRESOURCEFILE DOES

SetCurResourceFile is documented in Chapter 45 of the Apple IIgs Toolbox Reference, Volume 3 (see especially "Resource File Search Sequence" near the beginning of the chapter).

This explanation might make you think SetCurResourceFile rearranges the search path, but it does not; instead, it just makes searches start at a different place in the path. SetCurResourceFile is useful for controlling what resource files are searched, not for changing the search order.

HOW THE TOOLBOX USES RESOURCES AS TEMPLATES

The toolbox uses several types of resources as templates for creating other objects. Examples include rControlList, rControlTemplate, and rWindParam1. The toolbox automatically releases these resources from memory as soon as it is through with them, so there is no need to create your template resources with special purge levels in an effort to free more memory. It is not a problem.

USING RESOURCES FROM WINDOW UPDATE ROUTINES

In System Software 6.0 and earlier there is no special code to set the current resource application when the system calls an application window update routine (See Apple IIgs Technical Note #71 for notes on NDAs and the current resource application).

To avoid a situation where a window update routine cannot get needed resources, obey the following rules:

1. Application window update routines must either (a) assume that the resource application has the same value it had when the window was created, or (b) save, set, and restore the current resource application, using GetCurResourceApp and SetCurResourceApp.

2. NDAs that start the Resource Manager must not call application window update routines, and they must not cause application window update routines to be called (for example, if an NDA calls TaskMaster to handle a modal dialog or movable modal dialog, the tmUpdate bit in wmTaskMask must be off).

CURRESOURCEAPP IN INFODEFPROCS AND CUSTOM WINDOWS

The current resource application has no guaranteed value when an information bar definition procedure or custom window definition procedure gets control. These must always save, set, and restore the current resource application using GetCurResourceApp and SetCurResourceApp.

STARTUPTOOLS OPENS RESOURCE FORKS READ-ONLY

When StartUpTools opens your application's resource fork, by default it opens it with read-only access. If your application needs to make changes to the resources on disk in System Software 5.0.4 and earlier, you need to close the fork and reopen it with read and write access. To close it, use GetCurResourceFile and CloseResourceFile; to reopen it, use LGetPathname2 and OpenResourceFile.

Note : You must update the resFileID field in the StartStop record if you close and reopen your resource fork. CloseResourceFile disposes the handles of any resources in memory from the file you're closing, so you must call DetachResource on any resources you need to keep. (If you pass an rToolSTartup resource to StartUpTools, the system detaches it for you automatically.)

In System Software 6.0 and later, setting bit 3 ($0008) of the startStopRefDesc tells the Tool Locator to open your resource fork with all allowed permissions instead of with just read permission.

CALLING STARTUPTOOLS FROM A SHELL APPLICATION (FILE TYPE $B5, EXE)

In System Software 5.0.4 and earlier, StartUpTools tries to open the current application's resource fork. It determines the pathname of the "current application" by examining prefix 9: and making a GET_NAME GS/OS call, but do not assume it will always construct the pathname this way. If you call StartUpTools from a shell application and expect it to open your EXE file's resource fork, you will be disappointed.

If GS/OS has launched your application, life is good--usually, though, a shell has loaded your shell application directly, so GET_NAME returns the name of the shell instead of the name of your application file.

To open your shell file's resource fork, call ResourceStartUp, get the pathname by calling LGetPathname2 on your user ID, and pass the pathname to OpenResourceFile. StartUpTools uses this strategy all the time in System Software 6.0 and later, meaning you don't have to.

WHAT'S NIL IN A RESOURCE MAP?

The resource maps for open resource files are kept in memory, and the structure is defined in chapter 45 of Apple IIgs Toolbox Reference, Volume 3.

The resHandle field of a resource reference record (ResRefRec) is defined as "Handle of resource in memory. A NIL value indicates that the resource has not been loaded into memory." In this case, NIL means that the middle two bytes of the four-byte field are zero. In other words, a NIL entry in the resource map may have a non-zero value in the low-order byte.

LOADRESOURCE AND SETRESLOAD(FALSE)

When you call LoadResource on a locked or fixed resource and SetResLoad is set to FALSE, you may get Memory Manager error $0204 (lockErr), because the Resource Manager tries to allocate a locked or fixed zero-length handle, which the Memory Manager does not permit.

ADJUSTING THE SEARCH DEPTH

If you wish to add some resource files to the beginning of a resource search path and adjust the depth so that the end point of the search is unchanged, it's tempting to use SetResourceFileDepth(0) to get the current depth, add one, and set this new depth with SetResourceFileDepth.

The problem is that the search depth is often -1 ($FFFF), meaning "search until the end of the chain." If you add your adjustment to -1, you do not usually get the intended effect. A solution is just to check for $FFFF and not adjust the depth in that case.

CURRESOURCEAPP AFTER RESOURCESHUTDOWN

After a ResourceShutDown call, the current resource application is always $401E. (The Resource Manager starts itself up at boot time with its own memory ID, $401E. Do not ever call ResourceShutDown while the current resource application is $401E.)

RESTORING THE CURRESOURCEAPP

If you need to start up and shut down the Resource Manager without disturbing the current resource application, call GetCurResourceApp before ResourceStartUp, and call SetCurResourceApp to restore the old value after ResourceShutDown.

It does not help to call GetCurResourceApp after ResourceStartUp, since the application just started up is always the current resource application.

Shell programs which start the Resource Manager need to call SetCurResourceApp after regaining control from a subprogram (for example, an EXE file) which may have started and shut down the Resource Manager, leaving the current resource application set to $401E instead of the shell's ID.

Shell programs that do not start the Resource Manager have nothing to worry about. In this case the current resource application is normally $401E, so when a subprogram calls ResourceShutDown life is still wonderful.

WHAT INFORMATION IS KEPT FOR EACH RESOURCE APPLICATION?

When you switch resource applications with SetCurResourceApp, that takes care of all the application-specific information the Resource Manager has.

There is no need to separately preserve the current resource file, the search depth, the SetResourceLoad setting, or any application resource converters that are logged in. All of this information is already recorded separately for each resource application.

"CHANGED" IS A RESOURCE ATTRIBUTE

This seems obvious when first reading the documentation, but it has a consequence that isn't so obvious.

If you mark a resource as changed with MarkResourceChanged and later use SetResourceAttr to change that resource's attributes, you must include resChanged in the attributes you specify or the Resource Manager does not still know the resource has changed.

This means you can undo a MarkResourceChanged call, but it also means you need to preserve the resChanged bit across SetResourceAttr calls if you don't want to accidentally achieve the same effect.

The Resource Manager clears the resChanged attribute when a resource is written to disk; the attribute indicates the data in memory is more recent than what's on disk. Normally, adding a resource with AddResource sets this bit because the resource isn't actually written to disk until the resource file is updated.

However, if AddResource has to make the file longer (by extending the EOF), it writes the resource to disk immediately. This means that in some cases, a resource added with AddResource will be properly added but the resChanged attribute will not be set. Don't be confused if this happens to you.

MAKING RESOURCE MANAGER CALLS FROM RESOURCE CONVERTERS

Don't. This would be a first-class example of reentrancy, and the Resource Manager is not reentrant in any class.

WHO OWNS HANDLES PASSED TO ADDRESOURCE?

When you pass a handle to AddResource, the Resource Manager is responsible for the handle unless AddResource returns an error. Once you call AddResource, the handle belongs to the Resource Manager and you must treat it like you would the handle to any other resource.

NAMED RESOURCE BUGS IN SYSTEM SOFTWARE 6.0

The new-for-6.0 Resource Manager function RMFindNamedResource compares the resource name you requested to named resources incorrectly. The comparison algorithm doesn't compare the lengths of the strings before starting to compare the characters. This means, for example, that if you request a resource named "Raymond" and the Resource Manager encounters a named resource named "Raymond" first, it will return the resource named "Raymond" instead. This anomaly also affects the HyperCard IIgs named-resource XCMD callback functions, even though they don't use the Resource Manager's named-resource calls.

This anomaly also affects RMLoadNamedResource, which calls RMFindNamedResource.

DEBUGGING INFORMATION

The following information is provided for your convenience during program development. It allows you to check exactly what user IDs are using the Resource Manager, what files are in their search paths, and what resource converters are logged in.

DO NOT depend on this information in your program; it is subject to change in future versions of the Resource Manager.

All the Resource Manager's data structures are rooted in the Resource Manager tool set's Work Area Pointer (WAP). To get the Resource Manager's WAP, call GetWAP (in the Tool Locator) with userOrSystem = $0000 and tsNum = $001E.

The WAP value is a handle to the Resource Manager's block of global data. Several interesting areas in this block are listed below.

+$0A2	curApp	Word	Offset into the globals block of the current resource application's Application Record.
+$2B0	sysFile	Long	Handle of system file map, or NIL if none.
+$2B4	sysConvertList	Long	Handle of system converter list, or NIL if none.
+$2B8	appList	20*n bytes	List of Application Records (20 bytes each).

Each Application Record has this format:

+000	appFlag	Word	0=entry available, 1=entry used, $FFFF = end of array.
+002	appID	Word	User ID of application.
+004	appFiles	Long	Handle of application's first resource map, NIL=none.
+008	appCur	Long	Handle of application's current resource map, NIL=none.
+012	appConverters	Long	Handle of application's converter list, NIL=none.

```
   +016   appReadFlag    Word   1=read resources, 0=don't read
                                (SetResourceLoad).
   +018   appFileDepth   Word   Number of files to search in this path.
```

Converter lists have this format:

```
   +000    n            Word        Number of entries in the table (entries
                                    can be unused).
   +002   theConverters 6*n bytes   List of converter entries (6 bytes each).
```

Each Converter entry has this format:

```
   +000  resType      Word     Resource type for this converter ($0000 for
                               unused entry).
   +002  convAddress  Long     Address of resource converter.
```

The format for a resource map is described starting on page 45-17 of Apple IIgs Toolbox Reference, Volume 3.

Remember, don't depend on this information in your application; use it during debugging, and use it to write debugging utilities.

Further Reference

- Apple IIgs Toolbox Reference, Volume 3
- Apple IIgs Technical Note #71, DA Tips and Techniques

### END OF FILE TN.IIGS.083

###
### FILE: TN.IIGS.084
###

Apple II
Technical Notes

Developer Technical Support

Apple IIgs #84: TaskMaster Madness

Written by: C.K. Haun <TR> July 1990

This Technical Note discusses the enhancements made to TaskMaster in System Software 5.0.

TaskMaster has been expanded to handle extended control actions and give you more information about events in System Software 5.0. This Note discusses some features of the expanded TaskMaster and TaskMasterDA, and how you can best exploit the new features in your applications.

Stop Making It So Difficult

Developers just want to work too hard. You get a neat new thing like the expanded TaskMaster, and you still want to do all the work yourself. The new TaskMaster does nearly everything for you, as long as you treat it correctly.

What this means is you do not have to call FindControl, TrackControl, TEIdle, LEKey, handle keystrokes for controls, keep track of click counts, or any of the other mundane event management tasks unless you specifically want to perform actions that TaskMaster does not perform. For the standard controls and situations this means that you do not have to do anything.

The magic keys to this life of freedom and ease are the five newly defined taskMask flag bits, labeled in the interfaces as tmContentControls, tmControlKey, tmControlMenu, tmMultiClick and tmIdleEvents. This Note looks at what the new bits do for you, but first a word of warning.

Warning: If you set any of these new bits, TaskMaster assumes you are using the new extended task record. This means that you cannot just go into an older program and set these bits and expect your program to work successfully. You also must allocate the additional space for the extended portion of the task record. If you do not, TaskMaster puts task data in areas that you do not expect, and Bad Things happen.

Bits 'o This, Bits 'o That

Click Bits

tmMultiClick tells TaskMaster to keep the new "click information" fields in the extended task record updated. This allows you to have TaskMaster keep track of multiclick events; the wmClickCount field is one, two or three depending on whether the last action was a single, double, or triple click. In fact, if you can click your mouse button fast enough, you can time quadruple clicks, sextuple clicks, or as high as you want, although anything over triple-clicking is nearly impossible for users to consistently manage. wmClickCount just gets incremented by one when the click falls within the double time interval. wmLastClickTick is updated with the system tick value at last click. wmLastClickPt contains the location of the last mouse click. TaskMaster calls GetDblTime internally to determine the correct time intervals for these values.

Idle Bits

tmIdleEvents tells TaskMaster to call the idle routines for controls that need idle events, like TextEdit controls and LineEdit controls. This also means that only the active control is blinking a cursor, since TaskMaster is working with the target bits of the extended control records to keep track of which TextEdit or LineEdit control is active and switching the target control in response to mouse clicks and Tab keypresses. This is also the area where you tell TaskMaster how to highlight your window controls. Using the Control Manager calls MakeNextCtlTarget and MakeThisCtlTarget allows you to specify which LineEdit or TextEdit control is active. You can use these calls to highlight input errors the user has made. For example, if someone has entered text in a LineEdit control that requires a number, you can alert the user if he enters non-numeric characters with an Alert or AlertWindow call. You can then direct the user to the LineEdit control that contains the bad entry by calling MakeThisCtlTarget with the handle of that LineEdit control. This deactivates any other target control and moves the insertion point to the LineEdit control that needs the correction.

Contentious Bits

tmContentControls, tmControlMenu and tmControlKey bits are the real workhorses of the expanded TaskMaster.

When the tmContentControls and tmControlMenu bits are set, TaskMaster handles the mouse activity side of events--tracking, highlighting or popping-up the selected control. If the control is a radio button, check box, pop-up menu or list control, TaskMaster also performs the correct action for the click, either setting the control value, scrolling the list, setting the pop-up menu to the selected item, and so on. TaskMaster then returns a taskCode of wInControl ($21). The control handle is stored in wmTaskData2, the part code of the part selected in wmTaskData3 and the control ID is in wmTaskData4. For many of the controls in your windows your application needs to take no further actions, TaskMaster has set the control values. When the user closes the window or clicks on a button that causes an action, you can then read the values of all the controls you care about at that point and do what you need to do, instead of keeping track as the user manipulates controls.

The last new bit, tmControlKey, works with the tmControlMenu bit to handle key events for your extended controls.

When a key event occurs, TaskMaster sends the event to the internal routine TaskMasterKey. TaskMasterKey first looks at the tmMenuKey bit (which has been in TaskMaster since the Window Manager was implemented). If it is set, then TaskMaster tries to handle the event as a menu event, calling MenuKey for the current menu bar.

Note: This also means that any key equivalents in your main menu bar (across the top of the desktop) take precedence over key equivalents in your window controls.

If this fails (or that bit is not set) and tmControlKey is set, then TaskMasterKey polls the controls in the currently open window for any controls that would like this keystroke, either for controls with a keyEquivalent field or a pop-up menu control with key equivalents for menu items. If it finds a control that wants the key event, it is handled very much like a mouse event. The action for the control is performed (checking a check box, for example) and the wmTaskData fields are filled as they would be for a mouse click, and an event code of wInControl ($21) is returned. If a key event did occur, you can differentiate it from a mouse event by looking at the wmWhat field of the taskRecord. Even though a wInControl event code was passed back by TaskMaster, the wmWhat field is either $0001 or $0003, the former for a mouse down event and the latter if a keystroke stimulated the wInControl event.

Even More Bits

All these new features rely very heavily on the changes made to the Control Manager in System Software 5.0. Many of the TaskMaster features, keystrokes, target controls, and so on only work if you have the moreFlags bits set correctly in your control definitions. If you are having difficulty with new TaskMaster features, check your control definitions against the information in the Control Manager chapter of Volume 3 of the Apple IIgs Toolbox Reference and Apple IIgs Technical Note #81, Extended Control Ecstasy.

Don't Get Goofy

There are some dangers in these new features, of course. By allowing built-in key equivalencies for almost all the controls that can exist in a window, it may be tempting to define key equivalents for everything, and create weird and unusual key combinations for your controls. Please remember the Human Interface Guidelines (specifically Human Interface Note #8, Keyboard Equivalents) and keep your use of keystroke equivalents to a minimum. Multimodifier keystrokes (Command-Option-Shift, for example) do not enhance the user's experience and can be very confusing.

NDAs Can Have Fun Too

TaskMasterDA has also been added to the Window Manager, providing your new desk accessories (NDAs) with the same kind of TaskMaster support your applications have. This lets you easily use extended controls inside NDAs, following the same basic rules as in an application. There are only a few things to worry about.

What Does That Stack Picture Really Mean?

The input to TaskMasterDA, as shown in Volume 3 of the Apple IIgs Toolbox Reference, is as follows:

```
        |_____________|
        |             |
        |    space    |  word, return space
        |_____________|
        |             |
        |  eventMask  |  word, eventMask, ignore
        |_____________|
        |             |
        |             |
        | taskRecord  |  long, pointer to taskRecord
        |   pointer   |
        |             |
        |_____________|
```

Figure 1-TaskMasterDA Stack Picture

The call returns a word value, the taskCode. The space and eventMask are self-explanatory. The book tells you that the eventMask is ignored, which makes sense since the host application has already gotten the event and you have already specified an eventMask in your NDA header, so you can use any value here. The taskRecPointer causes the confusion.

You do not pass a blank event record. When your NDA's action routine is called, the Y and X registers contain a pointer to the current event record with which the NDA is working. TaskMasterDA is filling that taskRecord with some information, so you want to move it into your NDA's data area so you can work with it later:

```
    phy
    phx                  ; push the pointer that was passed to us
    pushlong  #NDArecord  ; the space in my NDA for the extended event record
    pea       0
    pea       16          ; only 16 bytes, the original taskRecord size
    _BlockMove
```

It is very important that you only move 16 bytes. TaskMasterDA can act erratically if the extended portion of the event record has been filled with nonsense values. This can happen if your NDA is running in an application that does not use the extended task record and you are copying non-task data into the extended portion of the task record. By the way, as you are debugging your NDA and you run into situations where the wmTaskData field values are weird, this is more than likely the problem.

Also remember to make sure the wmTaskMask field in your NDA's TaskRecord is set and the extended portions of the TaskRecord are zeroed out before your NDA starts running; you want to set all these fields in your NDA's INIT routine.

Now you can call TaskMasterDA:

```
    pea       0           ; return space
    pea       $FFFF       ; eventMask, ignored
    pushlong  #nDArecord  ; our NDA event record
    _TaskMasterDA
    pla                   ; event code returned
```

You can then process the event in a convenient way. Remember again that TaskMaster has already done the control tracking for the controls in your NDA window. You have the same multiclick information, control handles and IDs.

Conclusion

TaskMaster is a wonderful thing that makes any programmer's job easier. So let it work for you. Learn the capabilities of the new fields and new controls, and take advantage of them. Let TaskMaster take care of the system details, while you concentrate on the features that make your application special.

Further Reference

- Apple IIgs Toolbox Reference, Volumes 1 through 3
- Apple IIgs Technical Note #81, Extended Control Ecstasy
- Human Interface Note #8, Keyboard Equivalents

### END OF FILE TN.IIGS.084

###
### FILE: TN.IIGS.085
###

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#85: Moving the Mouse

Written by: Matt Deatherage

July 1990

This Technical Note discusses moving the cursor on the screen without touching the mouse.

It is sometimes desirable to programmatically move the QuickDraw II cursor on the screen without requiring the user to touch the mouse. This can be effective, for example, in tutorial software that actually shows mouse actions such as pulling down menus and dragging windows.

There is not an easy or obvious way to do this in the toolbox. This is not a bad thing; it prevents overzealous programmers from zapping the mouse all over the screen for suspicious reasons. You must remember that the mouse belongs to the user, not to the application. If the user has put the mouse somewhere, it should only be a user's action that causes the cursor to move elsewhere. Most of the time that action is touching the mouse and physically moving it. Do not move the mouse except in response to a user-initiated command.

The most obvious way to move the mouse position--calling PosMouse with the new mouse position--is not sufficient; PosMouse does not update the current mouse position. When the mouse is next moved, a mouse interrupt comes through and the new deltas are added to the old mouse position, resulting in correct ReadMouse results after the mouse has been physically moved. Also, PosMouse does not update the cursor on the screen.

Faking Out the System

When you wish to move the mouse yourself, you are in effect replacing (or adding to) the standard mouse with a small programmatic mouse substitute--your code. This qualifies as a "device" and can be considered an Event Manager "device driver." You can then make the appropriate Event Manager call, FakeMouse. When calling FakeMouse, you supply all the mouse information yourself, allowing you to move the mouse, simulate button presses, and in general replace the mouse.

Further Reference

o Apple IIgs Toolbox Reference, Volumes 1-3

### END OF FILE TN.IIGS.085

```
#####################################################################
### FILE: TN.IIGS.086
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#86: Risking Resourceful Code

Revised by: Matt Deatherage March 1991
Written by: C.K. Haun <TR> September 1990

This Technical Note covers considerations you need to keep in mind when using code resources.

Changes since September 1990: Now lists XCMD and XFCN resources as "Apple's code" and notes that other restrictions apply to them as well.

Code resources are wonderful things that can make your life better than it ever was before. Code resources are necessary when writing CDevs and can be very useful for control definition procedures, code modules for extensible programs like resource editors in fact, almost anywhere where you use regular compiled and linked code. But to do it right, you need to keep some rules in mind.

Apple's Code, Apple's Rules

The first code resources covered are the ones defined as fully supported by the System Software. These are rCtlDefProc ($800C), rCodeResource ($8017), rCDEVCode ($8018), rXCMD ($801E) and rXFCN ($801F). Before looking at the specifics, this Note describes in general terms what happens when the Resource Manager loads a code resource.

When you call the Resource Manager with a request for a code resource (or when the system does, as with rCtlDefProcs), it loads it like a normal resource. The Resource Manager finds the resource in a resource map in the current search path, allocates a handle for the resource using the attributes in the resource attribute bits, and loads the resource into memory.

Now the Resource Manager examines the resConverter bit in the resource header. If this bit is set, indicating that this resource needs to be converted (asit should be for an rCtlDefProc), the Resource Manager checks its tables to seeif a resource converter has been logged in (with the ResourceConverter call).For code resources, the correct converter has been logged in by the manager associated with that resource type. For example, the Control Manager logs in the code resource converter for rCtlDefProc resource type.

For code resources, InitialLoad2 is used to load the OMF from memory. Then the Resource Manager returns a handle containing a pointer to the start ofthe loaded, relocated code.

Rule 1: Code resources must be smaller than 64K

The code resource converter uses the InitialLoad2 function of the System Loader to load and convert code resources. That means that code resources are restricted in the same way that loading from memory is. One of these restrictions is that the code must be 64K or less.

Rule 2: Compiled and linked code only

Again, since InitialLoad2 is used to convert the code resource, the data must be in OMF format since InitialLoad2 expects to relocate standard load segments. When you prepare your code for inclusion in a code resource, compile and link the code as you normally would for a stand-alone program. Use the file produced by the linker for inclusion in your resource fork. You can use Rez to move the code from a data fork to your application's resource fork with a line in your resource description file similar to the following:

```
read rCodeResource (MyCodeIDNumber,locked,convert)"MyCompiledAndLinkedCode";
```

Rule 3: One segment please

Multiple segments are theoretically possible with code resources, but you have to manage memory IDs and the memory that the additional code segments use yourself. Since the code resource converter calls InitialLoad2, it uses the Memory Manager ID for the current resource application, and you cannot specify a different user ID directly. By changing your current resource application ID (by making an additional call to ResourceStartUp with a modified master ID,for example) you could manage multisegment code resources.

Rule 4: No dynamic segments

The dynamic segment mechanism does not work with code resources. Of course, your application can still use dynamic segments, but not code resource dynamic segments.

Rule 5: Set the right attributes

There are two sets of attributes you need to be concerned about for a code resource. The first set includes the standard resource attributes; the second set covers the attributes that the code itself has in the OMF image.

You need both sets to get the functionality you want. The resource attributes determine how the Resource Manager handles the resource. The OMF attributes control what InitialLoad2 does when it converts your code from OMF in a resource handle to relocated executable code.

Remember, you need to set both sets of attributes.

The resource attributes you need to set are locked and convert. The locked flag is necessary to prevent the resource from moving while InitialLoad2 processes it, and the convert flag is needed to signal the Resource Manager to call the code resource converter.

You must set the static OMF attribute, the others (like no special memory) you set as appropriate for your code in your application.

Rule 6: Know where to go

The handle you get back from the Resource Manager when you load and convert a code resource points to the beginning of the relocated and ready-to-execute code, not to the image of the code that is stored in the resource fork. So you can immediately jump to this code to execute it.

You can override this if you like clear the resConverter bit in the resource attributes. If this bit is zero, the Resource Manager does not call any resource converter (including the code resource converter).

Rule 7: Remember the Write

Keep in mind that any resource that uses a converter uses that converter both for reading and writing the resource. If you write out a code resource, the Resource Manager calls the Write routine for the code resource converter, which currently writes without doing any conversion it does not reconvert the codein memory back to OMF format. However, some converters (perhaps one you write) could reconvert the resource before writing it out.

Your Code, Your Rules

If you want to define your own code resource type (with a resource type of less than $8000 and greater than 0) you may want to follow the same rules as the system code resources use. In fact, you can even use the same code resource converter, by using the ResourceConverter call with your resource type, andlog the code resource converter as the converter to use with your resource type, like the following:

```
       pha
       pha                     ; return space
       _GetCodeResConverter    ; Misc Tools call to return the loader
*                              ; relocation code pointer
*                              ; (leave it on the stack for the next call)
       pea $0678               ; resource type you want to convert with this
*                              ; converter, any Application type you wish
       pea %01                 ; add this converter to the Application
*                              ; converter list, and log this routine in
       _ResourceConverter
```

or you can do whatever you like with the resource, including not having a converter and doing all the relocation and memory management of the code yourself. This can give you the ability to add more functionality than the standard code resources provide dynamic segmentation is one feature you could implement if you want to handle all the details yourself.

Or, you can manage the code any way you want, but keep the built-in system functions in mind, and use as many of them as you can. Make your life simpler.

One Final Note

If one of your resources is marked convert and preload the Resource Manager only preloads that resource if the converter for that resource is logged in as a converter for that type. If the Resource Manager cannot find the converter, it does not preload the resource.

Further Reference

- o Apple IIgs Toolbox Reference, Volume 3
- o GS/OS Reference
- o HyperCard IIgs Script Language Guide
- o HyperCard IIgs Technical Note #1, Corrections to the Script Language Guide
- o Apple IIgs Sample Code #9, Lister

### END OF FILE TN.IIGS.086

```
#####################################################################
### FILE: TN.IIGS.087
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs #87: Patching the Tool Dispatcher

Written by: Mike Lagae and Dave Lyons   September 1990

This Technical Note presents the Apple standard way to patch into the Apple IIgs Tool Dispatcher vectors.

This Note presents MPW IIgs assembly-language code which provides the Apple-standard way for utilities to patch and unpatch the Tool Dispatcher vectors. If all Tool Dispatcher patches follow this protocol, patches can be installed and removed in any order, without ever accidentally unpatching somebody who patched in after the one getting removed.

Using this protocol, each patch begins with a header in a standard form--a form recognizable by these routines (see PatchHeader). This way routines (like RemoveE10000) can scan through the list of patches and remove one from the middle.

If your patch is going to stay in the system until shutdown, use this standard patch protocol anyway. This way other utilities can still recognize your patch and scan past it to find the next one. This Note is not just to show you a way to patch the tool dispatcher--it's to show you the way. If you patch tool dispatcher vectors in any other way, you strip other utilities of their ability to remove their patches.

Of course, patching the Tool Dispatcher vectors slows down all toolbox calls, so you shouldn't patch the tool dispatcher without a pretty good reason. If you need to patch a toolbox function, it is usually better to do it by modifying a tool set's function pointer table instead of patching the dispatcher.

The code in this note is specific to the System tool dispatch vectors ($E10000 and $E10004), but the same technique is recommended for the User tool dispatch vectors--just change $E10000 to $E10008, $E10004 to $E1000C, and ToolPointerTable to UserToolPointerTable.

What Is This Stuff?

This Note presents the following four routines.

PatchHeader is the simplest patch function that obeys the protocol. This is where you put your own patch code.

InstallE10000 installs a patch into the patch chain. For example:

```
       pushlong #PatchHeader
       jsl InstallE10000
       ply
       ply                    ;remove the input parameter
       bcc noError            ;error in A
```

RemoveE10000 removes a patch from the patch chain. For example:

```
       pushlong #PatchHeader
       jsl RemoveE10000
       ply
       ply                    ;remove the input parameter
       bcc noError            ;error in A
```

CheckPatch determines whether the specified address is the starting address of a standard patch. For example:

```
       pushlong #PatchHeader
       jsl CheckPatch
       ply
       ply                    ;remove the input parameter
       bcc validPatch
```

First, here are some comments and global equates.

```
****************************************************************************
*
* Patch.e10000 - Routines to patch into the toolbox dispatch
*                vectors at $e10000 and $e10004.
*
* By Michael Lagae
* Software Quality Assurance
* GS Toolbox Test Team
*
* July 18, 1989
*
* Written for the MPW IIGS Assembler -- Version 1.1b1, 4/9/90
*
* Copyright 1989-1990 by Apple Computer, Inc.
*
****************************************************************************

       case yes
       machine M65816
       string asis
       msb on
       print on

       export CheckPatch              ; Check for a valid patch header.
       export InstallE10000           ; Adds a patch into the toolbox
;                                       vectors.
       export RemoveE10000            ; Removes a toolbox dispatch vector
;                                       patch.
       export PatchHeader             ; The simplest toolbox dispatch
;                                       vector patch.

****************************************************************************
* Equates - Various equates required by these routines.
```

```
*
versionNumber         equ $0100          ; The version number of this library.

dispatch1             equ $e10000        ; The first toolbox dispatch vector.
dispatch2             equ $e10004        ; The second toolbox dispatch vector.
ToolPointerTable      equ $e103c0        ; Pointer to the active System TPT.
UserToolPointerTable  equ $e103c8        ; Pointer to the active User TPT.

; Error return values from the routines InstallE10000 and RemoveE10000

noError               equ $0000          ; Value returned if no error occurs.
badHeaderError        equ $8001          ; Patch header wasn't valid.
headerNotFoundError   equ $8002          ; Header to remove wasn't in the
;                                          linked list.
```

PatchHeader is the standard shell for the actual patch code. Your code goes in here, at NewDispatch2. When you get control at NewDispatch2, the function number is in X and there are two RTL addresses on the stack (pushed after the function's parameters).

Your patch code does not care whether the tool call is being made through the $E10000 or $E10004 vector--in either case you get control with two RTL addresses on the stack.

```
****************************************************************************
* PatchHeader - Header required of all routines that will be patched
*               into the toolbox dispatch vectors.
*
* Note:  The code between next1Vector and NewDispatch2 must be included
*        for all calls.  The code below NewDispatch2 only needs to be
*        included for patches that want to post patch the calls.
*
PatchHeader proc
       entry next1Vector,next2Vector
       entry dispatch1Vector,dispatch2Vector
       entry NewDispatch1,NewDispatch2

next1Vector                              ; Where dispatch1 should go when
;                                          finished.
       jml next1Vector                   ; (Filled in by InstallE10000).
next2Vector                              ; Where dispatch2 should go when
;                                          finished.
       jml next2Vector                   ; (Filled in by InstallE10000).
dispatch1Vector                          ; Holds the JML instruction from
;                                          $e10000.
       jml dispatch1Vector               ; (Filled in by InstallE10000).
dispatch2Vector                          ; Holds the JML instruction from
;                                          $e10004.
       jml dispatch2Vector               ; (Filled in by InstallE10000).

anRtl  rtl                               ; An RTL instruction.  Its address
;                                          will be
                                         ; pushed on the stack for dispatch1
;                                          calls.

NewDispatch1                             ; Entry point for dispatch1 toolbox
;                                          vector.
```

```
       phk                             ; Push program bank.
       pea anRtl-1                     ; Push the address of a RTL.

NewDispatch2                           ; Entry point for dispatch2 toolbox
;                                        vector.

; The following code should be included in the PatchHeader if the patch wants
; to perform post patching.  This code will determine if the call that was
; made actually exists and if it does, post patching can occur.  If the call
; doesn't exist, any pre-call routines can be executed, but the post patching
; shouldn't be attempted because the dispatcher will remove the second return
; address from the stack, thus not returning to your post patching routines.
; Stack equates for this routine.

aLong  equ $0001                  ; A temporary long value.
oldDP  equ aLong+4                ; Where the direct page is saved to.
oldTM  equ oldDP+2                ; Where the tool call number is saved.

       phx                        ; Save the call that's being made.
       phd                        ; Save the current direct page.
       lda >ToolPointerTable+2    ; Get the TPT to determine the number
       pha                        ; of tool sets loaded.
       lda >ToolPointerTable
       pha
       tsc                        ; Set the direct page to the stack.
       tcd
       txa                        ; See if this tool set exits.
       and #$00ff
       cmp [aLong]                ; Is it larger than the number of tool
;                                   sets?
       bcs @noCall                ; JIF this tool set doesn't exist.
       asl a
       asl a
       tay                        ; Now get the pointer to the FPT.
       lda [aLong],y
       tax
       iny
       iny
       lda [aLong],y
       sta aLong+2
       stx aLong
       lda oldTM                  ; Get the function number.
       and #$ff00
       xba
       cmp [aLong]                ; Compare it to the number of entries in
;                                   table.
@noCall
       pla                        ; Remove aLong from the stack.
       pla
       pld                        ; Restore the original direct page.
       plx                        ; Recover the tool number.

; At this point the carry flag is set if the tool call doesn't exist and clear
; if the tool call exits.  No post patching must occur if the carry flag is
; set.

       jmp next2Vector            ; Go to the original $e10004 jump
;                                   instruction.
```

```
       endp

****************************************************************************
* CheckPatch - Checks the passed toolbox dispatch vector to see if it
*              points to a valid patch.
*
* Input: Passed via the stack following C conventions.
*     newPatchAddr (long) - Address of the patch routine.
*
* Output:
*     If newPatchAddr is a valid patch -
*          Carry clear
*     If newPatchAddr is not a valid patch -
*          Carry set
*
CheckPatch proc

zprtl         equ $01               ; The address for the rtl on our direct
;                                     page.
newPatchAddr  equ zprtl+3           ; Address of patch (parameter to this
;                                     routine).

       tsc                          ; Make the stack the direct page after
;                                     saving
       phd                          ; the current direct page.
       tcd


       lda newPatchAddr+2           ; Simple check to check for a valid
;                                     pointer.
       and #$ff00
       bne BadPatch                 ; Wasn't zero, can't be a valid pointer.

       lda [newPatchAddr]           ; Check for the first JML instruction.
       and #$00ff
       cmp #$005c
       bne BadPatch

       ldy #$04                     ; Check for the second JML instruction.
       lda [newPatchAddr],y
       and #$00ff
       cmp #$005c
       bne BadPatch

       ldy #$08                     ; Check for the third JML instruction.
       lda [newPatchAddr],y
       and #$00ff
       cmp #$005c
       bne BadPatch

       ldy #$0c                     ; Check for the fourth JML instruction.
       lda [newPatchAddr],y
       and #$00ff
       cmp #$005c
       bne BadPatch

       ldy #$10                     ; Check for the rtl and phk instructions.
       lda [newPatchAddr],y
       cmp #$4b6b
       bne BadPatch

       iny                         ; Check for the phk and pea instructions.
       lda [newPatchAddr],y
       cmp #$f44b
       bne BadPatch

       clc                         ; Calculate the address of the rtl
;                                    instruction.
       lda newPatchAddr
       adc #$000f
       ldy #$13                    ; Check for address of the rtl
;                                    instruction.
       cmp [newPatchAddr],y
       bne BadPatch

GoodPatch
       pld                         ; Restore the direct page and report
       clc                         ; that it was a good patch.
       rtl

BadPatch
       pld                         ; Restore the direct page and report
       sec                         ; that something was wrong.
       rtl

       endp

****************************************************************************
* InstallE10000   - Sets the jump vector at $e10000 and $e10004 to point to
*                   the passed new toolbox dispatch vector patch.  This routine
*                   also updates the linked lists so that more than one routine
*                   can be patched into the dispatch vectors.
*
* Input: Passed via the stack following C conventions.
*     newPatchAddr (long) - Address of the patch routine.
*

* Output:
*     If an error occurred -
*          Carry set, Accumulator contains one of the following error codes:
*               badHeaderError
*     If no error occurred and patch was installed successfully -
*          Carry clear, Accumulator contains zero.
*
InstallE10000 proc

oldPatchAddr    equ $01                   ; Address of existing patch.
zprtl           equ oldPatchAddr+4        ; The address for the rtl.
zpsize          equ zprtl-oldPatchAddr    ; Size of direct page we'll have on
;                                           the stack.
newPatchAddr    equ zprtl+3               ; Address of patch (parameter to
;                                           this routine).

       tsc                         ; Move the stack pointer to point beyond
       sec                         ; the direct page variables that we'll
```

```
        sbc #zpsize               ; place on the stack.
        tcs
        phd                       ; Save the direct page register.
        tcd                       ; Set the direct page.
        php                       ; Disable interrupts
        sei

        pei newPatchAddr+2        ; Check if patch header is valid.
        pei newPatchAddr
        jsl CheckPatch
        plx                       ; Remove the parameters from the stack.
        plx
        bcc @1                    ; Report the badHeaderError if detected.
        ldy #badHeaderError
        jmp Exit

@1      lda >dispatch1            ; Set up the next1Vector in the new patch.
        sta [newPatchAddr]        ; The JML instruction and low byte.
        lda >dispatch1+2
        ldy #$02
        sta [newPatchAddr],y      ; The middle and upper bytes.

        lda >dispatch2            ; Set up the next2Vector in the new patch.
        ldy #$04
        sta [newPatchAddr],y      ; The JML instruction and low byte.
        lda >dispatch2+2
        ldy #$06
        sta [newPatchAddr],y      ; The middle and upper bytes.

        lda >dispatch1+3          ; See if there's already a patch in
;                                   dispatch1.
        and #$00ff
        sta oldPatchAddr+2
        pha                       ; High byte of possible header address.
        lda >dispatch1+1
        sec
        sbc #$0011
        sta oldPatchAddr
        pha                       ; Low byte of possible header address.
        jsl CheckPatch
        plx
        plx
        bcs First                 ; JIF this will be the first patch
;                                   installed.

        ldy #$08                  ; Set up the dispatch1Vector in the new
;                                   patch.
        lda [oldPatchAddr],y
        sta [newPatchAddr],y      ; The JML instruction and low byte.
        ldy #$0a
        lda [oldPatchAddr],y
        sta [newPatchAddr],y      ; The middle and upper bytes.

        ldy #$0c                  ; Set up the dispatch2Vector in the new
;                                   patch.
        lda [oldPatchAddr],y
        sta [newPatchAddr],y      ; The JML instruction and low byte.
        ldy #$0e
```

```
       lda [oldPatchAddr],y
       sta [newPatchAddr],y         ; The middle and upper bytes.

       bra PatchIt                  ; Now patch dispatch1 and dispatch2.

First  ldy #$08                     ; Set up the dispatch1Vector in the new
;                                     patch.
       lda >dispatch1
       sta [newPatchAddr],y         ; The JML instruction and low byte.
       ldy #$0a
       lda >dispatch1+2
       sta [newPatchAddr],y         ; The middle and upper bytes.

       ldy #$0c                     ; Set up the dispatch2Vector in the new
;                                     patch.
       lda >dispatch2
       sta [newPatchAddr],y         ; The JML instruction and low byte.
       ldy #$0e
       lda >dispatch2+2
       sta [newPatchAddr],y         ; The middle and upper bytes.

PatchIt
       clc                          ; Calculate the address of the new
;                                     dispatch2.
       lda newPatchAddr
       adc #$0015
       sta newPatchAddr
       xba
       and #$ff00                   ; Mask in the JML instruction.
       ora #$005c
       sta >dispatch2               ; The JML instruction and low byte.
       lda newPatchAddr+1
       sta >dispatch2+2             ; The middle and upper bytes.

       sec                          ; Calculate the address of the new
;                                     dispatch1.
       lda newPatchAddr
       sbc #$0004
       sta newPatchAddr
       xba
       and #$ff00                   ; Mask in the JML instruction.
       ora #$005c
       sta >dispatch1               ; The JML instruction and low byte.
       lda newPatchAddr+1
       sta >dispatch1+2             ; The middle and upper bytes.

       ldy #noError                 ; Report that all went well.

Exit   plp                          ; Restore the interrupt state.
       pld                          ; Restore the previous direct page
;                                     register.
       tsc                          ; Restore the stack pointer.
       clc
       adc #zpsize
       tcs
       tya                          ; Value to return.
       beq @noerr
       sec                          ; Report that there was an error.
       rtl
@noerr clc                        ; Report that there was no error.
       rtl

       endp


****************************************************************************
* RemoveE10000 - Removes the specified patch from the dispatch1 and dispatch2
*                vectors and updates the linked lists for the remaining
*                toolbox patches.
*
* Input: Passed via the stack following C conventions.
*     patchToRemove (long) - Address of the patch to remove.
*
* Output:
*     If an error occurred -
*          Carry set, Accumulator contains one of the following error codes:
*               badHeaderError
*               headerNotFoundError
*     If no error occurred and patch was removed successfully -
*          Carry clear, Accumulator contains zero.
*
RemoveE10000 proc

patchDispAddr    equ $01                  ; Address of existing patch (and 1
;                                           extra byte).
prevHeader       equ patchDispAddr+5      ; Used to search through the
;                                           linked list.
zprtl            equ prevHeader+4         ; The address for the rtl.
zpsize           equ zprtl-patchDispAddr  ; Size of direct page we'll have
;                                           on the stack.
patchToRemove    equ zprtl+3              ; Address of patch (parameter to
;                                           this routine).

       tsc                        ; Move the stack pointer to point beyond
       sec                        ; the direct page variables that we'll
       sbc #zpsize                ; place on the stack.
       tcs
       phd                        ; Save the direct page register.
       tcd                        ; Set the direct page.
       php                        ; Disable interrupts
       sei

       pei patchToRemove+2        ; Check if patch header we were asked to
       pei patchToRemove          ; remove is a valid header.
       jsl CheckPatch
       plx                        ; Remove the parameters from the stack.
       plx
       bcc @1                     ; Report the badHeaderError if detected.
       ldy #badHeaderError
       jmp Exit

@1     clc                        ; Create the JML instruction that would
;                                   exist
       lda patchToRemove          ; if the patchToRemove was installed.
       adc #$0011
       sta patchDispAddr+1
       lda patchToRemove+2
       sta patchDispAddr+3
       lda patchDispAddr           ; Mask in the JML instruction.
       and #$ff00
       ora #$005c
       sta patchDispAddr

       cmp >dispatch1              ; Check if the patch to remove is the
;                                    first
       bne NotFirstOne             ; patch installed.
       lda >dispatch1+2
       cmp patchDispAddr+2
       bne NotFirstOne

       lda [patchToRemove]         ; Restore the Dispatch1 vector.
       sta >dispatch1
       ldy #$02
       lda [patchToRemove],y
       sta >dispatch1+2


       ldy #$04                    ; Restore the Dispatch2 vector.
       lda [patchToRemove],y
       sta >dispatch2
       ldy #$06
       lda [patchToRemove],y
       sta >dispatch2+2

       bra NoErr                   ; Everything went well.

NotFirstOne
       sec                         ; Assume that whatever is in dispatch1 is
       lda >dispatch1+1            ; patch and get the address of its header.
       sbc #$0011
       sta prevHeader              ; Low and middle bytes.
       lda >dispatch1+3
       and #$00ff
       sta prevHeader+2            ; Upper byte of header address.

@loop  pei prevHeader+2            ; Check if it really is a valid header.
       pei prevHeader
       jsl CheckPatch
       plx                         ; Remove the parameters from the stack.
       plx
       bcc @2                      ; Report that the patch that we asked to
       ldy #headerNotFoundError    ; remove wasn't found.
       bra Exit

@2     lda [prevHeader]            ; See if this patch points to patch we
       cmp patchDispAddr           ; want to remove.
       bne @nope
       ldy #$02
       lda [prevHeader],y
       cmp patchDispAddr+2
       bne @nope

       lda [patchToRemove]         ; Restore the next1Vector.
       sta [prevHeader]
       ldy #$02
       lda [patchToRemove],y
       sta [prevHeader],y

       ldy #$04                     ; Restore the next2Vector.
       lda [patchToRemove],y
       sta [prevHeader],y
       ldy #$06
       lda [patchToRemove],y
       sta [prevHeader],y

       bra NoErr                    ; Everything went well.

@nope  ldy #$02                     ; Get the address of the next patch
;                                     header.
       lda [prevHeader],y
       tax
       lda [prevHeader]
       sta prevHeader
       stx prevHeader+2

       sec
       lda prevHeader+1
       sbc #$11
       sta prevHeader
       lda prevHeader+3
       and #$00ff
       sta prevHeader+2


       bra @loop                    ; Now check this header.

NoErr  ldy #noError                 ; Report that all went well.

Exit   plp                          ; Restore the interrupt state.
       pld                          ; Restore the previous direct page
;                                     register.
       tsc                          ; Restore the stack pointer.
       clc
       adc #zpsize
       tcs
       tya                          ; Value to return.
       beq @noerr
       sec                          ; Report that there was an error.
       rtl
@noerr clc                          ; Report that there was no error.
       rtl

       endp

       end
```

Further Reference

---

  o Apple IIgs Toolbox Reference
  o Apple IIgs Technical Note #73, Using User Tool Sets

### END OF FILE TN.IIGS.087

```
#####################################################################
### FILE: TN.IIGS.088
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#88: The Page One Stack in a 16-Bit World

Written by: Dave Lyons

September 1990

This Technical Note clarifies the protocol for moving the stack pointer in and out of page one.

On page 13 of the Apple IIgs Firmware Reference, under "Save the value of the native-mode stack pointer," there is a code sample showing how to switch to the page-one stack by setting the stack pointer to $01xx, where xx is the contents of EMULSTACK at $01/0100.

However, the manual does not warn you about moving the stack pointer from page one to another area. When you do that, you must store the low byte of the stack pointer at EMULSTACK before moving the stack pointer out of page one. If you do not save the page-one stack properly, interrupt routines or some toolbox calls may destroy a part of the page one stack that you go back to later, expecting that return addresses are still there.

Note: If the auxiliary-memory stack and zero page are in use, you must use $01/0101 instead of $01/0100. See the Apple IIe Technical Reference Manual, pp. 153-154

Further Reference

- o Apple IIgs Firmware Reference
- o Apple IIe Technical Reference Manual

### END OF FILE TN.IIGS.088

```
#####################################################################
### FILE: TN.IIGS.089
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#89:   MessageByName--Catchy Messages

Written by: Dan Strnad & Dave Lyons

September 1990

This note clarifies MessageByName and provides examples of creating and retrieving a named message.

Did You Say You Want To Get A Message?

All you have to do is ask. Apple IIgs Toolbox Reference, Volume 3 already tells you how. Here's what the fine print says: with the createItFlag set to FALSE and the name of the message you are after in the nameString, you call MessageByName. What's unclear in the manual is that if the message was found, no error is returned, the createFlag is returned as FALSE, and messageID contains the ID you can pass to MessageCenter to retrieve the contents of the message. Here's an example of MessageByName in use.

The following code creates a named message.

```
CreateNamedMessage
            pha
            pha
            pea 1                                   ;create it
            pushlong #MsgBlock
            _MessageByName                          ;function $1701
            pla
            sta myMsgID                             ;keep the ID if you want
            pla                                     ;check the createFlag if
;                                                    you want
            ...

MsgBlock    dc.w MsgBlockEnd-MsgBlock
            dc.b 28,'XYZ Software:My Cool Product'    ;Pascal-style string
            ... more data goes here
MsgBlockEnd
```

The following code retrieves the message.

```
            pha
            pha
            pea 0                                   ;don't create message
            pushlong #MsgBlock
            _MessageByName                          ;function $1701
            ply                                     ;keep id of existing
;                                                    message
```

```
            pla                                             ;createFlag (ignore)
            bcs noMessage                                   ;carry set if an error
;                                                            occurred


            pea 2                                           ;MessageCenter action:GET
            phy                                             ;message ID for
;                                                            MessageCenter, below
            pha
            pha                                             ;space for NewHandle
;                                                            result
            lda #0                                          ;size of handle (0)
            pha
            pha
            ldx MyID                                        ;ID for empty
            phx
            pha                                             ;handle attributes (0)
            pha
            pha                                             ;no special location
            _NewHandle
            lda 3,s
            sta mcHandle+2
            lda 1,s
            sta mcHandle                                    ;keep a copy of the
;                                                            handle for later
            _MessageCenter                                  ;takes Action, Msg ID,
;                                                            and Handle

            lda mcHandle+2
            pha
            lda mcHandle
            pha
            phd
            tsc
            tcd
            ldy #2
            lda [3],y
            tax
            lda [3]
            sta 3
            stx 5

* now read data from the message at [3]
            ldy #$xxxx                                      ;index past the name
;                                                            string
            lda [3],y
            ...
            pld
            pla
            pla

            lda mcHandle+2
            pha
            lda mcHandle
            pha
            _DisposeHandle

noMessage   ...
```

```
mcHandle    dc.l 0
myMsgID     dc.w 0
```

MessageByName is available in Tool Locator versions 3.0 and later (System Software 5.0 and later).

Further Reference

- o Apple IIgs Toolbox Reference, Volumes 2-3

### END OF FILE TN.IIGS.089

```
#####################################################################
### FILE: TN.IIGS.090
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#90:   65816 Tips and Pitfalls

Revised by: Matt "Matt" Deatherage   March 1991
Written by: Dave "Dave" Lyons   September 1990

This Technical Note presents short 65816 assembly language examples
illustrating pitfalls and clever techniques.

Changes since November 1990: Added more explanations about the JSL table and
corrected a comment.

Dispatching Through an Address Table

The 65816 has a JSR ($aaaa,X) instruction for calling a selected subroutine
from a table of addresses, but it has no JSL ($aaaa,X) instruction. If you
need to dispatch to one of several routines that are not all in the same
bank,
you need an approach like the following. The idea is to perform a JSL to a
routine which does a long jump by pushing a three-byte "RTL address" on the
stack and then doing an RTL.

```
           jsl LngJmp           ;go jump to the routine
           ...

  LngJmp   asl a                ;take routine number in A and
           asl a                ; multiply it by 4
           tax                  ;put table index into X
           lda table+1,x        ;get "middle" word of address
           pha                  ; and push it
           lda table,x          ;get low word and
           dec a                ; decrement it by one
           phb                  ;push a single throw-away byte
           sta 1,s              ;store over low two of the 3 bytes
           rtl                  ;transfer control to the routine
table      dc.l routine1        ;table of 4-byte subroutine addresses
           dc.l routine2
           dc.l routine3
           ...
```

This code is correct because RTL pulls three bytes off the stack and
increments the two low bytes without incrementing the high byte.

Note: This approach to a table-based JSL is more flexible than JML ($XXXX) because it does not require any fixed-location storage or bank zero space, other than the stack.

On the other hand, the following code is not correct. The approach here is to make a table of addresses minus one.

```
           asl a             W
           asl a             R  ;multiply index by 4
           tax               O  ; and put it in X
           lda table+1,x     N  ;get the "middle" word
           pha               G  ; and push it
           lda table,x       !  ;get the low word
           phb               W  ;push a single throw-away byte
           sta 1,s           R  ;store over low two bytes
           rtl               O  ;transfer control to the routine
    table  dc.l routine1-1   N  ;table of 4-byte addresses minus one
           dc.l routine2-1   G
           dc.l routine3-1   !
           ...
```

This second sample code fragment fails if any of the routines in the table comes at the first byte of a bank. For example, if routine1 is at $060000, the address pushed is $05FFFF, and RTL transfers control to $050000, not $060000.

Dereferencing Handles Without Direct Page Space

When your code gets called with the D register undefined, you must not use direct page addressing without setting D to a known good value. Preserving and restoring locations on the caller's direct page is not reliable, because D could be pointing at bytes below the stack pointer (which can be destroyed by interrupts) or even at the $C0xx soft switches (that would make your direct page accesses accidentally fiddle with hardware).

A common way to get temporary direct page space is to point D at part of your stack. This following code dereferences a handle stored in the A and X registers (if the handle is $E01234 and refers to a block of memory at $056789, then on entry A=$00E0 and X=$1234, and on exit A=$0005 and X=$6789).

```
       phd                  ;save caller's direct-page register
       pha                  ;push high word of handle
       phx                  ;push low word of handle
       tsc                  ;get stack pointer in A
       tcd                  ;and put it in D
       lda [1]              ;get low word of master pointer (no ",Y"!)
       tax                  ; and put it in X
       ldy #$0002           ;offset to high word of master pointer
       lda [1],y            ;get high word
       ply                  ;remove low word of handle
       ply                  ; and high word
       pld                  ;restore the caller's direct-page register
```

Direct page addressing isn't the only way to address through pointers. Here's the same routine as before, but using the Data Bank register (B) instead of fiddling with D. (Note that handles do not have to be in bank $E0 or $E1, although they usually are.)

```
       phb                     ;save caller's data bank register
       pha                     ;push high word of handle on stack
       plb                     ;sets B to the bank byte of the pointer
       lda |$0002,x            ;load the high word of the master pointer
       pha                     ; and save it on the stack
       lda |$0000,x            ;load the low word of the master pointer
       tax                     ;and return it in X
       pla                     ;restore the high word in A
       plb                     ;pull the handle's high word high byte off the
                               ; stack
       plb                     ;restore the caller's data bank register
```

Emulation Mode Has 65816 Features

You don't have to switch into Native mode just to do an eight-bit operation with long addressing. Most 65816-specific instructions and addressing modes work in emulation mode in approximately the same way they work in eight-bit native mode. See the "Further Reference" for details.

Further Reference

- Apple IIgs Hardware Reference
- Programming the 65816, Including the 6502, 65C02 and 65802 (Eyes and Lichty, 1986, Brady)

### END OF FILE TN.IIGS.090

```
#####################################################################
### FILE: TN.IIGS.091
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#91: The Wonderful World of Universal Access

Revised by: Dave Lyons May 1992
Written by: Don J. Brady, Matt Deatherage, & Ron Lichty September 1990

This Technical Note discusses how your applications can be compatible with Universal Access software.

CHANGES SINCE JULY 1991: Added caution against reading the keyboard with interrupts disabled.

WHAT'S "UNIVERSAL ACCESS?"

Universal Access is the name given to software components designed to make Apple computers (in this case, the Apple IIgs) more accessible to people who might have difficulty using them. The Apple IIgs is very dependent on graphic objects, a keyboard and mouse; not all people can use these things very easily.

There are several components to Apple's Universal Access software:

- o CloseView. CloseView magnifies the Apple IIgs screen so that it's more easily seen by those with visual impairments. The hardware screen contains a magnification from two to twelve times as large as the "real" 32K Super Hi-Res graphics screen.

- o Video Keyboard. Video Keyboard is a New Desk Accessory that emulates a keyboard. A picture of a keyboard appears on the screen; a mouse-down event in any "key" makes Video Keyboard post a key-down event, so you can use a pointing device as a keyboard. ADB hardware is available to allow people to use head gear or other devices instead of mice; Video Keyboard lets these same devices replace the keyboard as well.

- o Easy Access. Easy Access comes in two parts: Sticky Keys and Mouse Keys. Sticky Keys makes the keyboard easier to use for those who have trouble pressing more than one key at a time; while Sticky Keys is activated, modifier keys may be released and still apply to the next keystroke. Mouse Keys allows the numeric keypad to be used as a mouse substitute. Sticky Keys and Mouse Keys are included in all ROM 03 Apple IIgs computers. The software versions allow all Apple IIGS computers to provide these functions, and provide additional icon feedback (in the upper right menu bar) for Sticky Keys.

HOW IT WORKS (ACCESS NOTHING AND CHECKS FOR FREE)

Universal Access generally works by replacing Apple IIgs toolbox functions. For example, CloseView patches QuickDraw so you do not draw to the actual screen, but to another buffer that CloseView can then magnify. Video Keyboard patches the Window Manager so that its keyboard window is always frontmost and fully visible (and accessible). Easy Access uses the ADB tools and the Event Manager to alter the way the hardware responds.

Since Universal Access changes the way the tools behave, your applications do not have to work very hard to be accessible to a broad range of physically challenged people. Just by following the rules, you have an accessible application. There are, however, a few guidelines you should keep in mind when designing your programs to make them as accessible as they can be.

UNIVERSAL ACCESS COMPATIBILITY GUIDELINES

- Don't disable interrupts and then try to read the keyboard. Easy Access on ROM 1 works at the Apple Desktop Bus level--if ADB interrupts are not being serviced, no keypresses will show up at $C000/$C025. Even Reset will not work, so the user may have to power down to regain control of the machine.

- Try to avoid using modal dialogs. Not only do lots of modal dialogs make for a cumbersome interface for everyone, they are especially annoying to those who have to move the mouse to a lot of OK buttons. More importantly, users cannot open NDAs like Video Keyboard while modal dialogs are frontmost.

  Video Keyboard can also be dragged in front of modal dialogs. If you are in the habit of using QuickDraw calls to draw items in Dialog Manager modal dialogs instead of creating custom dialog userItems, Video Keyboard users can drag the keyboard window in front of your dialog and erase the items (since the only items redrawn are those redrawn by the Dialog Manager's update routine). You can easily test this in all of your dialogs by obscuring each dialog with the Video Keyboard window a piece at a time, then moving Video Keyboard away, to be sure that all areas are completely redrawn.

  Let's say, for example, that you have a custom text item that changes between invocations of the same modal dialog. You might choose to draw the text yourself with LETextBox2 after creating the dialog with GetNewModalDialog but before letting the Dialog Manager handle events with ModalDialog:

```
     phx                      ; port: hi word from GetNewModalDialog
     pha                      ; port: lo word from GetNewModalDialog
     _SetPort

     lda OurText+2            ; pointer to text to draw in modal dialog
     pha
     lda OurText
     pha
     lda OurTextLength        ; Text length
     pha
     pea OurTextRect>>16      ; Text rectangle
     pea OurTextRect
     pea 0002                 ; Text justification (2 = fill)
```

```
_LETextBox2
```

To be Universal Access-friendly, you would, instead, implement a userItem routine like the following:

```
; . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
DrawDialogText
;
; DrawDialogText draws text pointed to by OurText into the Dialog.
; This userItem routine is called only by the Dialog Manager,
;   when it's implementing/updating the dialog.
; . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

     lda >OurText+2           ; pointer to text to draw in modal dialog
     pha
     lda >OurText             ; (long addressing: data bank unknown)
     pha
     lda >OurTextLength       ; Text length
     pha
     pea OurTextRect>>16      ; Text rectangle
     pea OurTextRect
     pea 0002                 ; Text justification (2 = fill)
     _LETextBox2

     lda 1,s                  ; get return address
     sta 7,s                  ; move to proper location
     lda 2,s                  ;  above input parameters
     sta 8,s

     pla                      ; move stack pointer up
     pla                      ;  to new return address location
     pla
     rtl
```

It will be called as a result of adding a template item like the following to the dialog template (note use of Item Value for the text length, since template Value fields are not used by userItems):

```
     TextTemplate  dc.w 3                         ; ID
     OurTextRect   dc.w TTop,TLeft,TBottom,TRight
                   dc.w UserItem+ItemDisable       ; Type
                   dc.l DrawDialogText             ; Pointer to our userItem
                                                   ; routine
     OurTextLength ds.w 1                          ; Text length (cheap place
;                                                  ; to put it)
                   dc.w 0000                       ; Item flag
                   dc.l 00000000                  ; Item color
```

Note that this is a simple example of a custom item routine; if you really had custom text that changed from invocation to invocation, you could use the existing Dialog Manager ParamText and longStatText2 item mechanisms.

o Use the Event Manager routines for event information. Do not access any hardware directly or use the lower-level Miscellaneous Tools routines for user event information--you steal that information from Universal Access. For example, use the Event Manager routine GetMouse to find the mouse location. Do not use ReadMouse or you steal mouse movement information from Universal Access.

o Call GetNextEvent or TaskMaster often. Long delays between calls do not let NDAs like Video Keyboard get events. If you cannot make these calls, at least call SystemTask.

o Do not assume that the hardware location of the screen is $E12000. Universal Access components that manipulate the entire screen (like CloseView) move the virtual screen so the hardware can be used for the magnified screen image.

  To find the screen location, look at the ptrToPixImage field in a grafPort after calling OpenPort (or in your window's window record after NewWindow). The image pointer gives the correct location of the screen.

  Assuming the current port is on screen, the following code finds the ptrToPixImage value:

```
   pha
   pha                     ;made space for port pointer
   _GetPort
   phd                     ;save direct page location
   tsc
   tcd                     ;port pointer is now at 3..6 on direct page
   ldy #4                  ;offset to high word of ptrToPixImage
   lda [3],y               ;got high word
   tax                     ;  in X
   ldy #2                  ;offset to low word of ptrToPixImage
   lda [3],y               ;got low word
   tay                     ;  in Y
   pld                     ;restored direct page location
   pla
   pla                     ;removed port pointer
```

The X and Y registers now contain the base address of the screen.

o Do not assume things about being the frontmost window. Even if FrontWindow says you have the frontmost window, your visRgn may have pieces missing. For example, the title bar of your window may be partially under the menu bar. Or there may be a floating "windoid" (like Video Keyboard's window) over part of your window.

  For these reasons you should not draw directly to the screen without first examining your window's visRgn. Do not just check for rectangularity--your visRgn could be rectangular and parts of your window still be obscured. If you use QuickDraw for all your drawing, QuickDraw automatically clips drawing activity to be entirely within the visRgn, so this is not a problem.

o Don't access QuickDraw data directly; use QuickDraw routines instead. For example, to access SCB data, use the QuickDraw routines GetSCB and SetSCB instead of reading the hardware at $E19D00. CloseView may have those SCBs changed to reflect a magnified portion of the screen. Also use GetColorEntry, SetColorEntry, GetColorTable, and SetColorTable. Don't access the hardware directly.

- Try to allocate memory after starting the tools. If you want to allocate memory before starting tools, do not use special memory. (Set the attrNoSpec bit in the attributes.)

Further Reference

- Apple IIgs Toolbox Reference
- Apple IIgs Firmware Reference
- Apple II Video Overlay Card Development Kit (APDA)

### END OF FILE TN.IIGS.091

##
### FILE: TN.IIGS.092
##

Apple II
Technical Notes

Developer Technical Support

Apple IIgs #92: Twisted Tales of TextEdit

Revised by: Dave Lyons — December 1991
Written by: C.K. Haun <TR> and Dave Lyons — September 1990

This Technical Note discusses some undocumented features and some bugs in the TextEdit tool set through System Software 5.0.4.
Changes since November 1990: Noted that a non-control TENew creates a Text Edit record for the current port.

TENew

TextEdit records you create with TENew are always tied to the current port at the time of the TENew call, whether or not the fNotControl bit is set. (For TextEdit controls, NewControl2 is the preferred call.)

TEInsert

Using the TEInsert call on an invisible TextEdit record causes the screen to scroll, exactly as if the TextEdit record were visible.

If you use LETextBox2 style text as input for a TEInsert call, any style change information contained at the end of the LETextBox2 text is ignored. To ensure that the style change is not ignored, append an additional character at the end of the block, then delete (with TESetSelect and TEDelete) the extra character after the TEInsert call.

TEGetText

The documentation for TEGetText says that a dataFormat value of $4 returns the text as "Formatted for input to LineEdit LETextBox2". This is not a reliable return method-this call may or may not succeed. Greater chance for success occurs with less than 4,000 characters in the TextEdit record.

TEGetText also supports getting just the text of the current selection range. Adding $0020 (onlyGetSelection) to the number passed in bufferDescriptor returns the text of the current selection. This technique does not work with data format LETextBox2, but does work with all other formats. Also, there is no corresponding bit for the associated style record, so you cannot get the style for just the current selection this way, if you request style information you get a styleRef for the entire TextEdit record.

TEClick

Using TEClick or TestControl on an inactive record currently causes that record to activate.

TERuler

Pixel tabbing values must all be greater than zero or TextEdit loops infinitely on a tab.

TEGetRuler & TESetRuler

TERuler, for the default ruler or any ruler that uses a tabType value of $1 returns a ruler four bytes longer than described in the documentation. The extra four bytes are all $FF, and they are the terminator characters for tabType $2 rulers. Expand your buffers by four bytes to prevent overwriting any data. TextEdit also expects the additional information on a TESetRuler call, so you should pad your ruler with four $FF bytes if you are using a type $1 ruler.

TESetText

Passing a zero-length class one input string (a word length string with the word set to zero) to TESetText causes TextEdit to crash.

TEPaintText

TEPaintText currently prints colored text in only four colors.

It's Not Dirty, It's Text

There has been some confusion about determining if a TextEdit record has been changed. The documentation has been a little vague, and the process itself has mislead some people. Here is The Truth: there is a TextEdit dirty flag, and you can use it and rely on it to tell you when a TextEdit record has changed.

The TextEdit dirty flag is bit 6 (fRecordDirty in the E16.TextEdit interface file) of the ctlFlag byte. This has caused some confusion because the ctlFlag byte is at offset $12 in the control definition template, and it is at offset $10 in the TextEdit or Control record. Just remember that it is not in the same place in the record as it is in the template.

If it is set, then the TextEdit or Control record has been changed since the last time the dirty bit was cleared. The dirty bit is clear initially when you create the TextEdit or Control record. Anytime after that, if the user enters text into the TextEdit record, TextEdit sets the dirty flag. It is up to your application to clear the dirty flag; TextEdit has no way of knowing when you've saved or cleared data.

Further Reference

---

o Apple IIgs Toolbox Reference, Volume 3

### END OF FILE TN.IIGS.092

```
#####################################################################
### FILE: TN.IIGS.093
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#93: Compatible Printing

Revised by: Matt Deatherage   May 1992
Written by: Matt Deatherage   September 1990

This Technical Note discusses printing on the Apple IIgs and how you can make your printing code more compatible.

CHANGES SINCE SEPTEMBER 1990: Added a note about expecting print records to keep the same attributes across Print Manager calls. Added the StyleWriter's iDev value.

HOW DOES PRINTING WORK ANYWAY?

There are, in general, two types of printing done on the Apple IIgs. The first kind is "desktop" printing, which uses the Apple IIgs Print Manager to render images created by QuickDraw II onto an output device. The other kind of printing is "text" printing, which is similar to the way classic Apple II applications print--you send ASCII text somewhere and a printer prints it as ASCII text. This printing normally involves no graphics and is very quick.

This Note covers both types of printing, and by understanding the internals and the methods used to print, you can avoid compatibility headaches in the future.

DESKTOP PRINTING

Desktop printing uses the Apple IIgs Print Manager. The process is described in detail in the Print Manager chapter of the Apple IIgs Toolbox Reference, and usually consists of a simple print loop:

```
     Open a document (PrOpenDoc)
          Open a page (PrOpenPage)
          Draw or Image the page in your favorite way
          Close the page (PrClosePage)
          Repeat for each page
     Close the document (PrCloseDoc)
     Print the document if it's spooled (PrPicFile)
```

Note that you should ALWAYS call PrPicFile at the end of your print loop. It completes the printing process, even for immediate or draft printing.

There's one real secret about the Print Manager that can cloud your understanding of printing--the Print Manager doesn't actually do anything. It loads, unloads, and keeps track of printer drivers and port drivers and performs some necessary housekeeping, but that's about it. Many people believe that the Print Manager is responsible for all imaging, managing documents, managing a printing grafPort and such, but it's not. (The myth is perpetuated by the Toolbox Reference which refers to these functions as handled by the Print Manager.) In fact, these functions are handled by printer drivers.

You actually call the printer driver for all of the routines in the print loop; all the Print Manager does is make sure the driver is loaded and dispatch to it. Therefore, most of the compatibility issues you have with printing are not with the Print Manager, but with printer drivers.

DEALING WITH THE PRINT RECORD

It's the printer driver's job to get information about a printing job from the user (it's the printer driver that handles the style and job dialog boxes, since the Print Manager cannot generically know what style and job options any printer can support), keep track of it, and print the document using those settings. Those settings are kept in a data structure associated with a document known as a print record.

Apple had only released two printer drivers at the time the first volume of the Toolbox Reference was published, and therefore the descriptions of the print record in that volume tend to be absolute. For example, the iDev field is documented as "one for an ImageWriter and three for a LaserWriter." In fact, the iDev field is the only method of print record interpretation available and there are several values for it:

```
     $0001 = ImageWriter
     $0002 = ImageWriter LQ
     $0003 = LaserWriter
     $0004 = Epson
     $0065 = StyleWriter
     $8001 = Generic dot-matrix (interprets the style subrecord like the
             ImageWriter driver)
     $8003 = Generic laser printer (interprets the style subrecord like the
             LaserWriter driver)
```

If you have checks in your code like "If it's not $0001, it must be a LaserWriter," you have problems with most of the other printer types.

The $8000 and greater iDev values are defined for third-party printer drivers. The printer driver has no way other than the print record to keep track of values for a given print job, so it has to store all such information in the print record. If all third-party drivers use proprietary style subrecord formats, no applications can read or set any of those values. Those drivers which can use the compatible $8000 and greater iDev values indicate to applications that the definitions in Toolbox Reference for the ImageWriter and LaserWriter drivers apply to these drivers as well. iDev values of $0002 or $0004 also interpret the style subrecord as the ImageWriter driver does.

PRINT RECORD RULES

Remember: the print record is the only way the printer driver has to maintain information about a particular job. The print record belongs to the user, the document, and the printer driver--NOT the application. Here are some rules for staying out of print record trouble.

- Always call PrValidate when changing fields in the print record. Even if a driver interprets the style subrecord like the ImageWriter driver, it may not support all the ImageWriter's style features (e.g., color printing). Calling PrValidate every time you change something in the print record gives the printer driver a chance to look at the havoc you've wreaked and correct it if necessary.

  You do not always get a feature you want. If a printer does not support color printing, you can set the "color" bit all day long and PrValidate clears it every time. You should be prepared for a new printer driver that does not support the features you want, and inform the user that the feature is not supported by this printer.

- Do not patch PrValidate to make it ignore bogus values in the print record unless instructed to do so by the printer driver author.

- Never, never tread on reserved fields in the print record. If you find a particular driver storing useful values some place, forget it. This is the only place a driver has to store information about a print job and some of it is not going to be supported.

  In particular, never try to interpret any values you may find in the printX subrecord of the print record. This subrecord is for the private use of printer drivers. Although printX is currently the worst compatibility risk, you must not tamper with other reserved fields.

- Don't assume that the print record will keep the same memory attributes across calls to the Print Manager (and therefore the printer driver). Specifically, don't assume that a print record will stay locked across calls to the Print Manager.

- If you want to learn more about printing, learn how printer drivers work. The specifications are in Apple IIgs Technical Note #35, appropriately entitled "Printer Driver Specifications." An understanding of how printer drivers do their work is an understanding of how printing works.

TEXT PRINTING

Text printing generally uses the built-in ASCII mode of most dot-matrix printers to print text quickly and efficiently.

Desktop printer drivers often have a "draft" mode, where they print text immediately instead of imaging it in the appropriate font and style. This is accomplished by intercepting low-level QuickDraw II routines called bottleneck procedures. When QuickDraw is called to draw text, the printer driver gets control instead and sends the text to the printer.

Although this is useful to users of desktop printer drivers, it is not a required feature of any printer driver, and those that do implement it each do so in their individual way. For example, the LaserWriter driver doesn't support this model of "draft" printing because the LaserWriter is normally a PostScript(R) device--sending straight ASCII to it doesn't necessarily work.

To imitate the way classic Apple II applications print, your application prompts the user for some device through which to print, and ASCII characters are sent through that device. There are a few ways to do this.

USING THE PRINT MANAGER

You can still use the Print Manager to print in ASCII mode by bypassing the printer driver. Simply use the Port Driver to send ASCII characters to the given target device with the PrDevWrite call. The specifications for Port Driver calls are in Apple IIgs Technical Note #36, also appropriately entitled "Port Driver Specifications." You make port driver calls as if they were Print Manager calls.

Although this method has been used, Apple does not recommend it. If the selected port driver is a network driver, this method is troublesome.

USING THE TEXT TOOLS

By using the Apple IIgs Text Tools, you can ask the user what slot to print through and send ASCII characters to that slot or port. Although this is better than using the Port Driver, it still has problems. The Text Tools cannot be fully GS/OS Slot Arbiter compatible; therefore, there might be GS/OS devices accessible to the user to which your application does not let him print. Also, it's difficult to detect which slots really have Text Tools' devices without knowing about Apple II firmware, and prompting the user for a slot number invites trying to print to the disk firwmare, which usually justs reboot the machine (unceremoniously).

USING GS/OS

GS/OS supports character drivers, such as printer interfaces, and using them is the best way to handle ASCII printing. GS/OS supports loaded drivers for character devices if you have them, and generates drivers for character devices it can recognize. In addition, GS/OS drivers have identification words so you can prompt with real messages instead of cryptic slot numbers.

You can use the GS/OS call DInfo to loop through all drivers and prepare a list of character drivers. You can then change their device IDs into text phrases, place them in a list, and prompt the user to select one. This call usually results in a list such as "Printer port, Modem port, Remote Print Manager, Printer interface, Text screen [the Console driver]." You may wish to change the names of the devices slightly to make the choice easier (e.g., "network printer" instead of "Remote Print Manager").

Apple strongly recommends using GS/OS for ASCII printing from 16-bit applications.

NOTE : The Remote Print Manager (.RPM) device driver in System Software 5.0 to 5.0.2 has a bug which causes character loss. System Software 5.0.3 fixes this bug.

Further Reference

- Apple IIgs Toolbox Reference
- GS/OS Reference
- Apple IIgs Technical Note #34, Low-level QuickDraw II Routines
- Apple IIgs Technical Note #35, Printer Driver Specifications
- Apple IIgs Technical Note #36, Port Driver Specifications

- Apple IIgs Technical Note #69, The Ins and Outs of Slot Arbitration
- Apple IIgs Technical Note #75, BeginUpdate Anomaly

### END OF FILE TN.IIGS.093

```
#####################################################################
### FILE: TN.IIGS.094
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#94: Packing It In (and Out)

Revised by: Dave Lyons May 1992
Written by: C.K. Haun <TR> September 1990

This Technical Note discusses a potential problem with the Miscellaneous Tools routine UnPackBytes.

CHANGES SINCE SEPTEMBER 1990: Noted that the problem detecting the end of the unpack-to buffer near the end of a bank is fixed in System 6.0.

PackBytes and UnPackBytes are handy data compression and expansion routines built into the Apple IIgs System Software. Using them can dramatically reduce the amount of space your application uses on disk or in memory, but you need to understand how these calls work to avoid problems in your applications.

BUFFER SIZE, BUFFER SIZE, BUFF, BUFF, BUFFER SIZE

There are some situations where the Miscellaneous Tools call UnPackBytes does not function as expected and can cause your application to loop infinitely while you're waiting for an unpacking process to finish.

The following packed data and code (in APW assembly) demonstrates the problem. It shows a small routine that unpacks data in two steps, simulating the situation in many applications where an arbitrary amount of data is unpacked in a variable amount of unpacking actions, depending on the results of the last unpack pass.

```
UnPackBuffer      ds    160                    ; area to unpack the data to
UnPackBufferPtr   dc    i4'UnPackBuffer'       ; pointer to unpacking buffer
UnPackBufferSize  ds    2
temp              ds    2

PackedData        dc h'FFFFFFFF'
EndPackData       anop
PackLength        dc i2'EndPackData-PackedData' ; how many bytes of packed
data

* In packbytes format $FFFF means '64 repeats of the next byte ($FF) taken as
* 4 bytes' as described on page 14-39 of Toolbox Reference, so
* this data should unpack into 512 $FF bytes

* The following code loops infinitely

                  lda   #160                   ; Unpack buffer size
                  sta   UnPackBufferSize
UnPackLoop        pea   0                     ; return space
                  pushlong #PackedData        ; pointer to packed data
                  pea   2                     ; size of the packed data,
                                              ; unpack two bytes
                                              ; at a time
                  pushlong #UnPackBufferPtr   ; pointer to pointer to
                                              ; unpacking buffer
                  pushlong #UnPackBufferSize  ; pointer to word with the
                                              ; size of the
                                              ;  unpacking buffer
                  _UnPackBytes
                  pla                         ; returns 0 bytes unpacked
                  sta   temp
                  lda   PackLength
                  sec
                  sbc   temp                  ; subtracting it from our
                                              ; known
                  sta   PackLength            ; length of packed data
                  bne   UnPackLoop            ; this is always be non-zero
```

The problem is in the data and the buffer size. UnPackBytes is being told to unpack two bytes ($FFFF), which generate 256 bytes of unpacked data, into a 160-byte buffer. Instead of reporting an error with this condition, UnPackBytes instead just does nothing and passes back zero as the returned number of bytes unpacked. If you are relying on the unpacked byte count returned to control your unpacking loop, then you may encounter this problem.

UnPackBytes can be used to unpack in multiple steps, of course, but it cannot unpack a partial record. It cannot unpack 160 bytes of the 256 bytes specified in this record because UnPackBytes does not maintain any state information, so it must unpack full records or do nothing. If the buffer had been 256 bytes, this call would have succeeded.

THE FIX

Fortunately, it's easy to avoid this situation if you know that it can exist. Simply, always supply UnPackBytes with a buffer that is big enough for it to unpack at least two bytes (a flag or count byte and a data byte). The largest value of a flag or count word possible is $FF, 64 repeats of the next byte taken as four bytes, which generates 256 unpacked bytes. So always give UnPackBytes a 256-byte long output buffer and you should never encounter this problem.

CHECK YOUR CURRENT APPLICATIONS

Please check your current applications to see if you could encounter this problem. One of the most likely places for this error to occur is in applications that process Apple Preferred (file type $C0, auxiliary type $0002) pictures. While most pictures currently available are screen-width or less (160 bytes or less per scan line), the Apple Preferred format and QuickDraw II both support pictures that are wider than the current Apple IIgs screen. If someone has created a picture with a PixelsPerScanLine value of 1,280 with a ModeWord of $0080, it would generate a scan line that was 320 bytes long. If a scan line in this hypothetical picture were all white, for example, the first two bytes of the packed scan line would be $FFFF, and applications that assume a standard maximum 160 bytes per scan line would not handle this correctly.

BUT THAT'S NOT ALL...

In System Software earlier than 6.0, UnPackBytes has some other buffering problems of which you need to be aware. The size and location of the input buffer (the buffer containing your packed data) can also cause problems.

You can ignore this section if your application requires System 6.0.

NOTE : These problems only occur if you are doing multipass unpacks. If you always unpack a packed data range in one pass (with one call to UnPackBytes for the whole data set) then you are not affected by these problems, and the restrictions described herein do not apply.

MULTIPASS RESTRICTIONS

When performing a multipass unpack (as described on pp. 14-43..44 of the Apple IIgs Toolbox Reference, Volume 1) the packed data needs to follow two rules.

Rule 1: Your packed data buffer cannot cross a bank boundary.
Rule 2: Your packed data buffer needs to be at least 65 bytes longer than the actual size of the data.

These rules are required by a bug in UnPackBytes. When UnPackBytes begins to unpack a record, it checks the record data to see if there are enough bytes in the current source buffer to unpack the number of bytes requested in the record header (described on pg. 14-39 of the Apple IIgs Toolbox Reference, Volume 1). If there are not enough bytes left for the current record (i.e., the header says to process 63 bytes, and there are only 30 left in the buffer), UnPackBytes returns to the caller. The caller then adjusts the source buffer for the next pass based on the amount of actual bytes unpacked, so the bytes left over from the last pass get processed the next time.

The problem occurs when the partial record is close to the end of a bank. When UnPackBytes checks to see if there is enough data left in the buffer, the check is flawed when the real end of the buffer is near the end of a bank, and a complete copy of the partial record would extend into the next bank. UnPackBytes erroneously thinks that the record is complete, and happily unpacks the remaining actual packed data, plus random information from the next bank. It continues to unpack nonsense data until it fills the unpacking buffer and the number of bytes unpacked returned by the UnPackBytes call is greater than the bufferSize parameter passed as input.

To prevent this bug from occurring, you need to make sure that the buffer for the packed data is at least one record length away from the end of a memory bank. Since the largest packed data record is one flag byte and 64 data bytes, adding 65 bytes to the end of your buffer does the trick. This ensures that your packed data is 65 bytes away from the end.

Following is an example of a safe way to prepare your packed data buffer for multipass unpacking, in APW assembly:

```
* Some data space
```

```
myCallBlock  dc     i2'2'                ; two parameters
fileRefNum   ds     2                    ; file reference number
EOFreturned  ds     4                    ; file length returned by this call
myIDNumber   ds     2                    ; your application memory manager ID
number
* assume that a packed data file is open, and it's a plain packed screen
image, not over 32K
             jsl    $E100A8              ; ask GS/OS for the length of the
data
             dc     i2'$2019'            ; Get_EOF call
             dc     i4'myCallBlock'

* Now we need a handle to read it into
             pha
             pha                         ; return space
             pea    0                    ; size, high word
             lda    EOFreturned          ; the actual size of the packed data
             sta    actualPackDataSize
             clc
             adc    #65                  ; ask for a handle 65 bytes longer
                                         ; than the data
             pha
             lda    myIDnumber           ; Memory Manager ID for your
                                         ; application
             pha
             pea    $8010                ; attrLocked and attrNoCross
             pea    0
             pea    0                    ; anywhere
             _NewHandle                  ; get the handle
```

Now you have a handle 65 bytes longer than your data that does not cross a bank boundary. You are ready to read in the data and perform a multipass unpack.

PACKBYTES BUFFERS COUNT TOO

PackBytes can also cause you problems if you do not plan for the worst-case situation. Unlike the other toolbox compression routine ACECompress, PackBytes is not guaranteed to shrink the source data. In fact, your data size may actually grow after a PackBytes call.

If you pass a data stream of 64 bytes, all with different values, to PackBytes, PackBytes puts 65 bytes in your output buffer--the 64 original data bytes and the flag byte of $3F, indicating "64 bytes follow, all different." Unless you preprocess or analyze your data before packing to avoid this situation, make sure your output buffer is large enough to hold the worst case situation, one additional byte generated for every 64 bytes passed to PackBytes for compression.

Further Reference

- Apple IIgs Toolbox Reference, Volumes 1-3
- File Type Note for File Type $C0, Auxiliary Type $0002, Apple Preferred Format

### END OF FILE TN.IIGS.094

###
### FILE: TN.IIGS.095
###

Apple II
Technical Notes

Developer Technical Support

Apple IIgs #95: ROM Diagnostic Errors

Written by: Dan Strnad September 1990

This Technical Note describes errors returned by the ROM Diagnostics on Apple IIgs systems.

The Built-In Diagnostics Revealed

The IIgs has a self-test capability in ROM. The self-test is activated by pressing Open-Apple and Option on power up, or Open-Apple, Option, and Reset. During the test, the test number is visible on the bottom of the screen followed by six zeros. After all tests are complete, a continuous 6 KHz one-second beep sounds and the screen displays a System Good message. If any test fails, the screen displays a message of the form System Bad: AABBCCDD on the lower left hand side and a staggered AABBCCDD on the upper left hand side to help read the error code in the event of a RAM failure. In the event of video failure, the failure code is also sent to the printer port. In the number contained in the error message, AA is the test number that failed and the failure code is embedded in the BB, CC, and DD fields. The complete failure codes for each of the 12 tests are as follows:

Self Test 1: ROM Test

AA =	01
BB =	Failed checksum
DD =	01 if the test encountered bad RAM and the error code is a RAM error code similar to the RAM Test error codes

For a failure in ROM, the ROM diagnostics also display RM on the top left hand corner of the screen.

Self Test 2: RAM Test

AA =	02
BB =	Bank Number (or $FF for ADB Tool call error)
CC =	Bit(s) failed

Self Test 3: Soft Switches and State Register Test

AA =	03
BB =	State Register bit (if any)
CC =	Low byte of soft switch address

Self Test 4: RAM Address Test

```
AA =    04
BB =    Failed bank number (or $FF for ADB Tool call error)
CCDD =  Failed address
```

Self Test 5: Speed Test

```
AA =    05
BB =    01:  Speed stuck slow
        02:  Speed stuck fast
```

Self Test 6: Serial Test

```
AA =    06:
BB =    01:  Register R/W
        04:  Tx Buffer empty status
        05:  Tx Buffer empty failure
        06:  All Sent Status fail
        07:  Rx Char available
        08:  Bad data
```

Self Test 7: Clock Test

```
AA =    07
DD =    01:  Fatal error occurred and the test is aborted
```

Self Test 8: Battery RAM Test

```
AA =    08
BB =    01:  Address test and CC = bad address
        02:  Non-volatile RAM failed and CC = pattern, DD = address
```

Self Test 9: Apple Desktop Bus Test

```
AA =    09
BBCC =  Bad checksum
DD =    01:  Apple Desktop Bus tools call encountered a fatal error, no
             checksum  computed.
```

Self Test 10: Shadow Register Test

```
AA =    0A
BB =    01:  Text page 1 fail
        02:  Text page 2 fail
        03:  Apple Desktop Bus Tool call error
        04:  Power On Clear bit error
```

Self Test 11: Interrupts Test

```
AA =    0B
BB =    01:  VBL interrupt time-out
        02:  VBL IRQ status fail
        03:  1/4 sec interrupt
        04:  1/4 sec interrupt
        05:
        06:  VGC IRQ
        07:  Scan line
```

```
Self Test 12:  Sound Test

AA =    0C
DD =    01:  RAM data error
        02:  RAM address error
        03:  Data register failed
        04:  Control register failed
        05:  Oscillator interrupt timeout
```

Further Reference

---

  o Apple IIgs Hardware Reference, Second Edition

### END OF FILE TN.IIGS.095

```
#####################################################################
### FILE: TN.IIGS.096
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs #96: Standard File Customization

Written by: Dan Strnad

November 1990

This Technical Note discusses particulars of using custom dialog boxes for the Open and Save File dialog boxes and custom drawing routines to display the files and folders listed.

About the Templates

Volume 3 of the Apple IIgs Toolbox Reference states the following about the Open File dialog box template for Standard File:

"The scroll bar item (item5) is not used for single-file calls. For multifile calls, this item contains the Accept Button Definition."

What is not stated explicitly is that, although the scroll bar item is not used for single-file calls, a place holder for it must be included in the dialog box template. Another particular not explicitly stated is that the strings used by the item templates must be Pascal strings; no listType field is provided in the extended list control record as was present in the List Manager's original list record structure.

Custom Item Draw Procedures

Custom item draw procedures have the rectangle in which the item is to be drawn, the List Manager's memrec structure corresponding to that item, and a handle to the extended list control record available on the stack. By including a custom item draw procedure, programs are able to get a handle to the extended list control record. The custom item draw procedure could also make the handle available to other routines, such as the dialogHook routine. With the handle, programs can now perform specialized operations during a standard file call, such as checking which item is selected before allowing the user to cancel. The code fragment below (from DTS Apple II Sample Code #18, AccessPriv) illustrates the use of SFPGetFile2 with a custom item draw routine.

```
static char SaveStr[] = "\pSave";
static char OpenStr[] = "\pOpen";
static char CloseStr[] = "\pClose";
static char DriveStr[] = "\pDrive";
static char CancelStr[] = "\pCancel";
static char FolderStr[] = "\pNew Folder";
static char AcceptStr[] = "\pAccept";
```

```
ItemTemplate OpenBut640 =   {1,
                            61,265,73,375,
                            buttonItem,
                            OpenStr,
                            0,
                            0,
                            0L};

ItemTemplate CloseBut640 =  {2,
                            79,265,91,375,
                            buttonItem,
                            CloseStr,
                            0,
                            0,
                            0L};

ItemTemplate NextBut640 =   {3,
                            25,265,37,375,
                            buttonItem,
                            DriveStr,
                            0,
                            0,
                            0L};

ItemTemplate CancelBut640 = {4,
                            97,265,109,375,
                            buttonItem,
                            CancelStr,
                            0,
                            0,
                            0L};

ItemTemplate Scroll640 =    {5,
                            43,265,55,375,
                            buttonItem,
                            AcceptStr,
                            0,
                            0,
                            0L};

ItemTemplate Path640 =      {6,
                            12,15,24,395,
                            userItem,
                            0L,
                            0,
                            0,
                            0L};

ItemTemplate Files640 =     {7,
                            25,18,107,215,
                            userItem + itemDisable,
                            0L,
                            0,
                            0,
                            0L};

ItemTemplate Prompt640 =    {8,
                             3,15,12,395,
                             statText + itemDisable,
                             0L,
                             0,
                             0,
                             0L};


/**************************************************************************
*
* myDialogHook
*
***************************************************************************/

pascal void myDialogHook(strip1,strip2)
long strip1;
long strip2;
{
}

/**************************************************************************
*
* CustomItemDraw
*
**************************************************************************/

pascal void CustomItemDraw(itemDrawPtr)
Pointer itemDrawPtr;
{
static unsigned int flag, dbr;        /* result, data bank register value */
byte          StringCount;
char          *ItemPascalString;
Word          ItemFileType;
Long          ItemAuxType;
Rect          *TheItemRectPtr;
MemRec        *TheMemRecPtr;
CtlRecHndl    TheSFListControlHndl;
Point         MyOldPenPos,
              MyNewPenPos;

static char FileString[] = "xxxx yyyyyyyy ";

/* save our data bank and set current to global page */
dbr = SaveDB();
/* Get the Rect from High on the Stack */
TheItemRectPtr = (Rect *)(*((long *)(((long)&itemDrawPtr)+ 36L)));
                                                 /* save old pen position */
GetPen(&MyOldPenPos);                            /* Set our pen position */
MyNewPenPos.h = TheItemRectPtr->h1 + 5;
MyNewPenPos.v = TheItemRectPtr->v2 -2;
MoveTo(MyNewPenPos);                             /* relocate the pen */

/* get our member record; this is just to reveal where it is on the stack */
TheMemRecPtr = (MemRec *)(*((long *)(((long)&itemDrawPtr)+ 32L)));

/* get the list cntrol handle; ditto */
TheSFListControlHndl = (CtlRecHndl)(*((long *)(((long)&itemDrawPtr)+ 28L)));
```

```
StringCount = (byte) *itemDrawPtr;              /* get the string length */
ItemPascalString = itemDrawPtr;                 /* set our user string */
ItemFileType =  *(Word *)(itemDrawPtr+StringCount+1L); /* get our FileType */
ItemAuxType =  *(Long *)(itemDrawPtr+StringCount+3L);  /* get our AuxType */

/* format for display */
sprintf(FileString, "%.4x-%.8lx ",ItemFileType,ItemAuxType);
c2pstr(FileString);                 /* turn it into a P string */
DrawString(FileString);             /* Draw it */
DrawString(ItemPascalString);       /* catenate File name to the other info */
FrameRect(TheItemRectPtr);
MoveTo(MyOldPenPos);                /* return the pen to starting position */
RestoreDB(dbr);                     /* restore our data bank */

}

/**************************************************************************
*
* ChooseFolder
*
* presents user with dialog to select folder to show/set privileges of
*
**************************************************************************/

void    SomeProc()
{
DialogTemplate GetDialog640;

GetDialog640.dtBoundsRect.v1 = 0;
GetDialog640.dtBoundsRect.h1 = 0;
GetDialog640.dtBoundsRect.v2 = 114;
GetDialog640.dtBoundsRect.h2 = 400;
GetDialog640.dtVisible = -1;
GetDialog640.dtRefCon = 0L;
GetDialog640.dtItemList[0] = &OpenBut640;
GetDialog640.dtItemList[1] = &CloseBut640;
GetDialog640.dtItemList[2] = &NextBut640;
GetDialog640.dtItemList[3] = &CancelBut640;
GetDialog640.dtItemList[4] = &Scroll640;
GetDialog640.dtItemList[5] = &Path640;
GetDialog640.dtItemList[6] = &Files640;
GetDialog640.dtItemList[7] = &Prompt640;
GetDialog640.dtItemList[8] = 0L;

SFPGetFile2(    /* user selection of folder to get/set privs of */
            120, 53,
            CustomItemDraw,
            refIsPointer,
            prompt,
            0L,
            0L,
            &GetDialog640,
            myDialogHook,
            &myReply
);
```

Further Reference

- Apple IIgs Toolbox Reference, Volumes 1 & 3
- DTS Apple II Sample Code #18, AccessPriv

### END OF FILE TN.IIGS.096

```
#####################################################################
### FILE: TN.IIGS.097
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#97: Picture Comments and Printing

Written by: Matt Deatherage, Suki Lee & Ben Koning November 1990

This Technical Note discusses QuickDraw Auxiliary picture comments and how they can be used to help control the printing process.

What's a Picture Comment?

Picture comments are a way in which extra information beyond normal QuickDraw II calls can be embedded in a QuickDraw II picture. Comments can contain virtually anything; they consist of a length, a handle containing the comment and a "kind" that identifies the general type of information in the comment. Picture comment kinds less than or equal to 256 ($100) are reserved for Apple Computer, Inc.

For comments to have any significance, there must be a way that a routine can take special action on them. One of the standard bottleneck procedures is called every time a picture comment is encountered, and it is passed the picture comment's kind, size, and handle on QuickDraw II's direct page. You can insert the address of a custom picture comment handler into the grafProcs field of a grafPort as described in Apple IIgs Technical Note #34, Low-Level QuickDraw II Routines. If no custom comment handler is present in the grafPort, the system calls its own StdComment routine, which ignores all comments.

The current picture comment handling routine (either a custom one or the system's default one) is called whenever a picture comment is generated (with the QuickDraw Auxiliary call PicComment) or played back from a picture (from within DrawPicture). Since the picture comment handling procedure is called when the comment is created, a picture does not have to be open for this facility to work.

Picture comments are ideal ways for applications to pass information to printer drivers as they are generated through toolbox calls and are easily accessible to any desktop program. If the printer driver is printing in immediate mode, it can intercept and act on the picture comment when it is generated. If the printer driver is printing in deferred mode and recording page images with QuickDraw II pictures, it can intercept and act on the picture comment when the picture is played back.

Apple's ImageWriter, ImageWriter LQ and LaserWriter drivers (from System Software 5.0.3) all support various kinds of picture comments for controlling printed output. Applications are encouraged to use these picture comments for finer control over printing. Authors of printer drivers are encouraged to act on these picture comments where appropriate, so applications which use them achieve similar results across printing platforms.

The LaserWriter Driver's Picture Comments

Version 2.2 and later of the LaserWriter driver support the following five PostScriptr picture comments:

Name	Kind	Size	Handle
PostScriptBegin	190	0	NIL
PostScriptEnd	191	0	NIL
PostScriptHandle	192	-	PostScript data
PostScriptFile	193	-	PostScript path name
TextIsPostScript	194	0	NIL

Table 1-PostScript Picture Comments

The print loop must be completed normally with or without any PostScript picture comments that are included. PostScript transmission must begin with the PostScriptBegin picture comment and end with the PostScriptEnd picture comment. Never nest PostScriptBegin and PostScriptEnd picture comments.

The PostScriptHandle picture comment takes a handle containing PostScript commands (in the form of ASCII data) and sends it to the LaserWriter. The size field must contain the size of the handle.

The PostScriptFile picture comment takes a handle containing the pathname of a disk file containing PostScript commands. The size field must contain the size of the pathname.

The TextIsPostScript picture comment takes text drawn through the QuickDraw II StdText bottleneck and sends it to the LaserWriter as PostScript. This picture comment has the effect, from the application's point of view, of interpreting all strings passed to DrawString and similar calls as PostScript. This picture comment is specific to LaserWriters (idev = $0003). Other drivers do not implement this picture comment; therefore, text drawn through QuickDraw II is simply printed-it is neither interpreted as PostScript nor ignored.

The driver does not check for PostScript errors, so the data sent to the LaserWriter must be correct. Always terminate PostScript text with a carriage return character. The transformation the driver uses flips text and prints it upside down on the page. Applications should set their own transformation matrices to serve their needs. Never use the LaserWriter's userdict-define a local dictionary for your application's use. Never use exitserver, initgraphics, grestoreall, erasepage, or showpage PostScript commands, as these commands can alter the driver's environment.

See Chapter 3 of the LaserWriter Reference Manual for some examples of how to use picture comments.

The ImageWriter Driver's Picture Comments

ImageWriter driver version 4.0 and later uses three picture comments to control alternate color selection:

Name	Kind	Size	Handle

Reserved	250	-	Reserved
FillColorTable	251	42	See below
ChangeSCBs	252	14	See below

The structure passed in the handle to FillColorTable looks like the following:

version	word	must be zero
signature	word	must be $A55A
tableno	word	the color table to be modified (0-15)
table	32 bytes	The new color values for the color table
reserved	long	must be zero

The structure passed in the handle to ChangeSCBs looks like the following:

version	word	must be zero
signature	word	must be $A55A
Y1	word	line number of first SCB to change (from zero to rPage.Y2)
Y2	word	line number of last SCB to change (from zero to rPage.Y2)
SCBvalue	word	the new scan line control byte
reserved	long	must be zero

PrOpenPage reinitializes the printing grafPort, so these picture comments should be used immediately after PrOpenPage to set custom colors. Having lines with both 320- and 640-mode is not recommended and will probably not be supported in the future.

The ImageWriter driver uses picture comment 250 to internally mark the end of a page. Applications must not use this picture comment; use PrClosePage to end a page's definition.

Further Reference

- o Apple IIgs Toolbox Reference, Volumes 1-3
- o Apple IIgs Technical Note #34, Low-Level QuickDraw II Routines
- o Apple IIgs Technical Note #35, Printer Driver Specifications
- o Apple IIgs Technical Note #93, Compatible Printing
- o d e v e l o p, October 1990, Issue 4, "Driving to Print"

PostScript is a registered trademark of Adobe Systems, Incorporated.

### END OF FILE TN.IIGS.097

```
#####################################################################
### FILE: TN.IIGS.098
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#98: Aren't Windows A Pane?

Revised by: Dave Lyons May 1992
Written by: Dave Lyons January 1991

This Technical Note describes interesting Window Manager things.

CHANGES SINCE JANUARY 1991: Noted that in System 6.0 it's safe to use Window color table resources. Added a section on changing the desktop pattern or picture.

CHANGING THE DESKTOP PATTERN OR PICTURE

The best way to set a new desktop pattern or picture is as follows. This works with the Finder and other desktop applications.

1. Use MessageCenter to delete message 2, the desktop message. (If there wasn't one, that's fine--there still isn't.)
2. Use MessageCenter to create a new message 2, containing the pattern or picture you want (see the Window Manager chapter of Apple IIgs Toolbox Reference, Volume 3).
3. Call Desktop (in the Window Manager) with a deskTopOp of 8 and a dtParam of $00000000. This notifies any part of the system that cares (such as the Finder) that there is a new desktop pattern.
4. Call Desktop with a deskTopOp of 4 and a dtParam of $00000000 and keep the result.
5. Call Desktop with a deskTopOp of 5 and use the result from step 4 as dtParam. This sets the desktop pattern to what it already is, forcing the desktop to redraw (this works whether a pattern, picture, or pointer to desktop-drawing routine is involved).

A WARNING ABOUT WINDOW COLOR TABLE HANDLES AND RESOURCES

The System 6.0 Window Manager fixes the problem described below. If your application requires System 6, you can safely ignore this section.

All versions of the Window Manager that support window color tables specified as handles or resources, up to and including System Software 5.0.4, work unreliably when a standard window's color table is supplied by handle or resource ID.

The problem is not immediately obvious; only one bit of memory is accidentally cleared, but the address is unpredictable in advance. (When unlocking the color table handle, the standard window definition procedure attempts to unlock the handle manually by turning off bit 15 of word offset +4 in the master pointer record. But it gets the high and low words of the handle reversed and usually turns off bit 15 of the word at offset $80E4 or $80E5 in some bank of RAM determined by the low byte of the handle.)

The solution is to avoid supplying color table handles or resource IDs to the Window Manager. Supply color table pointers instead. You can get a color table pointer from a color table resource ID by calling LoadResource on the color table resource, locking the handle and dereferencing it. Memory is less fragmented if color table resources used in this way are marked as attrFixed.

One method is to put the window color table pointer into the window template before calling NewWindow2. If you are creating the window from an rWindParam1 resource, you need to use LoadResource to get the template into RAM so that you can stuff the color table pointer into the template. (Be sure to change the moreFlags field to indicate that the color table is a pointer, if the template indicates it's a resource.) After you create the window with NewWindow2 (by handle), use ReleaseResource to release the rWindParam1 resource.

Another method is to create the window as invisible and pass the window color table pointer to SetFrameColor before calling ShowWindow.

Further Reference

- o   Apple IIgs Toolbox Reference, Volumes 2-3

### END OF FILE TN.IIGS.098

```
#####################################################################
### FILE: TN.IIGS.099
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#99: Supplemental Scrap Types

Revised by: Dave Lyons May 1992
Written by: Matt Deatherage & Matthew Reimer January 1991

This Technical Note describes public scrap types.

CHANGES SINCE MARCH 1991: Added information on Scrap Type $8003 (Resource Reference Scrap); added a cross-reference to HyperCard IIgs Technical Note #3.

The Apple IIgs Toolbox Reference lists only two known scrap types--text ($0000) and pictures ($0001). Other assigned scrap types are documented in this Note. The format used to describe the scraps is similar to that used in File Type Notes, where the offsets, given in the form (+xxx), determine the offset from the beginning of the scrap handle.

SAMPLED SOUND SCRAP (TYPE: $0002)

The following describes the Sampled Sound scrap format. It consists of a ten-byte header followed by the sample data bytes. This format is identical to the sampled sound resource format.

Format	(+000)	Word	This must always be zero.
Wave Size	(+002)	Word	Sample size in pages (256 bytes per page). For example, an 8K sample takes 32 pages; a 128K sample requires $200 pages.
Rel Pitch	(+004)	Word	The high byte of this word is a semitone value; the low byte is a fractional semitone. These values are used to tune the sample to correct pitch. (See HyperCard IIgs Technical Note #3, Tuning Sampled Sounds.)
Stereo	(+006)	Word	The output channel for this sound is in the low nibble of this word.
Sample rate	(+008)	Word	The sampling rate of the sound, in Hertz (Hz).
Sound	(+010)	Bytes	The sampled sound data. The bytes are all 8-bit samples. The sample starts here and continues until the end of the scrap.

TEXTEDIT STYLE SCRAP (TYPE: $0064)

The TextEdit Style Scrap format is the same as the TEFormat structure defined in Volume 3 of the Apple IIgs Toolbox Reference, which is also the same as the rStyleBlock resource format defined in the same volume.

ICON SCRAP (TYPE: $4945)

The Icon scrap format is the same as the format for Finder Icon Data records, documented in detail in the File Type Note for File Type $CA, Finder Icon Files. If there is more than one Icon Data record in a scrap, they are concatenated together with no intervening space.

MASK SCRAP (TYPE: $8001)

The Mask scrap format is exactly the same as the PICT scrap ($0001) format, except that the pixel image the picture draws contains only zeroes and ones. When drawn, this picture creates a mask. The mask has zeroes where the image can be seen through the mask, and ones where the mask does not allow the picture through. When pasting a Mask scrap, initialize the destination bitmap to zero and draw the picture.

You can create the mask image by using regular QuickDraw II calls (using ovals, rectangles, etc.) or you can create it independently and include it with PaintPixels or other pixel map manipulation routines.

COLOR TABLE SCRAP (TYPE: $8002)

The following describes the Color Table scrap format. The scrap contains color tables so that applications can keep custom colors with pictures copied to the clipboard. The scrap has the same format as the Apple Preferred Format picture PALETTES block:

NumColorTables	(+000)	Word	The count of the number of color tables in the scrap
ColorTableArray	(+002)	32 Bytes	The color tables for the scrap. There are NumColorTables of them, each 32 bytes long.

RESOURCE REFERENCE SCRAP (TYPE: $8003)

The Resource Reference scrap is designed to allow resource editors to exchange resource data through an external scrap file using the Scrap Manager.

resScrapType	(+000)	Word	Type of resource (within the resScrapPath file)
resScrapID	(+002)	Long	ID of resource (within the resScrapPath file)
resScrapPath	(+006)	WString	Full GS/OS class-one pathname to an extended file containing the specified resource.

If the specified resource contains references to other resources (for example, an rWindParam1 resource with a title string, control list, control templates, etc.), all the referenced resources must be present in the resScrapPath file.

It is the responsibility of the application using this scrap to handle resource ID conflicts that might arise from a Paste operation. The application should not modify or destroy the resScrapPath file.

Further Reference

- o Apple IIgs Toolbox Reference

- HyperCard IIgs Technical Note #3, Tuning Sampled Sounds
- File Type Note for file type $CA, all auxiliary types, Finder Icons File
- File Type Note for file type $C0, auxiliary type $0002, Apple Preferred Format

### END OF FILE TN.IIGS.099

```
#####################################################################
### FILE: TN.IIGS.100
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#100: VersionVille

Revised by: Matt Deatherage May 1992
Written by: Matt Deatherage January 1991

This Technical Note is all there is to know about versions, version formats and version numbers on the Apple IIgs.

CHANGES SINCE JANUARY 1991: Revised to include System Software 6.0.

VERSION NUMBER FORMATS

There are three kinds of version numbers on the Apple IIgs. Two of the three are documented elsewhere but are repeated here for convenience.

SYSTEM TOOL SET VERSIONS

The Apple IIgs system tools use a one-word version number. The high-order four bits of this word have special meaning. Bits 8-11 are the major version number and bits 0-7 are the minor version number. This is illustrated in Figure 1.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| F| E| D| C| B| A| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 |  |_____||   Major   |        Minor          |
 |     |   |  Release  |       Release         |
 |     |   |___________|_______________________|
 |     |
 |     +---- 1 = Special Features
 +---------- 1 = Prototype
```

Figure 1-Toolbox Version Numbers

Note that this definition is different and supersedes the definition in the Apple IIgs Toolbox Reference for system tool sets. Previous documentation reserves only bit 15 as the prototype bit; this has been expanded. Bits 14-12 of user tool set version words have no special meaning; they are still part of the major release.

```
   NOTE : When comparing the major and minor release version
          numbers to check the installed version of a system tool,
          mask off bits 15-12 first (for example, by using an AND
          #$0FFF instruction).
```

SMARTPORT OR GS/OS DRIVER VERSIONS

GS/OS drivers and SmartPort firmware drivers use an alternate one-word version number, described in Figure 2.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| F| E| D| C| B| A| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|___________|_______________________|___________|
   Major             Minor           $A = Alpha
  Release           Release          $B = Beta
                                     $E = Experimental
                                     $0 = Final
```

Figure 2--GS/OS Driver And SmartPort Version Numbers

APPLE IIGS LONG VERSION FORMAT

Long version format is a 32-bit (two-word) format similar to the standard Macintosh version numbering scheme defined in Macintosh Technical Note #189, Version Territory, except the four bytes are stored least significant byte first, as is standard on the Apple II, and the values of the stage are different. Figure 3 shows the format of a long version.

```
               High word
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|1F|1E|1D|1C|1B|1A|19|18|17|16|15|14|13|12|11|10| ...
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|     Major version     | Minor v.  | Bug vers. |
+-----------------------+-----------+-----------+
 Major version          |Minor      | Bug version
 (2 digits, BCD)        |version (1 | (2 digits, BCD)
 Example:               |digit, BCD)| Example:
 $25 = Version 25       |Example:   | $4 = Version
                        |$0 = Vers. | x.y.4
                         x.0
                                   Low word
                           +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                       ... | F| E| D| C| B| A| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
                           +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                           | Stage  | Must be zero |     Release version   |
                           +--------+--------------+-----------------------+
                      Stage:        |              | Release version (2
                      001 = develop |              | digits, BCD)
                      010 = alpha   |              | Example:
                      011 = beta    |              | Long version value of
                      100 = final   |              | $25048006 = Version
                      110 = release                  25.0.4f6
```

Figure 3-Long Version Numbers

Long version format allows for bug versions, unlike toolbox versions. Also, you can do unsigned long comparisons of long versions to determine which revision is later.

NOTE : If the version stage is 101 (release), the release version must be zero. For example, you may not have version 25.0.4 release 16. "Release version" implies that the product is no longer under development and has no developmental version numbers.

SYSTEM VERSION NUMBERS

The most important of the numerous version numbers in the system are the system tool version numbers. These numbers, passed to LoadTools, LoadOneTool or StartUpTools ensure that you're getting at least the version you want, or maybe a later one. This mechanism is your primary defense against old system software--by requiring the latest tool versions in your application, you are notified by the Tool Locator early in your program if the system has the latest system software installed or not.

Note that ROM 1 and ROM 3 have different version numbers for seven tools under 5.0.4--QuickDraw II, the Scheduler, ADB, SANE, Integer Math, Text Tools and the List Manager. In each case, the ROM 01 version is lower and should be used in your LoadOneTool, LoadTools or StartUpTools calls.
The current revision of Apple IIgs System Software is 6.0. Assuming a correct installation, requiring QuickDraw 3.7 in effect requires System Software 6.0, although you may check the system's rVersion resource in the system resource file if you require more detailed information about the system sovtware version.

System Tool Set Versions

Number	Tool	ROM 1	ROM 3
1	Tool Locator	$0301	$0301
2	Memory Manager	$0302	$0302
3	Misc Tools	$0302	$0302
4	QuickDraw II	$0307	$0307
5	Desk Manager	$0304	$0304
6	Event Manager	$0301	$0301
7	Scheduler	$0300	$0300
8	Sound Tools	$0303	$0303
9	ADB	$0300	$0300
10	SANE	$0300	$0300
11	Integer Math	$0300	$0300
12	Text Tools	$0300	$0300
13	[used internally]	$0300	$0300
14	Window Manager	$0303	$0303
15	Menu Manager	$0303	$0303
16	Control Manager	$0303	$0303
17	[System Loader]	$0400	$0400
18	QuickDraw II Aux	$0304	$0304
19	Print Manager	$0301	$0301
20	Line Edit	$0303	$0303
21	Dialog Manager	$0304	$0304
22	Scrap Manager	$0301	$0301
23	Standard File	$0303	$0303
25	Note Synthesizer	$0104	$0104
26	Note Sequencer	$0104	$0104
27	Font Manager	$0303	$0303
28	List Manager	$0303	$0303

29	ACE	$0103	$0103
30	Resource Manager	$0102	$0102
32	MIDI Tools	$0103	$0103
33	Video Overlay	$0103	$0103
34	Text Edit	$0103	$0103
35	MIDI Synth	$0100	$0100
38	Media Control	$0100	$0100

Toolbox Driver Version Numbers

Driver	Version
ImageWriter II	4.2
ImageWriter LQ	4.2
LaserWriter	3.2
StyleWriter	1.0
Epson	2.0
Printer Port Driver	2.1
Modem Port Driver	2.1
Parallel Card Port Driver	2.0
AppleTalk Port Driver	3.0
Pioneer 4200 (MC)	1.0
Pioneer 2000 (MC)	1.0
Apple CD SC (MC)	1.0

GS/OS Version Numbers

Component	Version
GS/OS	4.1
ProDOS FST	4.1
AppleShare FST	4.0
High Sierra FST	4.0
Character FST	4.0
DOS 3.3 FST	1.2
HFS FST	1.0
Pascal FST	1.0
AFP Driver	4.0
Apple II RAMCard driver	1.0
AppleDisk 3.5 Driver	5.3
AppleDisk 5.25 Driver	2.5
AppleTalk Main Driver	4.0
Console Driver	3.2
RPM Driver	4.0
SCSI CD Driver	6.0
SCSI HD Driver	6.0
SCSI Scanner Driver	6.0
SCSI Tape Driver	6.0
UniDisk 3.5 Driver	3.0

Control Panel Version Numbers

CDev	Version
AppleShare	2.0
Direct Connect Printer	1.1
FolderPriv	1.0
General	2.0

Keyboard	1.1
Media Control	1.1
MIDI	1.0
Modem Port	1.1
Monitor	1.1
Network Printer Namer	1.0
Network Printer Chooser	1.0
Network	1.0
Printer Port	1.1
RAM	1.1
SetStart	1.0
Slots	1.2
Sound	2.0
Time	2.0

Further Reference

- Apple IIgs Toolbox Reference
- GS/OS Reference
- GS/OS Technical Note #1, Contents of System Disk and System Tools
- File Type Note for File Type $C7, Control Panel Devices

### END OF FILE TN.IIGS.100

###
### FILE: TN.IIGS.101
###

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#101: Patching the Toolbox

Revised by: Dave Lyons May 1992
Written by: Dave Lyons May 1991

This Technical Note presents guidelines on when and how to patch Apple IIgs Toolbox functions.

CHANGES SINCE MAY 1991: Added a note about patching the Tool Locator and Desk Manager, and corrected a spelling error.

INTRODUCTION

There is normally no need to patch the toolbox; avoid patching whenever you can. If you must patch a toolbox function, be sure to have a good understanding of the call you're patching and how it interacts with the whole system.

No toolbox patch is risk-free. Future versions of the toolbox could change in ways that make your patch less useful. (For example, if you patched NewControl to have some global effect on controls being created, your patch became less useful when NewControl2 was introduced in System Software 5.0.)

For better compatibility, patch with care! If any parameters passed are outside the range that was allowed when you wrote your patch, just pass the call straight through; the new toolbox probably knows something your patch doesn't.

PATCHING THE TOOLBOX FROM AN APPLICATION

An application can easily patch a function for the duration of that application.

After starting up the tools, construct a Function Pointer Table (FPT) the same size as the existing FPT (call GetTSPtr and examine the first word of the table; multiply it by four to get the size of the FPT in bytes). The first longword of your FPT is the number of functions in the tool set; do not hard-code this value! Get it from the existing FPT on the fly. Fill the rest of your FPT with zeroes, except for the functions you want to patch. You must always patch the BootInit function (the first function) to return no error. Remember that the function pointer values are one less than the addresses of your replacement functions.

On exit, when you call TLShutDown your patch will be automatically removed. (If you're using ShutDownTools, you should call MMShutDown and TLShutDown after you call ShutDownTools.)

Note : In the description of SetTSPtr on page 24-19 of Apple IIgs Toolbox Reference, Volume 2, there are several references to the TPT. Keep in mind that the TPT is the Toolset Pointer Table, not the Function Table Pointer you pass to SetTSPtr. While SetTSPtr copies the TPT to RAM if necessary, it does not make a copy of the FPT. After you call SetTSPtr, the FPT you passed is being used, and any zero values in your table were filled in.

PATCHING THE TOOLBOX FROM A DESK ACCESSORY OR SETUP FILE

A permanent initialization file or Desk Accessory can patch toolbox functions at boot time by constructing an FPT for SetTSPtr, as described for an application, but there is an extra step to make the patch "stick."

Call LoadOneTool and then SetTSPtr; then call SetDefaultTPT (see Apple IIgs Toolbox Reference Volume 3, page 51-16).

It is not safe to call SetDefaultTPT while an application is running (temporary application patches would be made permanent, and later the application would go away). Since there are desk accessories that install other desk accessories while applications are running, desk accessory that wants to install a tool patch should make the class-one GS/OS GetName call; if the null string is returned, no application is executing yet, so it is safe to make the patch. (Otherwise the desk accessory should ask the user to put the desk accessory file in the System:Desk.Accs folder and restart the system.)

PATCHING THE TOOL LOCATOR OR DESK MANAGER

On ROM 3 systems, the SetTSPtr call treats toolsets 1 (Tool Locator) and 5 (Desk Manager) specially, for compatibility with system software versions earlier than 5.0.

You must pass a systemOrUser value of $0001 (not $0000) when patching one of these toolsets, or the SetTSPtr call will have no effect. Passing this special systemOrUser value works for other ROM versions, too--you don't have to check the ROM version.

AVOID TAIL PATCHING

The best kind of patch is a pre-patch or head patch: it does some extra work and then jumps to the original function (as found in the FPT before applying the patch). Make sure the A, X, and Y registers contain the same values when you jump to the original function as they did when the patch got control.

A "tail patch" which calls the original function and then regains control is much more of a compatibility risk, because there are several instances where System Software patches examine return addresses to fix problems in large toolbox calls which call small ones (by patching the small one to realize it's being called from the big one, many K of RAM remain available to your application).

If you tail patch a function which the system already patched, you may prevent the toolbox from working correctly.

PATCHING THE TOOL DISPATCHER

If you need to patch a large number of functions, especially for a general purpose utility like a debugger, it may make more sense to patch the tool dispatcher vectors instead of patching individual functions. See Apple IIgs Technical Note #87, Patching the Tool Dispatcher.

Further Reference

- Apple IIgs Toolbox Reference
- Apple IIgs Technical Note #87, Patching the Tool Dispatcher

### END OF FILE TN.IIGS.101

```
#######################################################################
### FILE: TN.IIGS.102
#######################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#102: Various Vectors

Revised by: Dave Lyons — May 1992
Written by: Dave Lyons — December 1991

This Technical Note describes system vectors that are not fully described in other documentation.

CHANGES SINCE DECEMBER 1991: Added information about the TOBRAMSETUP vector.

THE TOBRAMSETUP VECTOR

The TOBRAMSETUP vector is documented in Appendix D of the Apple IIgs Firmware Reference. Two clarifications are needed:

- o TOBRAMSETUP must be called in 8-bit native mode (SEP #$30).
- o Before System 6.0, TOBRAMSETUP required that the Bank register be $00 (bad things would happen if it was not). This requirement is gone in 6.0.

THE MOVE_INFO VECTOR

MOVE_INFO is a flexible, low-overhead data transfer routine. It can transfer buffer-to-buffer, buffer-to-location, location-to-buffer, and buffer-to-buffer reversing the order of the bytes.

Apple IIgs GS/OS Device Driver Reference tells you how to call MOVE_INFO from a GS/OS driver environment (JSL to $01FC70), but this requires the language-card RAM to be banked in correctly.

Another vector points to the same routine: $E10200. If you aren't a GS/OS device driver, it is more convenient to JSL to $E10200, because you don't have to worry about banking in the $01FCxx vectors. The $E10200 vector is available whenever GS/OS is active, under System Software 5.0 or later.

THE DYN_SLOT_ARBITER AND SET_SYS_SPEED VECTORS

Two other GS/OS System Service vectors are duplicated in bank $E1: SET_SYS_SPEED ($E10204) and DYN_SLOT_ARBITER ($E10208). Like MOVE_INFO, these are available when GS/OS is active under System Software 5.0 or later.

Further Reference

- Apple IIgs GS/OS Device Driver Reference
- Apple IIgs Firmware Reference

### END OF FILE TN.IIGS.102

```
#####################################################################
### FILE: TN.IIGS.103
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#103: Inline Procedure Name Format

Modified by: Matt Deatherage May 1992
Written by: Dave Lyons December 1991

This Technical Note describes a simple format for imbedding procedure names in object code, for use by debugging utilities.

CHANGES SINCE DECEMBER 1991: Changed &syscnt to &SYSCNT so it works with the CASE ON APW directive. Clarified the possible addition of parameters after the Pascal string.

GSBug 1.5b18 and later support a simple convention for including procedure names inline in the object code, for debugging purposes.

INLINE NAME FORMAT

```
       82 xx xx                     brl  pastName
       71 77                        dc.w $7771
       nn xx xx xx xx...            str  'the name string'
                          pastName  ...
```

That is, an imbedded name is a BRL around a signature word and a Pascal string. The name string can theoretically be up to 255 characters long, but in practice only short names are useful. For example, GSBug displays only the first 15 characters of a name when it is encountered, and only the first 11 when it appears as the operand of a JSR or JSL instruction.

Names in this format always start with a BRL, not a BRA or JMP. Signature word values other than $7771 are reserved for future definition, and more information may be added after the Pascal string.

Be careful what you name!

Be careful not to name something important--like a table, or a label from which you compute other addresses. The extra bytes generated by the inline name would mess up your calculations. If you name a heartbeat task, out-of-memory queue routine, or other construction that needs a special header, be sure to put the name where the executable code starts, not at the beginning of the header.

APW ASSEMBLY MACRO

The following macro is for the APW assembler. If you equate DebugSymbols to zero, the macro generates no object code. If DebugSymbols is nonzero, the

macro generates an inline name corresponding to its label.
Use the name macro anywhere you would use a label. For example:

```
DebugSymbols   GEQU 1
...
CountItems     name
```

The macro:

```
     MACRO
&lab name
&lab anop
     aif DebugSymbols=0,.pastName
     brl pastName&SYSCNT
     dc i'$7771'
     dc i1'L:&lab',c'&lab'
pastName&SYSCNT anop
.pastName
     MEND
```

MPW IIgs Assembly Macros

The following macros are for the MPW IIgs assembler. If you equate DebugSymbols to zero, the macros generate no object code. If DebugSymbols is nonzero, the macros generate inline names corresponding to their labels.

Use the name macro anywhere you would use a label. Use the procname macro in place of a proc directive, at the beginning of a procedure. For example:

```
DebugSymbols   equ 1
...
CountItems     name
TaskLoop       procname
```

The macros:

```
              macro
&lab          name
&lab
              if DebugSymbols<>0 then
              brl @pastName
              lclc &olds
&olds         setc &setting('string')
              string asis
              dc.w $7771
              dc.b &len(&lab),'&lab'
              string &olds
@pastName
              endif
              mend

* You can use procname instead of proc

              macro
&lab          procname &x
&lab          proc     &x
              if DebugSymbols<>0 then
              brl @pastName
              lclc &olds
&olds         setc &setting('string')
              string asis
              dc.w $7771
              dc.b &len(&lab),'&lab'
              string &olds
@pastName
              endif
              mend
```

WRITING UTILITIES THAT RECOGNIZE INLINE NAMES

If you write a utility that recognizes inline procedure names in this format, check for a signature word of $777x, not specifically $7771. This allows more information to be added to the format later (a signature of $7772 could mean there is a Pascal string followed by parameter-passing information, for example).

### END OF FILE TN.IIGS.103

```
#####################################################################
### FILE: TN.IIGS.104
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#104:   Font Manager Fundamentals

This Technical Note discusses information and philosophy of that typographical toolset, the Font Manager.

FixFontMenu only works once per FMStartUp

You may have noticed that none of the Font Manager calls that translate font family numbers to menu item IDs (or vice-versa) require a menu ID as a parameter. That's because the Font Manager was designed with the idea that an application would only need one font menu, so it keeps one correspondence in private static storage.

This means that once someone has called FixFontMenu, any later FixFontMenu call during that Font Manager session will destroy the results of the first one, unless all the parameters are identical. The Font Manager doesn't remove the font menu items, but it does not return the correct results from FamNum2ItemID or ItemID2FamNum.

This means if you're a new desk accessory, the Font Manager can't help you create a font menu--attempting to use FixFontMenu will make any application font menu useless. You can use Font Manager routines such as CountFamiles, FindFamily and GetFamInfo to obtain all the information necessary to build your own font menu (or font choosing dialog box, for that matter--but if you create a dialog for an NDA, remember that it has to fit in 320 mode also).

Font styling requires QuickDraw Auxiliary

The Font Manager can't create fonts with outline, shadow or italic styles unless QuickDraw Auxiliary (tool set #18) is present and started. These facts are mentioned in pieces other places, but not in one place--if you want normal Font Manager operations, you must load and start QuickDraw Auxiliary.

Further Reference

---

o   Apple IIgs Toolbox Reference, Volumes 1-3

### END OF FILE TN.IIGS.104

```
#####################################################################
### FILE: TN.IIGS.105
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#105: We Interrupt This CPU...

Written by: Matt Deatherage May 1992

This Technical Note supplements the discussion of how interrupts generally work (or don't work) on the Apple IIgs found in the Apple IIgs Firmware Reference. It also discusses how to patch into the interrupt chain and when not to use software interrupts.

THIS NOTE IS A SUPPLEMENT

That's right, a supplement. This is not the definitive, end-all discussion of interrupts on the Apple IIgs. Most of the information you need to know is available, and has been for several years, in the Apple IIgs Firmware Reference. If you're going to write an interrupt routine, you need to read Chapter 6 of the Firmware Reference.

No excuses. If you don't have the book, buy it or borrow it. People who use your software don't want to hear a sad story about how you wanted to spend the money on a couple of CDs instead of preventing their machine from crashing.

If you haven't read Chapter 6 of the Firmware Reference, do so before continuing; the rest of this Note will make much more sense if you're familiar with the material covered in that chapter.

A NOTE ABOUT TIMING

There are lots of times listed in this Note, concerning how fast certain kinds of interrupts must be serviced before they're lost. Please remember that all times listed are ideal times--actual times are likely to be shorter. For example, a maximum response time of a millisecond means you have one millisecond from the time the peripheral asserts the /IRQ line until the interrupt must be serviced. If interrupts are disabled for the first 750 microseconds (us) of that, then your maximum response time is 250 us. This is why we constantly remind programmers to keep interrupts disabled for absolutely the shortest time possible. Also, all times reflecting serial or AppleTalk interrupts already take into account the serial chip's internal 3-byte buffer.

SO WHAT THE HECK ARE ALL THOSE VECTORS?

At first, looking at all those various vectors seems pretty darned intimidating. However, the structure becomes clearer when you think about interrupt priority.

Some microprocessors allow interrupt requests to have priorities--higher priority interrupts can interrupt lower priority ones. The 65816 doesn't have this capability, so the best the Apple IIgs can do is check possible interrupt sources in highest-priority-first order. For example, AppleTalk interrupts must always be processed extremely quickly--from the time an AppleTalk interrupt is asserted, someone must read the data from the SCC within a maximum of 104.167 us or data can be lost. That's not very much time at all, especially considering that the system may have interrupts disabled, or may be running at 1 MHz speed when the interrupt fires.

Serial interrupts are next--at 19,200 baud, there's a maximum of 1.094 milliseconds to read data before it's lost. (Multiplication shows that 38,400 baud has a maximum of 547 us, and 57,600 baud has a maximum delay of 273.5 us. Not much at all.)

You'd hope the Interrupt Manager in ROM would be smart enough to service AppleTalk interrupts first and serial interrupts next, and in fact that's what it does. In fact, it services them so fast that not all the system information is saved before checking the hardware and dispatching (if necessary) to the IRQ.APTALK or IRQ.SERIAL vectors. See Apple IIgs Technical Note #24 for more information on which system state information isn't saved before calling those vectors.

The list of interrupt priorities is on page 180 of the Firmware Reference. What's not clear from any description of interrupt handling is that each internal interrupt source's vector is only called if the Interrupt Manager determines it is the source of the interrupt. For example, the IRQ.DSKACC vector is not called unless the user pressed Command-Control-Esc to generate the interrupt. This insures that external interrupt handlers for slot-based peripherals are dispatched to as quickly as possible--if each vectored routine had to determine interrupt ownership, every interrupt would have significantly more overhead.

There are two additions to the priority list in the Firmware Reference--the first is also an exception to the "interrupt handlers don't have to identify the interrupt" rule. On ROM 3 machines only, vector $E1021C (IRQ.MIDI) gets control immediately after determining the interrupt isn't an AppleTalk interrupt. MIDI data can come in so quickly that it needs higher priority than serial interrupts. However, to improve performance, routines called through this vector must return as fast as possible (faster would be better) to avoid delaying interrupts further down the chain, like serial interrupts. Also note that this vector doesn't exist on ROM 1.

The second addition is to the final priority, simply defined as "external slot." The documentation doesn't clearly indicate how this works--it kind of implies this is just calling IRQ.OTHER. In fact, if no IRQ.OTHER routine claims the interrupt, the system does some voodoo magic to switch to emulation mode and jumps through the vector at $03FE, just like all previous Apple II models. And just like in older systems, whatever code is pointed to by $03FE must end with an RTI instruction. This behavior is preserved for compatibility, although it is the slowest interrupt response available on the IIgs.

GETTING CONTROL IN TIME

Passing control to external handlers isn't always quick enough for some people. If you're writing a telecommunications program, for example, you have no more than 1.094 ms from the time a character is received to get it out of the SCC or you'll lose data at 19,200 baud.

The Interrupt Manager is a very tight piece of code--if it were running in RAM and the system was temporarily slowed down to 1 MHz, there would only be room for about two more instructions before AppleTalk would lose data. Since AppleTalk has to be serviced within 104.2 us (as discussed previously), and since IRQ.SERIAL is called as quickly as possible after IRQ.APTALK (the only delay is if you're on ROM 3 and a non-trivial MIDI interrupt handler is installed), patching in at IRQ.SERIAL poses no problems for most high-speed communications, even up to 57,600 baud. In other words, it's not necessary to patch any vector other than IRQ.SERIAL to achieve the results you want.
The problem comes when you have external communications hardware--making it through the internal interrupt chain is too slow if your external communications hardware has the same kinds of limitation the SCC does (namely, a 3-byte internal buffer). External vectors are only called after all the internal sources verify it's not their interrupt, and by that time your card may have lost data.

PATCHING THE MAIN INTERRUPT VECTOR

In these cases, where there is no possible way to service an interrupt in time through the Interrupt Manager's normal priority chain, and in these cases only, it's acceptable to patch out the main interrupt vector at $E10010 (preferably using GetVector and SetVector with reference number $0004). But even then, there are rules to follow.

1. You should duplicate the functionality of the main interrupt vector exactly until the point where you must gain control or lose data. For example, if your card requires that you service interrupts within a millisecond or lose data, AppleTalk interrupts still have higher priority over your interrupts because AppleTalk interrupts must be serviced within 104 us.   In this example case, your code should duplicate the functionality of the Interrupt Manager up through and including the call to IRQ.APTALK, and then (and only then) call your interrupt handler, where you handle the interrupt if it's yours and pass control to the rest of the interrupt chain if it's not.

2. You should only service your interrupts before AppleTalk if your interrupts require servicing in less than 104 us. If they don't, give AppleTalk first shot. If they do, you must clearly inform the user, both in documentation and on the screen, that if they proceed with this function network services may be interrupted, and that they may have to restart the system to restore them. Users must also have the option to back out and cancel at this point. No, this isn't a pleasant message to deliver, but it's much nicer than to completely disconnect AppleTalk and lock up the system if it was booted from a server.

3. You should only patch out the main interrupt vector when absolutely necessary. For example, if you're communicating with hardware that runs at multiple speeds and only the highest speed generates interrupts that require patching the main vector, you should not be patching the main vector when not using that highest speed. For

telecommunication programs, this means different interrupt handling routines depending on baud rates. To do this any other way lessens the reliability of other high-speed interrupt-driven peripherals in the system.

And remember, it's only acceptable to patch the main interrupt vector when there is no other way to service interrupts fast enough. At all other times, even in the same program, service your interrupts in other ways.

VECTORS VS. BINDING VS. ALLOCATING

There are three main ways to get into the IIgs interrupt-handling chain--by patching vectors directly, by using the ProDOS 8 or ProDOS 16 call ALLOC_INTERRUPT, and by using the GS/OS call BindInt. Each behaves differently and has advantages and disadvantages. We'll go from the highest level to the lowest in discussing them.

BINDINT--EASY TO USE, BUT NOT AS EASY TO CONTROL

BindInt's vector reference numbers (VRNs) are designed to correspond to vectors in the IIgs Interrupt Manager's chain. Comparing the list of numbers on page 265 of GS/OS Reference to the list of vectors starting on page 266 of the Apple IIgs Firmware Reference will make this more obvious.

When you call BindInt, GS/OS replaces the address in the appropriate interrupt vector with an address inside GS/OS. The routine it points to calls all the routines bound to that vector, including the one that was originally installed (usually the ROM's built-in SEC/RTL address). That is, if IRQ.VBL pointed to the Miscellaneous Tools' Heartbeat Task code before a program made four separate BindInt calls to VRN $000C, then after those calls completed, IRQ.VBL would point to code inside GS/OS that called all four bound routines and the Miscellaneous Tools' Heartbeat Task code.

This is why each bound routine is told (through the microprocessor's carry flag) if one of the other routines has already claimed the interrupt and why preserving that status is important. BindInt is a convenient way to get code time during various kinds of interrupts, but you should note that you can't control in what order bound handlers are called.

ALLOC_INTERRUPT--OLD STYLE INTERRUPT MANAGEMENT

ALLOC_INTERRUPT and the ProDOS 8 equivalent, ALLOC_INT take the address of the routine you pass and keep it in an internal table. When an interrupt occurs, each address in the table is called in turn until one of the interrupt handlers claims it. In older days, failure by any of the installed interrupt handlers to claim the interrupt would bring the system to a crashing halt--nowadays unclaimed interrupts are ignored by both ProDOS 8 and GS/OS.

What the manuals don't tell you is that any routine installed in this way is called after the system has jumped through address $03FE in bank zero--in other words, at the last possible chance. For any kind of timing-sensitive interrupts, these routines are not sufficient.

The table that stores these routines is of a fixed size--ProDOS 8's table holds four routines, and GS/OS's holds 16. If you try to install more handlers than that, you'll get an error from the operating system.

PATCHING VECTORS--HIGH LEVEL OF CONTROL, HIGH RISK

The lowest level at which you can get control is by directly patching the Interrupt Manager's vectors as documented in the Firmware Reference. Although this lets you get control as soon as the Interrupt Manager determines which vector to call, it also carries some compatibility risks.

Any BindInt calls with VRNs that reference a vector you patch make GS/OS take your routine's address and store it internally. This is a problem for anyone who daisy-chained into the same interrupt vector after you did--there's no good way to disconnect yourself without disconnecting everyone who patched in after you. This is Bad.

If you patch vectors directly, you have to check the vector when you're ready to remove your routine. If the vector doesn't still point to your address, someone else has patched into the vector after you and you can't remove yourself. In these cases, you have to leave a "code stub" that takes no action other than passing control along to the address that was installed when you patched in, and you have to leave that code stub at the same address as your interrupt handler. (Since you don't know who has patched the vector after you, you have no way to communicate with those programs and tell them you're going away.)

This means your interrupt handler can't be in your main program. If it is, when GS/OS calls UserShutDown to remove your program from memory, you'll orphan one or more pointers to your interrupt handler (which doesn't exist anymore). You must allocate memory and load your interrupt handler with a different user ID than your main program so your code stub can survive when your program quits. Also note that this means repeated launchings of your program could leave lots and lots of code stubs in memory--so if you can find a way other than patching vectors directly, you're encouraged to use it.

SOFTWARE INTERRUPTS--BRK AND COP

Sometimes developers forget that BRK and COP instructions are in fact software interrupts--when the IIgs's 65816 encounters one of these instructions, it goes through the same Interrupt Manager procedures that all interrupts go through.

Among other things, this means that encountering one of these instructions inside an interrupt routine will overwrite all the system's saved information (such as registers or system state variables) with new ones, meaning you'll never be able to return from the first interrupt. This isn't too much of a problem with BRK (except when debugging interrupt routines), but a recent fad popularity for COP makes this worth mentioning.

Some developers are trying to use COP instructions for all kinds of general-purpose mechanisms, but the system is not designed to handle this. Using a COP instruction to pass control to a shell or a library routine in production-level code is not acceptable for several reasons. First, any COP instruction inside an interrupt handler will bring the system to its knees. Second, there is no arbitration for the COP vector so multiple users of it will collide. Third, although a COP instruction takes only two bytes, it takes many hundreds more cycles to execute than a JSL instruction, slowing the system down for no reason.

COP instructions are perfectly acceptable in non-production level (debugging) code, but developers should not use them as a way for different program modules to communicate. Such use is not supported and is strongly discouraged by Apple.

BEFORE WE RTI--A SUMMARY

This Note covers many issues concerning interrupts, so here's a summary. This isn't all the explanation--refer to individual topics for discussions and reasons.

- Never disable interrupts for longer than necessary--you make life really difficult on routines that rely on high-speed interrupt capability.

- Interrupt routines should patch in as late as possible in the interrupt process without losing data. If your interrupt source doesn't need servicing as fast as AppleTalk does, don't patch in before AppleTalk.

- Patching the main interrupt vector at $E10010 is only acceptable if there's no possible way to service external interrupts quickly enough (internal interrupt sources, like serial ports, should always use other vectors), and even then the vector most only be patched while necessary; if a slower interrupt source is used in the same program, unpatch the vector.

- Different methods of installing interrupt handlers give you different levels of control. BindInt is the overall best method, although you can't control in what order bound routines are called.

- COP instructions are unacceptable in non-debugging code; they should never take the place of JSL instructions or other methods of inter-process communication.

Further Reference

- Apple IIgs Firmware Reference
- Apple IIgs Toolbox Reference, Volume 3
- GS/OS Reference
- ProDOS 8 Technical Reference Manual
- Apple IIgs Technical Note #24, Apple IIgs Toolbox Reference updates

### END OF FILE TN.IIGS.105

```
#####################################################################
### FILE: TN.IIGS.106
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs #106: ADB Addendum

Written by: Dave Lyons

May 1992

This Technical Note documents some bits in the ADB SendInfo data byte for setModes and clearModes.

SendInfo is documented in Volume 1 of Apple IIgs Toolbox Reference, but it doesn't tell you what any of the bits in the setModes/clearModes data byte are for. Well, here are the useful ones:

Bit	Value	Description
6	$40	Shift+CapsLock=Lowercase mode
4	$10	Keyboard buffering
3	$08	Dual-speed keys
2	$04	Fast space/delete keys

For example, to turn off keyboard buffering without altering the user's Battery RAM, you can do the following:

```
             pea 1                   ;number of data bytes
             pushlong #modesToClear  ;pointer to data byte
             pea 5                   ;modeCmd = clearModes
             _SendInfo
             ...

modesToClear dc.b $10                ;bit 4 = keyboard buffering
```

Note that the user's control panel setting will become current again if they hit Command-Ctrl-Esc (the system calls the TOBRAMSETUP vector at $E10094 to update the system from Battery RAM).

Further Reference

o   Apple IIgs Toolbox Reference, Volume 1

### END OF FILE TN.IIGS.106

```
#####################################################################
### FILE: TN.IIGS.107
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple IIgs #107: Tool Locator Tribulations

Written by: Dave Lyons May 1992

This Technical Note tells you what to watch out for in the Tool Locator.

SHUTDOWNTOOLS AND SYSTEM 6.0

In System 6.0, ShutDownTools inappropriately calls HideCursor even if QuickDraw II is not started. The results are unpredictable.

If your application does not use QuickDraw II but does use ShutDownTools, you may need to start up and shut down your tools manually instead.

Note that the HideCursor problem does not occur in the (unusual) case that the System 6.0 noResourceMgr bit (value $0010) is set.

CONTENTS OF THE STARTUPTOOLS TOOL TABLE

You should not include the Tool Locator or Memory Manager in your tool table. Instead, call TLStartUp and MMStartUp before calling StartUpTools, and call MMShutDown and TLShutDown after ShutDownTools.

Since StartUpTools automatically starts the Resource Manager for you, you should not include Resource Manager in the tool table, either. Doing so has no ill effect in System Software 6.0 and later, but in System Software 5.0 through 5.0.4 you got duplicate ResourceStartUp and ResourceShutDown calls.

The order of the tool table entries does not matter. (Toolbox Reference 3, page 51-8, says "Although StartUpTools handles the order of tool startup for you...", but this is widely overlooked.)

Further Reference

o Apple IIgs Toolbox Reference, Volume 3

### END OF FILE TN.IIGS.107

###
### FILE: TN.IIGS.108
###

Apple II
Technical Notes

Developer Technical Support

Apple IIgs
#108: Finder Funkiness

Written by: Dave Lyons

May 1992

This Technical Note tells you what to watch out for in Finder 6.0.

ICON SEARCH ORDER

When the Finder looks for an icon it uses the first match it finds. When more than one icon would match, the order is important.

Some icons are built right into the Finder's resource fork--those are always searched last. Other than that, the Finder searches in device-number order (for example, icons on device number $0001, the boot device, override icons on other devices).

On each disk, icons in Desktop files override icons in old-style icon files. Among old-style icon files in the same Icons folder, each icon file overrides subsequent (as returned by GetDirEntry) files in the same directory. Within an icons file, earlier icons override later icons.

If you create a "generic" icon that matches a broad class of files--for example, all files of a particular file type--you have to be very careful where you put that icon. A generic icon in any file's rBundle will wind up in a Desktop file, where it will override some old-style icon files (or all of them, if the Desktop file is on the boot disk).

There's really no good place for a custom generic icon. (Well, the Finder's resource fork would be a good place, but we recommend not messing with that.) A halfway-good place is in old-style icons folders, at the end, on the highest-numbered convenient device (for example, your third hard drive partition of three).

Note that the 6.0 Finder's matching order for old-style icons is more or less the reverse of what it was in previous versions.

FILENAME MATCHING AND WILDCARDS

When an icon matches by filename and has a leading wildcard, the match always fails if there are any lowercase characters in the string. For example, "*.TXT" is fine, but "*.Txt" never matches.

Also, a leading wildcard matches one or more characters, instead of (as intended) zero or more characters. "*ICONS" matches "MyIcons" and "Other.Icons", but not "Icons". You can usually work around this by omitting the character after the wildcards: "*CONS" matches all three.

These notes apply both to old-style icon files and to new matchFilename structures.

SHUT DOWN DEFAULT IS NOT CONFIGURABLE

The System 6.0 Finder Documentation shows one of the words in the rRectList(1) resource as the default choice for the Shut Down dialog. Actually, the default is not configurable, and this word in the resource should remain zero. Utilities can customize the Finder's "Shut Down..." command by accepting the finderSaysMItemSelected request.

Further Reference

- System 6.0 Documentation

### END OF FILE TN.IIGS.108

```
#####################################################################
### 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:                  *
0000:                  *  80-column screen dump
0000:                  *
0000:                  *  By
0000:                  *     Greg Seitz
0000:                  *     12-Jul-84
0000:                  *
0000:                  *  This program will allow you to dump the contents
0000:                  *  of your 80-column text screen to whatever device is
0000:                  *  connected through COUT.  If it is still connected to
0000:                  *  the screen, you will obviously be printing back
0000:                  *  what you were reading.
0000:                  *
0000:        FBC1      BASCALC   EQU   $FBC1         ;convert A reg to line addr
on scrn
0000:        FDED      COUT      EQU   $FDED         ;A register out as ASCII
0000:        C001      SET80COL  EQU   $C001         ;enable page 1/2 switches to
control aux
0000:        C055      TXTPAGE2  EQU   $C055         ;page 2 or Aux depending
0000:        C054      TXTPAGE1  EQU   $C054         ;page 1 or main depending
0000:        0028      BASL      EQU   $28           ;BASCALC puts base addr. here
0000:        0029      BASH      EQU   $29           ;and high byte here.
0000:                  *
1000:        1000                ORG   $1000         ;or anywhere
1000:        1000      SCREENDMP EQU   *
1000:A2 00                       LDX   #0            ;START AT LINE 0
1002:                  *
1002:8A                SCRNLP    TXA                 ;CALL BASCALC
1003:20 C1 FB                    JSR   BASCALC       ;FOR ADDRESS OF LINE X
1006:A0 00                       LDY   #00           ;DO 80 CHARS STARTING FROM
CHARACTER 0
1008:                  *
1008:        1008      SCRNLP2   EQU   *
1008:8D 01 C0                    STA   SET80COL      ;SET UP FOR MAIN/AUX
SWITCHING
100B:8D 55 C0                    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 2.2

Revised by: Jim Luther   May 1991
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.

Changes since November 1988: Converted the identification routine from Apple II Assembler/Editor (EDASM) source code to Apple IIgs Programmer's Workshop (APW) Assembler source code. Added the Apple IIe Card for the Macintosh LC to the identification routine's lookup table and memory check routine. Made minor corrections to text.

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 (2.2) of the identification routine returns several things:

o   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

$06 = Apple IIe Card for the Macintosh LC

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 three versions of the IIgs ROM (there will always be some owners who have not yet upgraded from ROM 00 to ROM 01). These versions are identified starting at $01 for the earliest. Therefore, the current IIc will return ROMlevel = $05, the current IIgs will return ROMlevel = $03, 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

```
            keep ID2.2

            list on

            org $2000

            longa off
            longi off

*********************************************
*                                           *
*  Apple II Family Identification Program   *
*                                           *
*               Version 2.2                 *
*                                           *
*               March, 1990                 *
*                                           *
*  Includes support for the Apple IIe Card  *
*  for the Macintosh LC.                    *
*                                           *
*********************************************


;  First, some global equates for the routine:
```

```
PROGRAM    start

IIplain    equ $01          ;Apple II
IIplus     equ $02          ;Apple II+
IIIem      equ $03          ;Apple /// in emulation mode
IIe        equ $04          ;Apple IIe
IIc        equ $05          ;Apple IIc
IIeCard    equ $06          ;Apple IIe Card for the Macintosh LC

safe       equ $0001        ;start of code relocated to zp
location   equ $06          ;zero page location to use

test1      equ $AA          ;test byte #1
test2      equ $55          ;lsr of test1
test3      equ $88          ;test byte #3
test4      equ $EE          ;test byte #4

begpage1   equ $400         ;beginning of text page 1
begpage2   equ $800         ;beginning of text page 2
begsprse   equ $C00         ;byte after text page 2

clr80col   equ $C000        ;disable 80-column store
set80col   equ $C001        ;enable 80-column store
rdmainram  equ $C002        ;read main ram
rdcardram  equ $C003        ;read aux ram
wrmainram  equ $C004        ;write main ram
wrcardram  equ $C005        ;write aux ram
rdramrd    equ $C013        ;are we reading aux ram?
rdaltzp    equ $C016        ;are we reading aux zero page?
rd80col    equ $C018        ;are we using 80-columns?
rdtext     equ $C01A        ;read if text is displayed
rdpage2    equ $C01C        ;read if page 2 is displayed
txtclr     equ $C050        ;switch in graphics
txtset     equ $C051        ;switch in text
txtpage1   equ $C054        ;switch in page 1
txtpage2   equ $C055        ;switch in page 2
ramin      equ $C080        ;read LC bank 2, write protected
romin      equ $C081        ;read ROM, 2 reads write enable LC
lcbank1    equ $C08B        ;LC bank 1 enable

lc1        equ $E000        ;bytes to save for LC
lc2        equ $D000        ;save/restore routine
lc3        equ $D400
lc4        equ $D800

idroutine  equ $FE1F        ;IIgs id routine

;  Start by saving the state of the language card banks and
;  by switching in main ROM.

strt       php              ;save the processor state
           sei              ;before disabling interrupts
           lda lc1          ;save four bytes from
           sta save         ;ROM/RAM area for later
           lda lc2          ;restoring of RAM/ROM
           sta save+1       ;to original condition
           lda lc3
           sta save+2
           lda lc4
           sta save+3
           lda $C081        ;read ROM
           lda $C081
           lda #0           ;start by assuming unknown machine
           sta machine
           sta romlevel

IdStart    lda location     ;save zero page locations
           sta save+4       ;for later restoration
           lda location+1
           sta save+5
           lda #$FB         ;all ID bytes are in page $FB
           sta location+1   ;save in zero page as high byte
           ldx #0           ;init pointer to start of ID table
loop       lda IDTable,x    ;get the machine we are testing for
           sta machine      ;and save it
           lda IDTable+1,x  ;get the ROM level we are testing for
           sta romlevel     ;and save it
           ora machine      ;are both zero?
           beq matched      ;yes - at end of list - leave

loop2      inx              ;bump index to loc/byte pair to test
           inx
           lda IDTable,x    ;get the byte that should be in ROM
           beq matched      ;if zero, we're at end of list
           sta location     ;save in zero page

           ldy #0           ;init index for indirect addressing
           lda IDTable+1,x  ;get the byte that should be in ROM
           cmp (Location),y ;is it there?
           beq loop2        ;yes, so keep on looping

loop3      inx              ;we didn't match. Scoot to the end of the
           inx              ;line in the ID table so we can start
           lda IDTable,x    ;checking for another machine
           bne loop3
           inx              ;point to start of next line
           bne loop         ;should always be taken

matched    anop

;  Here we check the 16-bit ID routine at idroutine ($FE1F).  If it
;  returns with carry clear, we call it again in 16-bit
;  mode to provide more information on the machine.

idIIgs     sec              ;set the carry bit
           jsr idroutine    ;Apple IIgs ID Routine
           bcc idIIgs2      ;it's a IIgs or equivalent
           jmp IIgsOut      ;nope, go check memory
idIIgs2    lda machine      ;get the value for machine
           ora #$80         ;and set the high bit
           sta machine      ;put it back
           clc              ;get ready to switch into native mode
           xce
           php              ;save the processor status
           rep #$30         ;sets 16-bit registers
           longa on
           longi on
           jsr idroutine    ;call the ID routine again
           sta IIgsA        ;16-bit store!
           stx IIgsX        ;16-bit store!
           sty IIgsY        ;16-bit store!
           plp              ;restores 8-bit registers
           xce              ;switches back to whatever it was before
           longa off
           longi off

           ldy IIgsY        ;get the ROM vers number (starts at 0)
           cpy #$02         ;is it ROM 01 or 00?
           bcs idIIgs3      ;if not, don't increment
           iny              ;bump it up for romlevel
idIIgs3    sty romlevel     ;and put it there
           cpy #$01         ;is it the first ROM?
           bne IIgsOut      ;no, go on with things
           lda IIgsY+1      ;check the other byte too
           bne IIgsOut      ;nope, it's a IIgs successor
           lda #$7F         ;fix faulty ROM 00 on the IIgs
           sta IIgsA
IIgsOut    anop

*****************************************
* This part of the code checks for the  *
* memory configuration of the machine.  *
* If it's a IIgs, we've already stored  *
* the total memory from above.  If it's *
* a IIc or a IIe Card, we know it's     *
* 128K; if it's a ][+, we know it's at  *
* least 48K and maybe 64K.  We won't    *
* check for less than 48K, since that's *
* a really rare circumstance.           *
*****************************************

exit       lda machine      ;get the machine kind
           bmi exit128      ;it's a 16-bit machine (has 128K)
           cmp #IIc         ;is it a IIc?
           beq exit128      ;yup, it's got 128K
           cmp #IIeCard     ;is it a IIe Card?
           beq exit128      ;yes, it's got 128K
           cmp #IIe         ;is it a IIe?
           bne contexit     ;yes, go muck with aux memory
           jmp muckaux
contexit   cmp #IIIem       ;is it a /// in emulation?
           bne exitII       ;nope, it's a ][ or ][+
           lda #48          ;/// emulation has 48K
           jmp exita
exit128    lda #128         ;128K
exita      sta memory
exit1      lda lc1          ;time to restore the LC
           cmp save         ;if all 4 bytes are the same
           bne exit2        ;then LC was never on so
           lda lc2          ;do nothing
           cmp save+1
           bne exit2
           lda lc3
           cmp save+2
           bne exit2
           lda lc4
           cmp save+3
           beq exit6
exit2      lda $C088        ;no match! so turn first LC
           lda lc1          ;bank on and check
           cmp save
           beq exit3
           lda $C080
           jmp exit6
exit3      lda lc2
           cmp save+1       ;if all locations check
           beq exit4        ;then do more more else
           lda $C080        ;turn on bank 2
           jmp exit6
exit4      lda lc3          ;check second byte in bank 1
           cmp save+2
           beq exit5
           lda $C080        ;select bank 2
           jmp exit6
exit5      lda lc4          ;check third byte in bank 1
           cmp save+3
           beq exit6
           lda $C080        ;select bank 2
exit6      plp              ;restore interrupt status
           lda save+4       ;put zero page back
           sta location
           lda save+5       ;like we found it
           sta location+1
           rts              ;and go home.

exitII     lda lcbank1      ;force in language card
           lda lcbank1      ;bank 1
           ldx lc2          ;save the byte there
           lda #test1       ;use this as a test byte
           sta lc2
           eor lc2          ;if the same, should return zero
           bne noLC
           lsr lc2          ;check twice just to be sure
           lda #test2       ;this is the shifted value
           eor lc2          ;here's the second check
           bne noLC
           stx lc2          ;put it back!
           lda #64          ;there's 64K here
           jmp exita
noLC       lda #48          ;no restore - no LC!
           jmp exita        ;and get out of here

muckaux    ldx rdtext       ;remember graphics in X
           lda rdpage2      ;remember current video display
           asl A            ;in the carry bit
           lda #test3       ;another test character
           bit rd80col      ;remember video mode in N
           sta set80col     ;enable 80-column store
           php              ;save N and C flags
           sta txtpage2     ;set page two
           sta txtset       ;set text
           ldy begpage1     ;save first character
           sta begpage1     ;and replace it with test character
           lda begpage1     ;get it back
           sty begpage1     ;and put back what was there
           plp
           bcs muck2        ;stay in page 2
           sta txtpage1     ;restore page 1
muck1      bmi muck2        ;stay in 80-columns
           sta $c000        ;turn off 80-columns
muck2      tay              ;save returned character
           txa              ;get graphics/text setting
           bmi muck3
           sta txtclr       ;turn graphics back on
muck3      cpy #test3       ;finally compare it
           bne nocard       ;no 80-column card!
           lda rdramrd      ;is aux memory being read?
           bmi muck128      ;yup, there's 128K!
           lda rdaltzp      ;is aux zero page used?
           bmi muck128      ;yup!
           ldy #done-start
move       ldx start-1,y    ;swap section of zero page
           lda |safe-1,y    ;code needing safe location during
           stx safe-1,y     ;reading of aux mem
           sta start-1,Y
           dey
           bne move
           jmp |safe        ;jump to safe ground
back       php              ;save status
           ldy #done-start  ;move zero page back
move2      lda start-1,y
           sta |safe-1,y
           dey
           bne move2
           pla
           bcs noaux
isaux      jmp muck128      ;there is 128K

*  You can put your own routine at "noaux" if you wish to
*  distinguish between 64K without an 80-column card and
*  64K with an 80-column card.

noaux      anop
nocard     lda #64          ;only 64K
           jmp exita
muck128    jmp exit128      ;there's 128K

*  This is the routine run in the safe area not affected
*  by bank-switching the main and aux RAM.

start      lda #test4       ;yet another test byte
           sta wrcardram    ;write to aux while on main zero page
           sta rdcardram    ;read aux ram as well
           sta begpage2     ;check for sparse memory mapping
           lda begsprse     ;if sparse, these will be the same
           cmp #test4       ;value since they're 1K apart
           bne auxmem       ;yup, there's 128K!
           asl begsprse     ;may have been lucky so we'll
           lda begpage2     ;change the value and see what happens
           cmp begsprse
           bne auxmem
           sec             ;oops, no auxiliary memory
           bcs goback
auxmem     clc
goback     sta wrmainram   ;write main RAM
           sta rdmainram   ;read main RAM
           jmp back        ;continue with program in main mem
done       nop             ;end of relocated program marker


*  The storage locations for the returned machine ID:

machine    ds  1           ;the type of Apple II
romlevel   ds  1           ;which revision of the machine
memory     ds  1           ;how much memory (up to 128K)
IIgsA      ds  2           ;16-bit field
IIgsX      ds  2           ;16-bit field
IIgsY      ds  2           ;16-bit field
save       ds  6           ;six bytes for saved data

IDTable    dc  I1'1,1'     ;Apple ][
           dc  H'B3 38 00'

           dc  I1'2,1'     ;Apple ][+
           dc  H'B3 EA 1E AD 00'

           dc  I1'3,1'     ;Apple /// (emulation)
           dc  H'B3 EA 1E 8A 00'

           dc  I1'4,1'     ;Apple IIe (original)
           dc  H'B3 06 C0 EA 00'

;  Note: You must check for the Apple IIe Card BEFORE you
;  check for the enhanced Apple IIe since the first
;  two identification bytes are the same.

           dc  I1'6,1'     ;Apple IIe Card for the Macintosh LC (1st
release)
           dc  H'B3 06 C0 E0 DD 02 BE 00 00'

           dc  I1'4,2'     ;Apple IIe (enhanced)
           dc  H'B3 06 C0 E0 00'

           dc  I1'5,1'     ;Apple IIc (original)
           dc  H'B3 06 C0 00 BF FF 00'

           dc  I1'5,2'     ;Apple IIc (3.5 ROM)
           dc  H'B3 06 C0 00 BF 00 00'

           dc  I1'5,3'     ;Apple IIc (Mem. Exp)
           dc  H'B3 06 C0 00 BF 03 00'

           dc  I1'5,4'     ;Apple IIc (Rev. Mem. Exp.)
           dc  H'B3 06 C0 00 BF 04 00'

           dc  I1'5,5'     ;Apple IIc Plus
           dc  H'B3 06 C0 00 BF 05 00'
           dc  I1'0,0'      ;end of table

           end
```

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

- Apple II File Type Note $19
- Apple II File Type Note $1A
- 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: Jim Luther — May 1991
Written by: Cameron Birse and Matt Deatherage — December 1986

This Technical Note describes the ROM identification bytes in the Apple II family.

Changes since November 1988: Added the identification bytes needed to identify the Apple IIe Card for Macintosh LC.

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	$FBDD	$FBBE	$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 IIe Option Card	$06		$E0	$02	$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 Apple IIe Card for Macintosh LC uses the same identification bytes ($FBB3 and $FBC0) as an enhanced Apple IIe. Location $FBDD allows you to tell the difference between the Apple IIe Card and an enhanced Apple IIe because $FBDD will always contain the value $02 on the Apple IIe Card. Location $FBBE is the version byte for the Apple IIe Card (just as $FBBF is the version byte for the Apple IIc family) and is $00 for the first release of the Apple IIe Card.

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

- o Apple II Miscellaneous Technical Note #2, Apple II Family Identification Routines 2.2

### 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      TXTPAGE2  EQU   $C055         ;TURN ON PAGE 2 (READ)
0300:        C000      CLR80COL  EQU   $C000         ;TURN OFF 80 STORE (WRITE)
0300:        C001      SET80COL  EQU   $C001         ;TURN ON 80 STORE (WRITE)
0300:        0028      BASL      EQU   $28           ;BASE ADDRESS OF SCREEN
LOCATION
0300:        0029      BASH      EQU   $29
0300:        057B      OURCH     EQU   $57B          ;80 COLUMNS HORIZ. POSITION
0300:        05FB      OURCV     equ   $5fb          ;80 col vertical pos
0300:        0006      char      equ   6             ;place to hand character back
to basic
0300:                  *
0300:
*************************************************************
0300:                  *   GETCHAR - This routine gets an ascii character from the
*
0300:                  *   80 column display memory of the Apple IIe. It assumes
*
0300:                  *   that main memory is switched in and that the base addrs
*
0300:                  *   of the line has already been calculated and resides
*
0300:                  *   in BASL and BASH. It is meant to be called from BASIC
*
0300:                  *   as follows:
*
0300:                  *                     CALL 768
*
0300:                  *                     A = PEEK (6)
*
0300:                  *                     A$ = CHR$(A)
*
0300:                  *   As you can see, the character is returned in location
*
0300:                  *   $6 in zero page. This routine is offered as an example.
*
0300:                  *   No guaranties are made regarding its fitness for any
*
0300:                  *   purpose.           By Cameron Birse 6/10/86
*
0300:
*************************************************************
0300:                  *
0300:        0300      getchr    equ   *             ;get the char at the current
cursor loc.
0300:A9 01                       lda   #$01          ;mask for horiz test
0302:2C 7B 05                    bit   OURCH         ;are we in main or aux mem?
0305:D0 17   031E                bne   main          ;if bit 0 of OURCH is set,
then main mem
0307:        0307      aux       equ   *
0307:AD 7B 05                    lda   OURCH         ;get horiz pos.
030A:18                          clc                 ;clear the carry for divide
030B:6A                          ror   a             ;divide by two
030C:A8                          tay                 ;put the result in y
030D:8D 01 C0                    sta   SET80COL      ;turn on 80 store
0310:AD 55 C0                    lda   TXTPAGE2      ;flip to aux text page
0313:B1 28                       lda   (basl),y      ;get the character
0315:85 06                       sta   char
0317:AE 54 C0                    ldx   TXTPAGE1      ;turn off aux text page
```

```
031A:8D 00 C0       43           sta   CLR80COL      ;turn off 80 store
031D:60             44           rts
031E:        031E   45 main      equ   *
031E:AD 7B 05       46           lda   OURCH         ;get horiz pos.
0321:18             47           clc                 ;clear the carry for divide
0322:6A             48           ror   a             ;divide by two
0323:A8             49           tay                 ;put the result in y
0324:B1 28          50           lda   (basl),y      ;get the character
0326:85 06          51           sta   char
0328:60             52           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

- o Apple IIGS Firmware Reference
- o Apple IIe Technical Reference Manual
- o Apple IIc Technical Reference Manual, Second Edition

### 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, Second Edition, 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

o Apple IIGS Firmware Reference
o Apple IIe Technical Reference Manual
o Apple IIc Technical Reference Manual, Second Edition

### 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

Revised by: Matt Deatherage May 1992
Written by: Matt Deatherage July 1989

This Technical Note discusses recommended guidelines to ensure future compatibility and maintain workable standards for telecommunication programs.

CHANGES SINCE JULY 1989: Rewritten to be more explicit in passages.

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. That means if your terminal program uses both 4800 and 9600 baud, it should use the GS/OS drivers for 4800 baud and another method for 9600 baud--you cause more problems than you solve by using non-recommended methods for all speeds.

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 6.0, the driver can't help you. It is necessary to go directly to the firmware or hardware and risk 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 do 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 true for Apple IIe computers with Workstation Cards, and is also 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. The main interrupt handler is already very tight code, and it runs in ROM 10% faster than any code you can replace it with in RAM. If you patch out the main interrupt vector and add more than about two instructions to the code path before returning to the ROM, AppleTalk will lose data. If you patch out the main interrupt vector, you make it impossible for Apple to add additional functionality by patching the same vector without breaking AppleTalk--and the vector is there for system software's convenience, not yours.

  I can't make it any plainer than this--do not patch out the main interrupt vector unless it absolutely, positively cannot be avoided. The only cases we know about where it absolutely can't be avoided are very high-speed communications from slot-based cards; high-speed serial communications from the serial ports can be handled by patching the serial interrupt vector (see Apple IIgs Technical Note #18, Do-It-Yourself SCC Interrupts). If you have to patch the main interrupt vector to run at 38400 bps, unpatch it when you switch to 2400 baud. Only patch the vector while it's absolutely necessary, and don't leave it continually patched just because it's easier. You're breaking things when you do that, whether your testing reveals it to you or not.

  If you must patch out the main interrupt vector, make it very clear to your users, both in the documentation and on-screen, that other system services like AppleTalk will not function and may not return until the computer is restarted. Give them a chance to back out.

- 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.

- 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

- Apple IIgs Firmware Reference
- Apple IIgs Hardware Reference
- Apple IIgs Technical Note #18, Do-It-Yourself SCC Interrupts
- Apple II File Type Notes, File Type $E0, Auxiliary Type $8000
- Apple II File Type Notes, File Type $E0, Auxiliary Type $8002

### END OF FILE TN.MISC.014

###
### FILE: TN.MISC.015
###

Apple II
Technical Notes

Developer Technical Support

Apple II Miscellaneous
#15: Compatibility Across Apple II Models

Revised by: Dave Lyons May 1992
Written by: Dave Lyons January 1990

This Technical Note explains how you can get in big trouble with soft switches.

CHANGES SINCE JANUARY 1990: Added caution against calling the 80-column firmware with RAMRD or RAMWRT enabled.

CALL FIRMWARE WITH NORMAL MEMORY MAPPING

Firmware behaves unpredictably if you call it with nonstandard memory mapping in effect.

For example, do not call the 80-column firmware with RAMRD (RDCARDRAM) or RAMWRT (WRCARDRAM) turned on. If you do, the firmware accidentally accesses auxiliary-memory screen holes instead of main-memory screen holes, including MSLOT ($07F8 in main memory). This can cause the system to crash.

READ DEFINED SOFT SWITCHES ONLY

When a soft switch location is defined on one Apple II model but not others, it is not safe to read the soft switch and later decide whether to use the value that was read. The following two examples demonstrate the hazards of this method.

An application must read KEYMODREG ($C025) only after determining that it is running on an Apple IIgs (using IDROUTINE at $FE1F). Reading KEYMODREG and later ignoring the result if not on an Apple IIgs does not work.

NEWVIDEO ($C029) is also defined only on the Apple IIgs. Again, an application must know that it is running on an Apple IIgs before reading or writing this location. (If your application uses double-high resolution, check for an Apple IIgs before attempting to set the monochrome-double-hires bit in NEWVIDEO.)

Both of these locations are reserved on the Apple IIc Plus, and reading from or writing to them currently causes the Apple IIc Plus ROM to be swapped out and replaced by additional ROM, instantly killing your application.

Further Reference

- Apple IIgs Hardware Reference
- Apple IIe Technical Reference Manual
- Apple IIc Technical Reference Manual, Second Edition
- Miscellaneous Technical Note #7, Apple II Family Identification

### END OF FILE TN.MISC.015

```
#####################################################################
### FILE: TN.MISC.016
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple II Miscellaneous
#16: Apple II Parallel Interface Card Firmware

Written by: Jim Luther July 1990

This Technical Note documents the commands the Apple II Parallel Interface Card's firmware supports and how to find the slot occupied by an Apple II Parallel Interface Card.

Parallel Printer Interface Card Commands

The manual that shipped with the Apple II Parallel Interface Card states correctly that its firmware can be "identical to the firmware in the earlier Apple II Centronics(R) Printer Card" or that it can be "identical to the firmware in the earlier Apple II Parallel Printer Card." However, the manual did not correctly document the commands the Parallel Interface Cards can handle or explain the commands clearly.

Apple II Parallel Interface Card commands, embedded in the output flow to the card's firmware, are invoked by the BASIC output routines. The following three options affect the output data flow and can be controlled by sending control codes as commands to the firmware:

Flow option	Description
Video echo	When this option is on, characters sent to the printer are echoed to the video screen (via COUT1) after they are sent out the parallel port. Note: When an Apple II Parallel Interface Card in Parallel Printer mode is echoing characters to the video screen, the line length is forced by the card's firmware to 40.
Line length	When the video echo option is off, the line length (the number of character sent before a forced carriage-return) can be set in the range of 40 to 255 characters. When video echo is on, the line length is forced to 40.
Automatic line-feed	When this option is on, the firmware generates and sends a line-feed character after each carriage-return character sent. Note: The automatic line-feed option can only be used when the Apple II Parallel Interface Card is in Parallel Printer mode.

All commands are preceded by a command character. The normal command character is Control-I (ASCII $09). If you want to change the command character from Control-I to another command character (for example, Control-W), send Control-I, Control-W. To change back, send Control-W, Control-I. The format of the commands is as follows:

    {command-character} {command-string}

There are two types of commands:

- o Commands that only change the video echo mode or the automatic line-feed mode. The command format for these commands consist of an uppercase letter by itself (for example, I to restore the default settings).
- o Commands that set the line length and change the video echo mode or automatic line-feed mode. The command format for these commands consists of a number, represented by n, followed by an uppercase letter with no space between the characters. The number, in the range of 40 to 255 characters is the new line length setting (for example, 80N to set the line length to 80).

Commands in Centronics Printer Mode (switch 6 on)

In Centronics Printer mode, the default settings are:

- o Video echo option on
- o 40-column line length
- o Command character = Control-I

The Centronics mode firmware does not support the option to automatically generate line-feed characters after carriage return characters. When the interface card is initialized in Centronics mode, it also sends the Centronics MicroPrinter 40-column mode control character to the printer (ASCII $9E).

The following are the commands supported by the Apple II Parallel Interface Card when it is in Centronics Printer mode:

nN

The N command always does these two things:

- o Turn the video echo option off.
- o Send the Centronics MicroPrinter 80-column mode control character to the printer (ASCII $1D).

In addition, if the number n is included, the line length is set to n. The number n must be in the range of 40 to 255 characters.

nO

The O command always does the following:

- o Turn the video echo option on.

In addition, if the number n is included, the line length is set to n. The number n must be in the range of 40 to 255 characters.

Commands in Parallel Printer Mode (switch 6 off)

In Parallel Printer mode, the default settings are:

- o Video echo option on
- o 40-column line length
- o Automatic line-feed option on
- o Command character = Control-I

The following are the commands supported by the Apple II Parallel Interface Card when it is in Parallel Printer mode:

I or M

The I and the M commands always do these three things:

- o Turn the video echo option on
- o Force the line length to 40
- o Turn the automatic line-feed option on

K or O

The K and the O commands always do these three things:

- o Turn the video echo option on
- o Force the line length to 40
- o Toggle the automatic line-feed option

nH or nL

The H and the L commands always do these two things:

- o Turn the video echo option off
- o Turn the automatic line-feed option off

In addition, if the number n is included, the line length is set to n. The number n must be in the range of 40 to 255 characters.

nJ or nN

The J and the N commands always do the following:

- o Turn the video echo option off

In addition, if the number n is included, the line length is set to n. The number n must be in the range of 40 to 255 characters. These commands do not affect the automatic line-feed option setting.

Finding the Parallel Printer Interface Card

The Apple II Parallel Interface Card manual does a good job of describing the Apple Pascal 1.1 interface standard. However, publishing that information in that manual is very misleading since the Apple II Parallel Interface Card does not support any part of the Pascal 1.1 interface standard.

However, since most programs use the Pascal 1.1 device signature bytes to identify peripheral cards, here are the values you find in the Pascal 1.1 device signature byte addresses:

Address	Value
$Cs05	$48
$Cs07	$48

Remember, these values do not correspond to any signature bytes defined by the Pascal 1.1 interface standard. The address $Cs0B contains the value $58 and the address $Cs0C contains the value $FF, (the last two bytes of a JSR $FF58 to identify the slot) but these two locations should not be used to identify parallel cards in general.

Further Reference

- o Apple II Parallel Interface Card Manual

Centronics is a registered trademark of Centronics Data Computer Corporation.

### END OF FILE TN.MISC.016

```
#####################################################################
### FILE: TN.MISC.017
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Apple II Miscellaneous #17: Buried Treasures of the Video Overlay Card

Written by: Dan Hitchens September 1990

This Technical Note describes some of the more esoteric features of the Video Overlay Card.

What All Does This Thing Do?

The Video Overlay Card contains a duplicate of the Apple IIgs video circuitry to help accomplish its task. This makes some pretty unusual (or esoteric, or downright weird) display behavior possible. Although this Note describes these techniques, the APDA document "Apple II Video Overlay Card Developers Notes" is considered the Bible of the VOC and should be consulted for further details, such as how to make the function calls described in this Note and what the constant values are.

RAM Page Select

Normally, the Mega II video circuitry on the Apple IIgs looks at bank $E1 (the auxiliary bank on Apple IIe) from $2000 to $9FFF for the Super High-Res graphics display buffer (pixel data, scan-line control bytes, and color palettes). This is the default for the VOC, and as such, a monitor connected to the VOC displays the same Super High-Res graphics as a monitor connected to the Apple IIgs main logic board.

A feature the VOC card has beyond the Apple IIgs video circuitry is its ability to select between bank $E1 (the auxiliary bank on the IIe) and bank $E0 (main bank on IIe) for its graphics display buffer. By specifying bank $E0 (or main bank on the IIe) , $E0/2000 to $E0/9FFF becomes the display buffer from which the pixel data, scan-line control bytes, and color palettes are found.

Another added feature of the VOC related to RAM Page Select is its ability to interlace between banks $E1 and $E0 (described later in this Note).

Note: When you select bank $E0 as the graphics display bank, you need to ensure that the display buffer memory is linear (remember, the Super High-Res graphics display buffer needs to be linear).

Technique

PASCAL:
```
      VDGGControl(VDRAMPageSel,VDAux); {selects Aux bank $E1}
      VDGGControl(VDRAMPageSel,VDMain); {selects Main bank $E0}
      VDGGControl(VDMainPageLin,VDEnable); {enable main page $E0 linearization}
      VDGGControl(VDMainPageLin,VDDisable); {disable main page linearization}
```

Uses

- o Dual monitors. A monitor connected to the main logic board of the IIgs has its display buffer in the normal $E1 bank, while a monitor connected to the VOC has its display buffer in bank $E0.

- o Double-buffered graphics. This is the ability to draw graphics in one bank while displaying from the other, then switching display banks when appropriate.

Interlacing

As mentioned earlier, another added feature of the VOC is its ability to perform 400 line interlacing. By enabling interlacing on the VOC, the card displays the even and odd lines from different memory areas depending on the setting of the RAM page select. If interlacing is enabled and the auxiliary bank is selected (the default at reset), then the VOC displays both the even and odd scan lines from bank $E1, giving the same display as if interlacing was not enabled at all (the 200 lines from bank $E1 are interlaced with the same 200 lines from bank $E1). If interlacing is enabled and the main bank is selected, then the VOC displays both the even and odd scan lines from bank $E0 (same sort of thing, the 200 lines from bank $E0 are interlaced with the same 200 lines from bank $E0.)

Now for the useful part,--if interlacing is enabled and RAM Page Select is set to interlace, then interlacing occurs between banks $E1 and $E0 (the odd 200 lines come from bank $E1 and the even 200 lines come from bank $E0).

Remember in Super High-Res interlace mode, not only the pixel data but also the scan line control bytes and color palettes can be different between the two banks and need to be loaded. Also be sure to turn main page linearization on before loading data into the $E0 bank (see discussion on RAM Page Select).

Technique

PASCAL:

```
    VDGGControl(VDRAMPageSel,VDInterlace); {selects interlacing between banks
                                            $E1 and $E0}
    VDGGControl(VDInterlaceMode,VDEnable); {enables 400 line interlace mode}
    VDGGControl(VDInterlaceMOde,VDDisable); {disables 400 lines interlace mode}
```

Uses

- o 400-line display. A monitor connected to the VOC can display 400 different lines when interlace is enabled and RAM Page Select is set to interlace.

  Warning: The technique of interlacing can cause noticeable flicker when pixels on adjacent horizontal lines are different.

Dual Monitors

Because the VOC has a duplicate of the video circuitry of the Apple IIgs, it has the capability to have two monitors displaying the same or totally different information at the same time. The VOC doesn't have to be in the same display mode as the main logic board. It can be in the same video mode or in a totally different one (i.e., the main logic board is in text mode while the VOC is in Super High-Res mode.)

The technique required for putting the VOC into a video mode different from that of the main logic board's circuitry is to enable and disable, at the appropriate time, the bus to the VOC. To go into the video mode in which you want the VOC to remain, you must first make sure the bus is enabled (power on default). Once you have the VOC in the mode you want, disable the bus with VDGGControl(VDGGBus, VDEnable), then go into the video mode in which you want the main logic board. After you have the main logic board into the mode you want, reenable the bus with VDGGControl(VDGGBus, VDDisable). Remember to reenable the bus or the VOC screen remains frozen--memory writes to the video display buffers are not written through to the VOC's internal memory display buffers.

Technique

```
PASCAL:
   VDGGControl(VDGGBus,VDEnable);  {Enables the GGBus_Disable circuitry which
                                    inhibits writes to the VOC display memory.}
   VDGGControl(VDGGBus,VDDisable); {Disables the GGBus_Disable circuitry which
                                    enables writes to the VOC display memory
                                    (power on default)}
```

Uses

- o Two monitors with different display modes. A monitor connected to the VOC can display one mode while a monitor connected to the main logic board is displaying another. There are many different applications that could benefit from this type of monitor arrangement. For example, you might need one monitor to display control information, while another is displaying graphics (a debugger could trace code in text mode on one monitor, while the other monitor is displaying the actual graphics in real time.)

Apple IIe

One of the outstanding features of the VOC is its ability to provide all the video modes found on the IIgs to the Apple IIe community, including Super High-Res graphics (320x200 or 640x200 bit resolution). Not only can Apple IIe owners experience the world of Super High-Res graphics with the VOC, they also can perform all the previously mentioned functions (such as RAM page select, interlacing, and dual monitors).

Super High-Res graphics mode on the IIe is enabled the same way as on the IIgs. The New Video register, which is located at $C029, is the register which controls the mode. The following is a short description of the bits for this register as found in the Apple IIgs Hardware Reference:

```
    Bit 7:    0    Enables old Apple IIe video modes
              1    Enables Super High-Res video mode
    Bit 6:    0    Memory linearization is disabled (old Apple IIe memory
                   map configuration)
              1    Memory linearization is enabled (memory from $2000 to
                   $9D00 becomes one linear address space.)
    Bit 5:    0    Double Hi-Res graphics are displayed in color
              1    Double Hi-Res graphics are displayed in black and white
    Bits 4-0:      Undefined
```

From this description, all that is needed to switch into Super High-Res display mode, is to set bits six and seven at memory location $C029. When you want to go back to the old Apple IIe video modes, just clear bits six and seven.

Note: The Super High-Res graphics display buffer on the Apple IIe is located in Auxiliary memory from $2000 to $9FFF (unless its been changed to Main page with RAM page select, as described earlier in this Note). Within this display buffer, you must set up the following three types of data:

$2000-$9CFF	Pixel Data
$9D00-$9DFF	Scan-line control bytes
$9E00-$9FFF	Color palettes

For more details, see the Apple IIgs Hardware Reference.

Implementation Technique

The thing to remember is to make sure memory linearization is on ($C029 bit 6=1) before you fill the Super High-Res buffer ($2000-$9FFF). If you write a program that moves data into the display buffer with memory linearization off ($C029 bit 6=0), then enable Super High-Res by turning bits six and seven on, you do not get the results you anticipate. What you see is a scrambled mess on the screen, because you filled the display buffer with memory linearization off (the VOC filled its display buffer in a nonlinear manner), and when you turned Super High-Res mode on, the VOC expected to see memory in a linear manner.

Apple IIe Super High-Res Demo Program

```
****************************************************************
*
*   A simple Apple IIe super high res. screen demo load program for
*   use with the "Apple II Video Overlay Card"
*
*    Copyright Apple Computer, Inc. 1989
*    All rights reserved.
*
*    Programmed by: Dan Hitchens May 1989
*
****************************************************************
*
*    This program loads super high res. screen dumped files which
*   were named "SCREEN.x" (where x starts at zero and increments
*   up sequentially.)  When this program doesn't find the next
*   sequential file, it beeps and starts looking back at
*   "SCREEN.0" again (this program loops endlessly.)
*
        longi off
        longa off
        MACHINE M65c02
        ENTRY begin:CODE

****************************************************************

begin        PROC
BEll1        EQU $FBDD            ;beep subroutine
MLI          EQU $BF00            ;ProDos-8 machine language interface
AuxMove      EQU $C311            ;Aux. move firmware routine
OpenCMD      EQU $C8              ;Open command
ReadCMD      EQU $CA              ;Read command
CloseCMD     EQU $CC              ;Close command

             lda #$40
             sta $C029            ;turn on memory linearization

NextFile     nop
;First init. a few counters

             lda #$20
             sta AuxAddr          ;init aux. address counter

;Now Open the file
             jsr MLI              ;perform an open command
             dc.b OpenCmd         ;passed command (open)
             dc.w OpenBlk         ;passed block address
             beq OKOpen2          ;branch if able to open file

;We arrive here if unable to open the file (probably wasn't there)
InitZero     jsr Bell1            ;sound the bell
             lda #'0'             ;re-init to "SCREEN.0"
             sta FileNum-1        ;init. SCREEN.x to .0
             jmp NextFile         ;go try opening "SCREEN.0"

OKOpen2      lda ORefNum          ;get ref. num from open command
             sta RRefNum          ;store for read command
             sta CRefNum          ;store for close command

OKOpen       nop
;Now read $4000 bytes into $3000
             jsr MLI              ;call machine language interface
             dc.b ReadCMD         ;passed command (read)
             dc.w ReadBlk         ;passed block address
             sta MyError          ;save returned error (if any)
             bcs AllDone          ;branch if error occurred

;Set starting address to $3000
             lda #0
             sta $3c
             lda #$30
             sta $3d
;Set ending address to $6fff
             lda #$ff
             sta $3e
             lda #$6f
             sta $3f
;Set destination starting addr to AuxAddr (its incremented)
             lda #0
             sta $42
             lda AuxAddr
             sta $43

             sec                     ;to indicate move from main to aux mem.
             jsr AuxMove             ;now move the data to aux. memory

;Now increment AuxAddr:=AuxAddr+$4000
             lda AuxAddr
             clc
             adc #$40
             sta AuxAddr
             lda MyError             ;get error (if any)
             beq OKOpen              ;continue reading until error (eof)

;Now we can turn on the super high res screen
AllDone      nop
             lda #$c0
             sta $c029


;Now delay for awhile (just a big delay loop)
             lda #0
             tax
             tay
             lda #12
xloop        dex
             nop
             nop
             nop
             nop
             nop
             nop
             nop
             nop
             nop

             bne xloop
yloop        dey
             bne xloop
             dec a
             bne xloop

;Now Close the file
             jsr MLI                 ;do a Close command
             dc.b CloseCmd           ;passed command (close)
             dc.w CloseBlk           ;passed block address

;Now increment to next file
             lda FileNum-1           ;get current file number
             cmp #'9'                ;is it nine
             beq InitZero            ;branch if already at nine
             clc
             adc #1                  ;else add one to next "SCREEN.x"
             sta FileNum-1
             jmp NextFile            ;go try and read next file

;------------------------------------------------------------------
```

```
        STRING PASCAL               ;we want pascal strings
OpenBlk      dc.b 3
             dc.W Pathname
             dc.W $2700
ORefNum      ds.b 1                 ;returned ref_num
Pathname     dc.b 'SCREEN.0'
FileNum      ds.b 1
ReadBlk      dc.b 4                 ;no. of parameters
RRefNum      ds.b 1                 ;ref. number (stuffed from Open command)
             dc.w $3000             ;data buffer pointer
             dc.w $4000             ;request count
             ds.w 1                 ;actual data transferred count

CloseBlk     dc.b 1                 ;no. of parameters
CRefNum      ds.b 1                 ;ref. number (Stuffed from Open command)

AuxAddr      ds.b 1                 ;aux. address pointer

MyError      ds.b 1
;--------------------------------------------------------------------
            ENDP
        END
```

Further Reference

- o Apple IIgs Hardware Reference
- o Apple II Video Overlay Card Developers Notes (APDA)

### END OF FILE TN.MISC.017

```
########################################################################
### 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, Second Edition

### 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 1990
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.
Changes since November 1988: Added the mouse ID bytes since they are no longer included in other documentation.

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 behave as an Apple mouse.

In addition to the standard Pascal 1.1 Firmware Protocol ID bytes, the AppleMouse II card is identified by a value of $20 at $Cn0C ("X-Y Pointing device, type zero") and a value of $D6 at $CnFB, where n is the slot number. The $CnFB value is not part of the Pascal 1.1 Firmware Protocol.

### 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, Second Edition, 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, Second Edition

### 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 :

```
o    The 48K Run-Time System  V1.2    (standard and stripped)
o    The 64K Run-Time System  V1.3    (standard only)
o    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:

- System HOMECURSOR, CLEARSCREEN, and CLEARLINE functions
- System prompt function
- Compiler, Assembler, Linker, Editor, and Filer
- 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:

o   BUFFER[OFFSET] points to the character following the identifier just found.
o   ID contains the first eight alphanumeric characters of the identifier just found, left justified and padded with spaces as necessary.
o   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.
o   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.PASC.017
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

Pascal #17: SYSTEM.APPLE Patch V2.0

Revised by: Dan Strnad — March 1990
Written by: Dan Strnad — May 1988

This Technical Note documents the Apple Pascal SYSTEM.APPLE patch version 2.0 (available as part of the Apple Pascal 1.3 package from APDA), which fixes two Pascal system bugs found in the 64K and 128K, regular and run-time, version 1.2 and 1.3 Pascal systems. A companion program confirms the correct installation of these patches.

SYSTEM.APPLE patch 2.0 corrects the following two bugs:

1. UNITSTATUS calls to device numbers six, seven, and eight malfunction in Apple II Pascal 1.3.
2. On Apple IIGS machines, disk access to 5.25" drives may fail under Apple II Pascal.

The following limitations exist with these patches:

1. Keyboard input from an external terminal does not work; however, SYSTEM.APPLE patch 1.0 is also available to support external terminals. SYSTEM.APPLE patch 1.0 fixes only the UNITSTATUS problem; it does not address the 5.25" drive problem on the Apple IIGS.

2. Type-ahead buffering may drop keystrokes under the following circumstances:
   a. On an Apple IIGS if the user has enabled keyboard buffering from the Control Panel.
   b. On an Apple IIc series computer, if an Apple II Pascal program directly accesses the firmware-level keyboard buffering.

Apple II Pascal 1.3 (and 1.2) still provide type-ahead keystroke buffering on all Apple IIs.

To use the patch program, X)ecute it from Pascal with the name PATCH. To confirm the proper installation of the patch, perform an X)ecute with the name CONFIRM.

### END OF FILE TN.PASC.017

```
#####################################################################
### 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
o 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
- ProDOS 8 Technical Reference Manual
- 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

- 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

- 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
o   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

- ProDOS 8 Technical Reference Manual
- 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

o ProDOS 8 Technical Reference Manual
o 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

o 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
o   ProDOS 8 Technical Reference Manual
o   ProDOS 8 Update Manual
o   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: Dave Lyons, Keith Rollin, & Matt Deatherage   November 1989
Written by: Greg Seitz   December 1983

This Technical Note presents an assembly language example of a recursive directory reading routine which is AppleShare compatible.
Changes since November 1988: The routine now ignores the file_count field in a directory, and it properly increments ThisBlock. More discussion of AppleShare volumes is included.

This Note presents a routine in assembly language for recursively cataloging a ProDOS directory. 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), displays the contents of each subdirectory as it is encountered.

READ_BLOCK is not used, since it does not work with AppleShare servers. READ is used instead, since it works for AppleShare volumes as well as local disks. 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 our place, 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.

This routine is generally compatible with AppleShare volumes, but it is impossible to guarantee a complete traversal of all the accessible files on an AppleShare volume: another user on the same volume can add or remove files or directories at any time. If entries are added or removed, some filenames may be displayed twice or missed completely. Be sure that your programs deal with this sort of situation adequately.

We assume that AppleShare is in short naming mode (as it is by default under ProDOS 8). If you enable long naming mode, then illegal characters in filenames will not be translated into question marks. In this case, the code would need to be modified to deal with non-ASCII characters. Also, the ChopName routine would need to be aware that a slash (/) character could be contained inside the name of a directory that had been added to the pathname. (As the code stands, such directories fail to open, but their names are still temporarily added to the pathname.)

When the catalog routine encounters an error, it displays a brief message and continues. It is important not to abort on an error, since AppleShare volumes generally contain files and folders with names that are inaccessible to ProDOS, as well as folders that are inaccessible to your program's user (error $4E, access error).

The code example 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.

```
----- NEXT OBJECT FILE NAME IS CATALOG.0
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:                   *     Dave Lyons 11/89
0800:                   *
0800:                   * This program shows the latest "Apple Approved"
0800:                   * method for reading a directory under ProDOS 8.
0800:                   * READ_BLOCK is not used, since it is incompatible
0800:                   * with AppleShare file servers.
0800:                   *
0800:                   * November 1989: The file_count field is no longer
0800:                   * used (all references to ThisEntry were removed).
0800:                   * This is because the file count can change on the fly
0800:                   * on AppleShare volumes.  (Note that the old code was
0800:                   * accidentally decrementing the file count when it
0800:                   * found an entry for a deleted file, so some files
0800:                   * could be left off the end of the list.)
0800:                   *
0800:                   * Also, ThisBlock now gets incremented when a chunk
0800:                   * of data is read from a directory.  Previously, this
0800:                   * routine could get stuck in an endless loop when
0800:                   * a subdirectory was found outside the first block of
0800:                   * its parent directory.
0800:                   *
0800:                   * Limitations:  This routine cannot reach any
0800:                   * subdirectory whose pathname is longer than 64
0800:                   * characters, and it will not operate correctly if
0800:                   * any subdirectory is more than 255 blocks long
0800:                   * (because ThisBlock is only one byte).
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:                   * ProDOS command numbers
0800:                   *
```

```
0800:        BF00   MLI       equ   $BF00         ; MLI entry point
0800:        00C7   mliGetPfx equ   $C7           ; GET_PREFIX
0800:        00C8   mliOpen   equ   $C8           ; Open a file command
0800:        00CA   mliRead   equ   $CA           ; Read a file command
0800:        00CC   mliClose  equ   $CC           ; Close a file command
0800:        00CE   mliSetMark equ  $CE           ; SET_MARK command
0800:        004C   EndOfFile equ   $4C           ; EndOfFile error
0800:               *
0800:               * BASIC.SYSTEM stuff
0800:               *
0800:        BEF5   GetBufr   equ   $BEF5         ; BASIC.SYSTEM get buffer
routine

01 CATALOG         ProDOS Catalog Routine             14-OCT-89  16:20 PAGE 3

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:               *
0800:               * Monitor routines
0800:               *
0800:        FDED   cout      equ   $FDED         ; output a character
0800:        FD8E   crout     equ   $FD8E         ; output a RETURN
0800:        FDDA   prbyte    equ   $FDDA         ; print byte in hex
0800:        00A0   space     equ   $A0           ; a space character
0800:               *
0800:               *******************************************************
0800:               *
0800:        0800   Start     equ   *
0800:               *
0800:               * Simple routine to test the recursive ReadDir
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 D7 09                 sta   ioBuf+1
080A:               *
080A:               * Use the current prefix for the name of the
080A:               * directory to display.  Note that the string we
080A:               * pass to ReadDir has to end with a "/", and that
080A:               * the result of GET_PREFIX does.
080A:               *
080A:20 00 BF                 jsr   MLI
080D:C7                       db    mliGetPfx
080E:E8 09                    dw    GetPParms
0810:B0 0C   081E             bcs   exit
0812:               *
0812:A9 00                    lda   #0
0814:8D CE 09                 sta   Depth
0817:               *
0817:A9 EB                    lda   #nameBuffer
```

```
0819:A2 0B          102           ldx   #<nameBuffer
081B:20 1F 08       103           jsr   ReadDir
081E:               104 *
081E:         081E  105 exit      equ   *
081E:60             106           rts
081F:               107 *
081F:               108 ******************************************************
081F:               109 ******************************************************
081F:               110 *
081F:         081F  111 ReadDir   equ   *
081F:               112 *
081F:               113 *  This is the actual recursive routine. It takes as
081F:               114 *  input a pointer to the directory name to read in
081F:               115 *  A,X (lo,hi), opens it, and starts to read the
081F:               116 *  entries. When it encounters a filename, it calls

01 CATALOG          ProDOS Catalog Routine              14-OCT-89  16:20 PAGE 4

081F:               117 *  the routine "VisitFile". When it encounters a
081F:               118 *  directory name, it calls "VisitDir".
081F:               119 *
081F:               120 *  The directory pathname string must end with a "/"
081F:               121 *  character.
081F:               122 *
081F:               123 ******************************************************
081F:               124 *
081F:85 80          125           sta   dirName       ; save a pointer to name
0821:86 81          126           stx   dirName+1
0823:               127 *
0823:8D D4 09       128           sta   openName      ; set up OpenFile params
0826:8E D5 09       129           stx   openName+1
0829:               130 *
0829:         0829  131 ReadDir1  equ   *             ; recursive entry point
0829:20 79 08       132           jsr   OpenDir       ; open the directory as a
file
082C:B0 1F    084D  133           bcs   done
082E:               134 *
082E:4C 48 08       135           jmp   nextEntry     ; jump to the end of the loop
0831:               136 *
0831:         0831  137 loop      equ   *
0831:A0 00          138           ldy   #oType        ; get type of current entry
0833:B1 82          139           lda   (entPtr),y
0835:29 F0          140           and   #$F0          ; look at 4 high bits
0837:C9 00          141           cmp   #0            ; inactive entry?
0839:F0 0D    0848  142           beq   nextEntry     ; yes - bump to next one
083B:C9 D0          143           cmp   #$D0          ; is it a directory?
083D:F0 06    0845  144           beq   ItsADir       ; yes, so call VisitDir
083F:20 B3 08       145           jsr   VisitFile     ; no, it's a file
0842:4C 48 08       146           jmp   nextEntry
0845:               147 *
0845:20 BA 08       148 ItsADir   jsr   VisitDir
0848:         0848  149 nextEntry equ   *
0848:20 77 09       150           jsr   GetNext       ; get pointer to next entry
084B:90 E4    0831  151           bcc   loop          ; Carry set means we're done
084D:         084D  152 done      equ   *             ; moved before PHA (11/89
DAL)
084D:48             153           pha                 ; save error code
084E:               154 *
```

```
084E:20 00 BF      155          jsr   MLI           ; close the directory
0851:CC            156          db    mliClose
0852:E1 09         157          dw    CloseParms
0854:              158 *
0854:68            159          pla                 ;we're expecting EndOfFile
error
0855:C9 4C         160          cmp   #EndOfFile
0857:F0 1F   0878  161          beq   hitDirEnd
0859:              162 *
0859:              163 * We got an error other than EndOfFile--report the
0859:              164 * error clumsily ("ERR=$xx").
0859:              165 *
0859:48            166          pha
085A:A9 C5         167          lda   #'E'|$80
085C:20 ED FD      168          jsr   cout
085F:A9 D2         169          lda   #'R'|$80
0861:20 ED FD      170          jsr   cout
0864:20 ED FD      171          jsr   cout
0867:A9 BD         172          lda   #'='|$80
0869:20 ED FD      173          jsr   cout
086C:A9 A4         174          lda   #'$'|$80

01 CATALOG         ProDOS Catalog Routine             14-OCT-89  16:20 PAGE 5

086E:20 ED FD      175          jsr   cout
0871:68            176          pla
0872:20 DA FD      177          jsr   prbyte
0875:20 8E FD      178          jsr   crout
0878:              179 *
0878:        0878  180 hitDirEnd equ   *
0878:60            181          rts
0879:              182 *
0879:              183 *****************************************************
0879:              184 *
0879:        0879  185 OpenDir   equ   *
0879:              186 *
0879:              187 *  Opens the directory pointed to by OpenParms
0879:              188 *  parameter block. This pointer should be init-
0879:              189 *  ialized BEFORE this routine is called. If the
0879:              190 *  file is successfully opened, the following
0879:              191 *  variables are set:
0879:              192 *
0879:              193 *     xRefNum     ; all the refnums
0879:              194 *     entryLen    ; size of directory entries
0879:              195 *     entPtr     ; pointer to current entry
0879:              196 *     ThisBEntry  ; entry number within this block
0879:              197 *     ThisBlock   ; offset (in blocks) into dir.
0879:              198 *
0879:20 00 BF      199          jsr   MLI           ; open dir as a file
087C:C8            200          db    mliOpen
087D:D3 09         201          dw    OpenParms
087F:B0 31   08B2  202          bcs   OpenDone
0881:              203 *
0881:AD D8 09      204          lda   oRefNum       ; copy the refnum return-
0884:8D DA 09      205          sta   rRefNum       ; ed by Open into the
0887:8D E2 09      206          sta   cRefNum       ; other param blocks.
088A:8D E4 09      207          sta   sRefNum
088D:              208 *
```

```
088D:20 00 BF      209          jsr   MLI           ; read the first block
0890:CA            210          db    mliRead
0891:D9 09         211          dw    ReadParms
0893:B0 1D   08B2  212          bcs   OpenDone
0895:              213 *
0895:AD 0E 0A      214          lda   buffer+oEntLen ; init 'entryLen'
0898:8D D1 09      215          sta   entryLen
089B:              216 *
089B:A9 EF         217          lda   #buffer+4     ; init ptr to first entry
089D:85 82         218          sta   entPtr
089F:A9 09         219          lda   #<buffer+4
08A1:85 83         220          sta   entPtr+1
08A3:              221 *
08A3:AD 0F 0A      222          lda   buffer+oEntblk ; init these values based on
08A6:8D CF 09      223          sta   ThisBEntry    ; values in the dir header
08A9:8D D2 09      224          sta   entPerBlk
08AC:              225 *
08AC:A9 00         226          lda   #0            ; init block offset into dir.
08AE:8D D0 09      227          sta   ThisBlock
08B1:              228 *
08B1:18            229          clc                 ; say that open was OK
08B2:              230 *
08B2:        08B2  231 OpenDone  equ   *
08B2:60            232          rts

01 CATALOG         ProDOS Catalog Routine             14-OCT-89  16:20 PAGE 6

08B3:              233 *
08B3:              234 *******************************************************
08B3:              235 *
08B3:        08B3  236 VisitFile equ   *
08B3:              237 *
08B3:              238 * Do whatever is necessary when we encounter a
08B3:              239 * file entry in the directory. In this case, we
08B3:              240 * print the name of the file.
08B3:              241 *
08B3:20 AC 09      242          jsr   PrintEntry
08B6:20 8E FD      243          jsr   crout
08B9:60            244          rts
08BA:              245 *
08BA:              246 *******************************************************
08BA:              247 *
08BA:        08BA  248 VisitDir  equ   *
08BA:              249 *
08BA:              250 * Print the name of the subdirectory we are looking
08BA:              251 * at, appending a "/" to it (to indicate that it's
08BA:              252 * a directory), and then calling RecursDir to list
08BA:              253 * everything in that directory.
08BA:              254 *
08BA:20 AC 09      255          jsr   PrintEntry    ; print dir's name
08BD:A9 AF         256          lda   #'/'|$80      ; tack on / at end
08BF:20 ED FD      257          jsr   cout
08C2:20 8E FD      258          jsr   crout
08C5:              259 *
08C5:20 C9 08      260          jsr   RecursDir     ; enumerate all entries in
sub-dir.
08C8:              261 *
08C8:60            262          rts
```

```
08C9:              *
08C9:              *******************************************************
08C9:              *
08C9:        08C9  RecursDir equ   *
08C9:              *
08C9:              * This routine calls ReadDir recursively. It
08C9:              *
08C9:              * - increments the recursion depth counter,
08C9:              * - saves certain variables onto the stack
08C9:              * - closes the current directory
08C9:              * - creates the name of the new directory
08C9:              * - calls ReadDir (recursively)
08C9:              * - restores the variables from the stack
08C9:              * - restores directory name to original value
08C9:              * - re-opens the old directory
08C9:              * - moves to our last position within it
08C9:              * - decrements the recursion depth counter
08C9:              *
08C9:EE CE 09                inc   Depth         ; bump this for recursive
call
08CC:              *
08CC:              * Save everything we can think of (the women,
08CC:              * the children, the beer, etc.).
08CC:              *
08CC:A5 83                   lda   entPtr+1
08CE:48                      pha
08CF:A5 82                   lda   entPtr
08D1:48                      pha
08D2:AD CF 09                lda   ThisBEntry

01 CATALOG         ProDOS Catalog Routine              14-OCT-89  16:20 PAGE 7

08D5:48                      pha
08D6:AD D0 09                lda   ThisBlock
08D9:48                      pha
08DA:AD D1 09                lda   entryLen
08DD:48                      pha
08DE:AD D2 09                lda   entPerblk
08E1:48                      pha
08E2:              *
08E2:              * Close the current directory, as ReadDir will
08E2:              * open files of its own, and we don't want to
08E2:              * have a bunch of open files lying around.
08E2:              *
08E2:20 00 BF                jsr   MLI
08E5:CC                      db    mliClose
08E6:E1 09                   dw    CloseParms
08E8:              *
08E8:20 2F 09                jsr   ExtendName    ; make new dir name
08EB:              *
08EB:20 29 08                jsr   ReadDir1      ; enumerate the subdirectory
08EE:              *
08EE:20 65 09                jsr   ChopName      ; restore old directory name
08F1:              *
08F1:20 79 08                jsr   OpenDir       ; re-open it back up
08F4:90 01   08F7            bcc   reOpened
08F6:              *
08F6:              * Can't continue from this point--exit in
```

```
08F6:              317 * whatever way is appropriate for your
08F6:              318 * program.
08F6:              319 *
08F6:00            320          brk
08F7:              321 *
08F7:         08F7 322 reOpened  equ   *
08F7:              323 *
08F7:              324 * Restore everything that we saved before
08F7:              325 *
08F7:68            326          pla
08F8:8D D2 09      327          sta   entPerBlk
08FB:68            328          pla
08FC:8D D1 09      329          sta   entryLen
08FF:68            330          pla
0900:8D D0 09      331          sta   ThisBlock
0903:68            332          pla
0904:8D CF 09      333          sta   ThisBEntry
0907:68            334          pla
0908:85 82         335          sta   entPtr
090A:68            336          pla
090B:85 83         337          sta   entPtr+1
090D:              338 *
090D:A9 00         339          lda   #0
090F:8D E5 09      340          sta   Mark
0912:8D E7 09      341          sta   Mark+2
0915:AD D0 09      342          lda   ThisBlock     ; reset last position in dir
0918:0A            343          asl   a             ; = to block # times 512
0919:8D E6 09      344          sta   Mark+1
091C:2E E7 09      345          rol   Mark+2
091F:              346 *
091F:20 00 BF      347          jsr   MLI           ; reset the file marker
0922:CE            348          db    mliSetMark

01 CATALOG         ProDOS Catalog Routine             14-OCT-89  16:20 PAGE 8

0923:E3 09         349          dw    SetMParms
0925:              350 *
0925:20 00 BF      351          jsr   MLI           ; now read in the block we
0928:CA            352          db    mliRead       ; were on last.
0929:D9 09         353          dw    ReadParms
092B:              354 *
092B:CE CE 09      355          dec   Depth
092E:60            356          rts
092F:              357 *
092F:              358 *******************************************************
092F:              359 *
092F:         092F 360 ExtendName equ  *
092F:              361 *
092F:              362 * Append the name in the current directory entry
092F:              363 * to the name in the directory name buffer. This
092F:              364 * will allow us to descend another level into the
092F:              365 * disk hierarchy when we call ReadDir.
092F:              366 *
092F:A0 00         367          ldy   #0            ; get length of string to
copy
0931:B1 82         368          lda   (entPtr),y
0933:29 0F         369          and   #$0F
0935:8D 62 09      370          sta   extCnt        ; save the length here
```

```
0938:8C 63 09      371          sty   srcPtr        ; init src ptr to zero
093B:              372 *
093B:A0 00         373          ldy   #0            ; init dest ptr to end of
093D:B1 80         374          lda   (dirName),y   ; the current directory name
093F:8D 64 09      375          sta   destPtr
0942:              376 *
0942:        0942  377 extloop  equ   *
0942:EE 63 09      378          inc   srcPtr        ; bump to next char to read
0945:EE 64 09      379          inc   destPtr       ; bump to next empty location
0948:AC 63 09      380          ldy   srcPtr        ; get char of sub-dir name
094B:B1 82         381          lda   (entPtr),y
094D:AC 64 09      382          ldy   destPtr       ; tack on to end of cur. dir.
0950:91 80         383          sta   (dirName),y
0952:CE 62 09      384          dec   extCnt        ; done all chars?
0955:D0 EB   0942  385          bne   extloop       ; no - so do more
0957:              386 *
0957:C8            387          iny
0958:A9 2F         388          lda   #'/'          ; tack "/" on to the end
095A:91 80         389          sta   (dirName),y
095C:              390 *
095C:98            391          tya                 ; fix length of filename to
open
095D:A0 00         392          ldy   #0
095F:91 80         393          sta   (dirName),y
0961:              394 *
0961:60            395          rts
0962:              396 *
0962:        0001  397 extCnt   ds    1
0963:        0001  398 srcPtr   ds    1
0964:        0001  399 destPtr  ds    1
0965:              400 *
0965:              401 *
0965:              402 *****************************************************
0965:              403 *
0965:        0965  404 ChopName equ   *
0965:              405 *
0965:              406 * Scans the current directory name, and chops

01 CATALOG         ProDOS Catalog Routine             14-OCT-89  16:20 PAGE 9

0965:              407 * off characters until it gets to a /.
0965:              408 *
0965:A0 00         409          ldy   #0            ; get len of current dir.
0967:B1 80         410          lda   (dirName),y
0969:A8            411          tay
096A:        096A  412 ChopLoop equ   *
096A:88            413          dey                 ; bump to previous char
096B:B1 80         414          lda   (dirName),y
096D:C9 2F         415          cmp   #'/'
096F:D0 F9   096A  416          bne   ChopLoop
0971:98            417          tya
0972:A0 00         418          ldy   #0
0974:91 80         419          sta   (dirName),y
0976:60            420          rts
0977:              421 *
0977:              422 *****************************************************
0977:              423 *
0977:        0977  424 GetNext  equ   *
```

```
0977:              *
0977:              * This routine is responsible for making a pointer
0977:              * to the next entry in the directory. If there are
0977:              * still entries to be processed in this block, then
0977:              * we simply bump the pointer by the size of the
0977:              * directory entry. If we have finished with this
0977:              * block, then we read in the next block, point to
0977:              * the first entry, and increment our block counter.
0977:              *
0977:CE CF 09                 dec   ThisBEntry    ; dec count for this block
097A:F0 10   098C            beq   ReadNext      ; done w/this block, get next
one
097C:              *
097C:18                      clc                 ; else bump up index
097D:A5 82                   lda   entPtr
097F:6D D1 09                adc   entryLen
0982:85 82                   sta   entPtr
0984:A5 83                   lda   entPtr+1
0986:69 00                   adc   #0
0988:85 83                   sta   entPtr+1
098A:18                      clc                 ; say that the buffer's good
098B:60                      rts
098C:              *
098C:        098C  ReadNext  equ   *
098C:20 00 BF                jsr   MLI           ; get the next block
098F:CA                      db    mliRead
0990:D9 09                   dw    ReadParms
0992:B0 16   09AA            bcs   DirDone
0994:              *
0994:EE D0 09                inc   ThisBlock
0997:              *
0997:A9 EF                   lda   #buffer+4     ; set entry pointer to
beginning
0999:85 82                   sta   entPtr        ; of first entry in block
099B:A9 09                   lda   #<buffer+4
099D:85 83                   sta   entPtr+1
099F:              *
099F:AD D2 09                lda   entPerBlk     ; re-init 'entries in this
block'
09A2:8D CF 09                sta   ThisBEntry
09A5:CE CF 09                dec   ThisBEntry
09A8:18                      clc                 ; return 'No error'
09A9:60                      rts

01 CATALOG         ProDOS Catalog Routine             14-OCT-89  16:20 PAGE 10

09AA:              *
09AA:        09AA  DirDone   equ   *
09AA:38                      sec               ; return 'an error occurred'
(error in A)
09AB:60                      rts
09AC:              *
09AC:              *******************************************************
09AC:              *
09AC:        09AC  PrintEntry equ  *
09AC:              *
09AC:              * Using the pointer to the current entry, this
09AC:              * routine prints the entry name. It also pays
```

```
09AC:              * attention to the recursion depth, and indents
09AC:              * by 2 spaces for every level.
09AC:              *
09AC:AD CE 09                lda   Depth         ; indent two blanks for each
level
09AF:0A                      asl   a             ; of directory nesting
09B0:AA                      tax
09B1:F0 08   09BB            beq   spcDone
09B3:A9 A0         spcloop   lda   #space
09B5:20 ED FD                jsr   cout
09B8:CA                      dex
09B9:D0 F8   09B3            bne   spcloop
09BB:        09BB  spcDone   equ   *
09BB:              *
09BB:A0 00                   ldy   #0            ; get byte that has the
length byte
09BD:B1 82                   lda   (entPtr),y
09BF:29 0F                   and   #$0F          ; get just the length
09C1:AA                      tax
09C2:        09C2  PrntLoop  equ   *
09C2:C8                      iny                 ; bump to the next char.
09C3:B1 82                   lda   (entPtr),y    ; get next char
09C5:09 80                   ora   #$80          ; COUT likes high bit set
09C7:20 ED FD                jsr   cout          ; print it
09CA:CA                      dex                 ; printed all chars?
09CB:D0 F5   09C2            bne   PrntLoop      ; no - keep going
09CD:60                      rts
09CE:              *
09CE:              *******************************************************
09CE:              *
09CE:              * Some global variables
09CE:              *
09CE:        0001  Depth     ds    1             ; amount of recursion
09CF:        0001  ThisBEntry ds   1             ; entry in this block
09D0:        0001  ThisBlock ds    1             ; block with dir
09D1:        0001  entryLen  ds    1             ; length of each directory
entry
09D2:        0001  entPerBlk ds    1             ; entries per block
09D3:              *
09D3:              *******************************************************
09D3:              *
09D3:              * ProDOS command parameter blocks
09D3:              *
09D3:        09D3  OpenParms equ   *
09D3:03                      db    3             ; number of parms
09D4:        0002  OpenName  ds    2             ; pointer to filename
09D6:00 00         ioBuf     dw    $0000         ; I/O buffer
09D8:        0001  oRefNum   ds    1             ; returned refnum
09D9:              *
09D9:        09D9  ReadParms equ   *

01 CATALOG         ProDOS Catalog Routine           14-OCT-89  16:20 PAGE 11

09D9:04                      db    4             ; number of parms
09DA:        0001  rRefNum   ds    1             ; refnum from Open
09DB:EB 09                   dw    buffer        ; pointer to buffer
09DD:00 02         reqAmt    dw    512           ; amount to read
09DF:        0002  retAmt    ds    2             ; amount actually read
```

```
09E1:               528 *
09E1:         09E1  529 CloseParms equ   *
09E1:01             530            db    1              ; number of parms
09E2:         0001  531 cRefNum    ds    1              ; refnum from Open
09E3:               532 *
09E3:         09E3  533 SetMParms equ    *
09E3:02             534            db    2              ; number of parms
09E4:         0001  535 sRefNum    ds    1              ; refnum from Open
09E5:         0003  536 Mark       ds    3              ; file position
09E8:               537 *
09E8:         09E8  538 GetPParms equ    *
09E8:01             539            db    1              ; number of parms
09E9:EB 0B          540            dw    nameBuffer     ; pointer to buffer
09EB:               541 *
09EB:         0200  542 buffer     ds    512            ; enough for whole block
0BEB:               543 *
0BEB:         0040  544 nameBuffer ds    64             ; space for directory name

01 SYMBOL TABLE    SORTED BY SYMBOL                    14-OCT-89  16:20 PAGE 12

 09EB BUFFER         096A CHOPLOOP       0965 CHOPNAME       09E1 CLOSEPARMS
 FDED COUT           09E2 CREFNUM        FD8E CROUT          09CE DEPTH
 0964 DESTPTR        09AA DIRDONE          80 DIRNAME        084D DONE
   4C ENDOFFILE      09D2 ENTPERBLK        82 ENTPTR         09D1 ENTRYLEN
 081E EXIT           0962 EXTCNT         092F EXTENDNAME     0942 EXTLOOP
 BEF5 GETBUFR        0977 GETNEXT        09E8 GETPPARMS      0878 HITDIREND
 09D6 IOBUF          0845 ITSADIR        0831 LOOP           09E5 MARK
   CC MLICLOSE         C7 MLIGETPFX        C8 MLIOPEN        BF00 MLI
   CA MLIREAD          CE MLISETMARK     0BEB NAMEBUFFER     0848 NEXTENTRY
   24 OENTBLK          23 OENTLEN        0879 OPENDIR        08B2 OPENDONE
 09D4 OPENNAME       09D3 OPENPARMS      09D8 OREFNUM          00 OTYPE
 FDDA PRBYTE         09AC PRINTENTRY     09C2 PRNTLOOP       0829 READDIR1
 081F READDIR        098C READNEXT       09D9 READPARMS      08C9 RECURSDIR
 08F7 REOPENED      ?09DD REQAMT        ?09DF RETAMT         09DA RREFNUM
 09E3 SETMPARMS        A0 SPACE          09BB SPCDONE        09B3 SPCLOOP
 0963 SRCPTR         09E4 SREFNUM       ?0800 START          09CF THISBENTRY
 09D0 THISBLOCK      08BA VISITDIR       08B3 VISITFILE
** SUCCESSFUL ASSEMBLY := NO ERRORS
** ASSEMBLER CREATED ON 15-JAN-84 21:28
** TOTAL LINES ASSEMBLED   544
** FREE SPACE PAGE COUNT   81
```

Further Reference

- ProDOS 8 Technical Reference Manual
- AppleShare Programmer's Guide to the Apple IIGS

### 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 March 1990
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 January 1990: Modified AppleTalk call code for compatibility with ProDOS 8 versions earlier than 1.5 and network-booted version 1.4.

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, DTS presents 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, start with a list of unit numbers for all currently installed disk devices. As you progress through the identification process, you identify some devices immediately, while others must wait until the end of the process for identification.

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 contains $0 and the byte immediately following the unit number contains an error code. For more information on the ON_LINE call, see section 4.4.6 of the ProDOS 8 Technical Reference Manual. A more detailed discussion of the error codes follows later in this Note.

To identify the devices in the device list, you need to know in which physical slot the hardware resides, so you 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 are mapped to slot 2; this mapping gives these two devices unit numbers of $20 and $A0 respectively, but the device's driver entry point is still 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. Here is the code example from that Note 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.

In the following example, note the check for ProDOS 8 version 1.5 or higher, and the simulation of a bad command error under older versions (the $42 call under ProDOS 8 version 1.4 always crashes if ProDOS was launched from a local disk):

```
Network     LDA    #$04        ;require at least ProDOS 8 1.4
            CMP    $BFFF       ;KVERSION (ProDOS 8 version)
            BEQ    MoreNetwork ;have to check further
            LDA    #$01        ;simulate bad command error
            BCS    ERROR       ;if 3 or less, no possibility of network
            BCC    NetCall     ;otherwise, try the network call
```

```
MoreNetwork LDA    $BF02       ;high byte of the MLI entry point
            AND    #$F0        ;strip off the low nibble
            CMP    #$C0        ;is the entry into the $Cn00 space?
            BEQ    NetCall     ;yes, so try AppleTalk
            LDA    #$01
            SEC
            BCS    ERROR       ;simulate bad command error

NetCall     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 contains $88 (Network Error), and the result code in the parameter block contains 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 retains the maximum number of current sessions possible and returns 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 is 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 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 has 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:

- The device is a Disk II with driver code inside ProDOS.
- 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.
- 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 does not recognize any card which is not an 80-column card in slot 3 (see ProDOS 8 Technical Note #15, How ProDOS 8 Treats Slot 3). 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 is always the physical slot number. Disk II devices can never be mirrored to another slot (the Disk II driver does not support it); therefore, it is 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 gives 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. The method for making SmartPort calls is outlined in the Apple IIc Technical Reference Manual, Second Edition 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, you 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, you 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 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, you interpret I/O error to mean the disk in the drive is either damaged or blank (not formatted). You interpret Device off-line to mean that there is no disk in the drive. You 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 (APDA)
- ProDOS 8 Technical Note #15, How ProDOS 8 Treats Slot 3
- ProDOS 8 Technical Note #20, Mirrored Devices and SmartPort
- ProDOS 8 Technical Note #23, ProDOS 8 Changes and Minutia
- ProDOS 8 Technical Note #26, Polite Use of Auxiliary Memory

### 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

- 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

Revised by: Matt Deatherage May 1992
Written by: Matt Deatherage July 1989

This Technical Note documents the change history of ProDOS 8 through V2.0.1, and it supersedes the information on this topic in the ProDOS 8 Technical Reference Manual and the ProDOS 8 Update.

CHANGES SINCE SEPTEMBER 1990: Updated to include ProDOS 8 version 2.0.1 and its known bugs. Replaced APDA references with Resource Central .

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, V2.0.1, which began shipping with Apple IIgs System Software 6.0. 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

Table 1-ProDOS 8 Documentation

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. If you have documentation that predates ProDOS 1.0, you should seek professional help from Resource Central at the address listed in Technical Note #0.
ProDOS 1.0.1

- o 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 expansibility 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.

- Changed the MLI entry to disable interrupts until after the MLIACTV flag is set and other ProDOS parameters are initialized.
- 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.
- 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 on-line 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.
- Added a spurious interrupt handler to allow up to 255 unclaimed interrupts before system death.
- 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).
- Changed MLIACTV to use an ASL instead of an LSR to turn "off" the flag.
- 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.
- 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.
- 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.
- Updated the slot-based clock driver's year table through 1991.
- 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.
- Changed loader code so ProDOS 8 could be loaded by ProDOS 16 without automatically executing the ATINIT and the first .SYSTEM file.
- 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.
- 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.)
- 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.

WARNING : The AppleTalk command, which was added in version 1.5, is present as a skeleton in this version. Unfortunately, it's not a useful skeleton. It moves a section of memory from a ProDOS location to another location and transfers control, totally oblivious of the fact that there is no code at this address.

Even more unfortunate, the server software that ships with the Apple IIe Workstation Card is such that when the IIe is booted over the network with that server software, it is version 1.4 (KVERSION = 4).

So if you boot version 1.4 from a local disk, making a $42 call is fatal. See ProDOS 8 Technical Note #21, Identifying ProDOS Devices, for a reliable way to identify AppleTalk volumes under ProDOS 8 version 1.4.

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 without booting over the server (see the warning under version 1.4). 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 (the cause of the previous warning) 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 do 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) may 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 causes 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.
- o On a IIgs, installs a patch into the GS/OS stack-based call vector so

that anyone calling GS/OS routines (like QDStartUp in ROM 03, for example) gets an appropriate error instead of performing a JSL into the stratosphere.

PRODOS 8 1.9

- o New selector and dispatcher code was added for machines with 80 columns. The old code is still present for machines without 80-column capability.
- o Fixed two bugs involved in booting into a ".SYSTEM" program larger than 38K. First, ProDOS 8 should be able to boot into a program as large as 39.75K, but was returning an error if the ".SYSTEM" program was larger than 38K. Second, when attempting to print the message "*** SYSTEM PROGRAM TOO LARGE ***", only one asterisk was printed. Both these bugs are fixed.
- o No longer requires a ".SYSTEM" file when booting. If ProDOS 8 does not find a ".SYSTEM" file and the enhanced selector and dispatcher code is installed, ProDOS 8 executes a QUIT call.
- o KVERSION is still $08.

PRODOS 8 V2.0.1

- o ProDOS 8 now supports more than two SmartPort devices per slot by remapping the third device and beyond to different slots. There's still a limit of 14 devices altogether, though.
- o ProDOS 8 version 2.0.1 and later require a 65C02 microprocessor or equivalent; you get RELOCATION/CONFIGURATION ERROR if you don't have one. ProDOS 8 tests for a 65C02 by setting binary-coded decimal (BCD) mode and adding $01 to $99, which is the largest negative BCD value representable in an 8-bit register. 65C02 microprocessors correctly clear the N flag when the addition wraps to zero; 6502 microprocessors do not.

  Since all of Apple's 65C02 or greater computers also have lower-case capability, the ProDOS 8 splash screen now uses lower-case letters. After only nine years, too.

- o The file's been rearranged again, so if you have a program that patches the P8 file, it probably breaks now. Please learn your lesson and write a .SYSTEM program that patches ProDOS 8 in memory and not on disk.
- o The prefix is now set correctly when launching Applesoft programs.
- o Old never-used code to handle call $42 is now gone.
- o Removed some RAM-disk code that was not used.
- o ProDOS 8 now sets the prefix to empty when you try to set the prefix to "/".
- o The Apple IIgs clock driver inside ProDOS 8 now limits the year to the range 00 through 99.
- o Sparse seedling files are now truncated properly.
- o When filling up a volume with a WRITE call, ProDOS 8 used to return the disk full error but leave the file's mark set past the file's EOF. This is now fixed.
- o If you try to mount a new volume but all eight VCB slots are filled, ProDOS 8 now tries to kick out the first volume in the table with no open files. If all volumes have open files, you'll still get error $55.

- o The new quit code (introduced with 1.9) now beeps and lets the user try another subdirectory if the one they chose can't be opened. Previously it went forward to the next volume.
- o The new quit code also now closes a directory if it gets a ProDOS error in the directory read loop.
- o When synthesizing a directory entry for a volume, ProDOS 8 always used to assume the directory was four blocks long (for 51 files). The /RAM disk's directory is shorter than this (one block), and ProDOS 8 no longer returns funky errors when trying to read past the end of this shortened directory. The EOF and blocks used are now returned as $200 and 1, respectively.
- o The system death messages are now displayed in the center of the 40-column screen, bordered by inverse spaces. This is an improvement over the line of garbage showing at the bottom of the screen since approximately version 1.5.
- o The new quit code was rearranged to clear the screen prior to loading the selected application. This insures that MSLOT ($07F8) points to the "boot" slot prior to starting the application. In this way, you can launch the ProDOS file from GS/OS to start up GS/OS. (Note that MSLOT must be set properly for this to work.)
- o If the device search code at start time finds a SCSI SmartPort, a SmartPort status call is issued to device #2. This lets the Apple II High-Speed SCSI card build its device tables and return the true number of devices connected. Without this, it always returns "4" for slot 5 or "2" for any other slot.
- o KVERSION is now $21.

KNOWN PRODOS 8 V2.0.1 BUGS

- o ProDOS 8 still doesn't behave perfectly when 14 or more devices are present. Specifically, the /RAM driver tends to install itself without checking to see whether or not there's room in the device table.

CAUTION : ProDOS 8's remapping of SmartPort devices may interfere with intelligent SmartPort peripherals that were already doing their own remapping. ProDOS 8 remaps additional SmartPort devices, even if the SmartPort firmware already did this on its own, and this can cause problems. We never said this would work, but we never said it wouldn't--ProDOS 8 has no way to determine what remapping has already been done. If you make such a card and your customers have problems, tell them to disable your SmartPort remapping and let ProDOS 8 do it all.

Further Reference

- o ProDOS 8 Technical Reference Manual
- o ProDOS 8 Updat
- o AppleShare Programmer's Guide to the Apple II
- o ProDOS 8 Technical Note #21, Identifying ProDOS Devices

### END OF FILE TN.PDOS.023

```
#######################################################################
### FILE: TN.PDOS.024
#######################################################################
```

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#24: BASIC.SYSTEM Revisions

Revised by: Matt Deatherage May 1992
Written by: Matt Deatherage July 1989

This Technical Note documents the change history of BASIC.SYSTEM through V1.5, which ships with Apple IIgs System Software 6.0. V1.0, the initial release, is not documented in this Note, and V1.1 is described in BASIC Programming with ProDOS.

CHANGES SINCE SEPTEMBER 1990: Revised to include BASIC.SYSTEM 1.5.

V1.1

- o Fixed a bug in variable packing (used by CHAIN, STORE, and RESTORE).
- o Changed the interpreter to use the ProDOS startup convention of a JMP instruction followed by two $EE bytes and a startup pathname buffer.
- o Removed a bad buffer address in the FIELD parameter of the READ routine.
- o 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.
- o Added the BYE command to allow ProDOS QUIT calls from BASIC.
- o Removed the limited support for run-time capabilities which had been present.

V1.2

- o 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).
- o 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

- o 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.
- o Fixed a bug in CHAIN and STORE where they expected one branch to go two ways at the same time.
- o Added the MTR command for easier access to the Monitor from BASIC.

- Made internal changes to the assembly process for easier project management. These changes do not affect the code image.

V1.4

- Fixed a bug which caused a BLOAD into an address marked as used in the global page to start performing a BSAVE on the file instead of returning the NO BUFFERS AVAILABLE message. For this reason, BASIC.SYSTEM version 1.3 should not be used.

V1.4.1

- Fixed a bug in the mark handling routines. When using the "B" parameter to indicate a byte to use as a file mark, the third and most significant byte would never be reset before the next use of B. For example, if you used a B value of $010000 and then used a B value of $2345, BASIC.SYSTEM 1.4 would use $012345 for the second B parameter value.

V1.5

- Fixed centuries-old bug where NOTRACE after a THEN (as in IF/THEN) disconnected BASIC.SYSTEM. Now it doesn't.
- BSAVE now modifies the auxtype of an existing file only if the file type is $06 (BIN).
- BASIC.SYSTEM can now launch (with "-") GS/OS applications. Files of type $B3 are passed through to an extended QUIT call to the ProDOS 8 MLI.
- $B3 files are now listed as S16 in the catalog.
- Fixed a bug in the READ command where reading from the slot 3 /RAM disk passed errors back to BASIC, making the program break without completing a legal operation.
- Code optimized and crunched slightly.

Further Reference

- BASIC Programming with ProDOS
- 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

Revised by: Matt Deatherage December 1991
Written by: Matt Deatherage July 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.

Changes since July 1989: Included new information on storing HFS Finder information in extended files' extended key blocks.

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.

Immediately following the mini-entry for the data fork may be up to two eighteen-byte entries, each with part of the HFS Finder information for this file. The first entry stores the first 16 bytes of the Finder information, and the second entry stores the second 16 bytes. The format is as follows:

entry_size	(+008)	Byte	Size of this entry; must be 18 ($12).
entry_type	(+009)	Byte	Type of this entry--1 for FInfo (first 16 bytes of Finder information), 2 for xFInfo (second 16 bytes).
FInfo	(+010)	16 Bytes	First sixteen bytes of Finder Info.
entry_size	(+026)	Byte	Size of this entry; must be 18 ($12).
entry_type	(+027)	Byte	Type of this entry--1 for FInfo (first 16 bytes of Finder information), 2 for xFInfo (second 16 bytes).
xFInfo	(+028)	16 Bytes	Second sixteen bytes of Finder Info.

Note: Although the ProDOS FST under GS/OS will only create both of the mini-entries, as described above, the ProDOS File System Manager (ProDOS FSM) for the Macintosh, which is part of the Apple IIe Card v2.0 software, may create only one of the entries, so you may find an entry_type of 2 at offset +009 in the block. If one of the entries is missing, it should be considered to be all zeroes.

All remaining bytes in the extended key block are reserved and must be zero.

Further Reference

- Apple II Pascal ProFile Manager Manual
- GS/OS Reference
- ProDOS 8 Technical Reference Manual

### END OF FILE TN.PDOS.025

```
#####################################################################
### FILE: TN.PDOS.026
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#26: Polite Use of Auxiliary Memory

Written by: Matt "Missed Manners" Deatherage January 1990

This Technical Note discusses the use of auxiliary memory, particularly the reserved areas, and this information supersedes the discussion in the ProDOS 8 Technical Reference Manual.

"I want to use auxiliary memory!"

Dear Missed Manners:

I'm having difficulty in a program I'm writing for 128K Apple II computers. My program is about to run out of memory. I have squeezed, packed and compressed this program until I can simply cajole no more room from it, and yet more room it needs. Apple has a large section of memory reserved, but my investigations reveal that this memory (in a language card, where it is doubly valuable since it stays put when main memory is swapped) seems to be unused. The ProDOS 8 Technical Reference Manual states unfailingly that the memory must not be used, but it seems to be wasting away! How can I politely use this valuable resource in my own application?

Gentle Developer:

Polite programming requires cooperation by both developers and system software, and it is the users who suffer when that cooperation is not maintained. Apple reserves memory for system software so that it can expand without breaking applications. Missed Manners hopes that he is not being too presumptuous by assuming that you would be appalled if Apple was required to expand ProDOS 8 and reclaim the memory from $B000 through $BFFF. He notes this situation would not be necessary if Apple were able to use memory it currently has reserved for such purposes.

However, if necessity requires more memory for your application, a polite inquiry to Apple may be sent. "Would it be possible for me to use some of Apple's reserved memory in my application without compatibility problems?" would be a polite request, for example. Using the memory without asking or demanding action would not only be impolite, it would pose future problems for an application. Those who do not program politely will eventually regret such a decision.

Conflicts and Arbitration

Some of the polite letters Apple has received on this subject point out that the built-in /RAM device uses almost all of the memory marked as "reserved" in the ProDOS 8 memory map. How can the system software expand into areas it's already using?

It can't, of course...unless it already has and you don't know it. This is partially the case. On the Apple IIGS, memory can be obtained through the Memory Manager, so adding new components to the system software is relatively easy. If memory is available, it is allocated by the Memory Manager and used by the application. If memory is not available, the program trying to install the component in question is told and the component is not installed. (If a vital part of the system can't be installed, the boot process grinds to an unceremonious, but grammatically correct, halt.)

Since the 8-bit Apple II family has no memory manager, applications and system software must mutually (and politely) agree which areas of memory belong to whom. If the system software is broken into components, some memory will be reserved for components which are not present at a given time. This is largely the case with the auxiliary language card memory on the 128K Apple II.

The area from $D100 through $DFFF in bank 2 of the auxiliary language card is for the use of third-party RAM-based drivers, to be discussed in a future ProDOS 8 Technical Note. At least one version of Apple II SANE is configured to load at $E000 in the auxiliary language card, which is perfectly acceptable since SANE is part of the system software (it just doesn't ship with the system).

Clearly, /RAM can't use this memory at the same time the system software does. This very dichotomy gives the Rule of Auxiliary Memory that simplifies this memory management.

> The Rule of Auxiliary Memory: If /RAM is enabled, all auxiliary memory above location $800 may be used by an application after first removing /RAM as discussed in the ProDOS 8 Technical Reference Manual. /RAM should be reinstalled upon completion.
>
> If /RAM is not enabled, then auxiliary memory above $800 may be used at the application programmer's discretion, but the areas marked as reserved must be respected.

System software use of this area should be denoted by the absence of /RAM. This means that if ProDOS 8 were to ever expand to run only on 128K machines and require auxiliary language card memory, that no /RAM device would be installed by default. Although this seems unlikely, it is nonetheless another indicator that your application should not depend on /RAM to operate.

Similarly, if /RAM is not present when your application is launched, you may not reenable it. If it is present, you may remove it to use the memory if you reinstall it when you're done.

Also note that auxiliary memory below $800 that is not on the 80-column text screen is always reserved and may never be used by applications.

Applications which use reserved memory areas without observing this rule run the risk of storing data over third-party RAM-based drivers (rendering their software useless to peripherals that may require such drivers, like third-party networks, devices for the visually impaired, or closed-system hard disks) or future system software.

Further Reference

- ProDOS 8 Technical Reference Manual

### END OF FILE TN.PDOS.026

```
#####################################################################
### FILE: TN.PDOS.027
#####################################################################
```

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#27: Hybrid Applications

Written by: Matt Deatherage
March 1990

This Technical Note discusses considerations for "hybrid" applications, which use Apple IIGS-specific features from ProDOS 8.

Why Use Hybrid Features?

There are many reasons not to write hybrid applications. If your target machine is the Apple IIGS, it's pretty silly to write a ProDOS 8-based application. You are limited to the slower I/O model of ProDOS 8, you cannot access foreign file systems or large CD-ROM volumes, you cannot reliably access the toolbox (patches to the toolbox are only loaded when GS/OS is booted, which forces you to require GS/OS to be booted), and you cannot work with desk accessories that do disk access (CDAs cannot reliably "save and restore" an area of bank zero to use for ProDOS 8 disk access because they don't know if an interrupt handling routine is located there).

However, applications targeted for all Apple II computers may reasonably wish to take advantage of IIGS features. For example, a word processor or telecommunications program may want to use extra IIGS memory. This Note is your spiritual guide to such features.

Memory Management

Applications wishing to use extended (beyond the lower 128K) memory on the IIGS must, like all IIGS applications, get it from the Memory Manager. This is not a consideration for non-hybrid applications for two reasons. First, when GS/OS launches a ProDOS 8 program, it reserves all of the lower 128K memory for ProDOS 8, so no other component (tool, desk accessory, INIT) can accidentally use that memory. (In fact, if some of the memory is not available, GS/OS refuses to launch ProDOS 8 at all.) Second, when ProDOS 8 is directly booted, none of the memory is allocated since these other components, which might be using the Memory Manager, aren't loaded either.

If your ProDOS 8 application was launched by GS/OS, all of the managed lower 128K has already been allocated for you by GS/OS. If you call MMStartUp, the user ID returned is one belonging to GS/OS. In such cases, the auxiliary field of the user ID is already being used by GS/OS and must not be altered by your application. You also must not call any Memory Manager routine which works on all handles of a given user ID, such as DisposeAll or HUnlockAll. You must manage all handles individually and not by user ID. You may, if you wish, call GetNewID to get a new user ID for use in a user ID-based memory management system. The ID should be of type $1000 (application).

You can tell whether your application was launched by GS/OS by checking OS_BOOT, the byte value at $E100BD. OS_BOOT is $00 when the boot OS was ProDOS 8, indicating that your application was not loaded by GS/OS. If this is the case and you want to use extended IIGS memory, you should call GetNewID to obtain a new application ID then use NewHandle to allocate four handles to hold the memory normally reserved for ProDOS 8 by GS/OS. You should obtain memory at $00/0800 (size $B800), $01/0800 (size $B800), $E0/2000 (size $4000) and $E1/2000 (size $8000). You may then use MMStartUp to register yourself with the Memory Manager; MMStartUp fails if it's being called from an unallocated memory block, so you must allocate the memory your application occupies first.

Further Reference

o Apple IIGS Technical Note #17, Application Startup and the MMStartUp User ID

### END OF FILE TN.PDOS.027

##
### FILE: TN.PDOS.028
##

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#28: ProDOS Dates--2000 and Beyond

Written by: Dave Lyons September 1990

This Technical Note explains how ProDOS year values range from zero to ninety-nine and represent the years 1940 through 2039.

The ProDOS date format uses sixteen bits: seven bits for the year, four for the month, and five for the day (see the ProDOS 8 Technical Reference Manual, page 71). Dates are represented in this format in the parameter blocks for ProDOS 8 MLI calls and in the directories of ProDOS volumes.

In seven bits, 128 different years could be represented, but the proper interpretation of those bits has never been defined clearly until now.

2000? I'll Be Dead By Then Anyway

It's only nine years, folks, and then things get weird. Is that ProDOS year 100 or ProDOS year 0? How do you compare two file-modification dates so it keeps working correctly?

Before you dismiss questions like this, consider just how sure you are that nobody will be using your software in nine years, or whether those few dedicated weirdos are going to call you up on January 1, 2000 to complain. There will be plenty of computer-related problems in 2000, so write your applications right today.

Some Choices

These two possible interpretations were considered and then rejected in favor of The Definition below.

1. Valid years would be from 0 to 99, meaning 1900 to 1999, so ProDOS dates would just "expire" at the end of 1999. No fun.

2. Valid years would be from 0 to 127, meaning 1900 to 2027. This is a little better, except that almost no existing software is prepared to deal with year values outside the 0-to-99 range.

So, you are left with...

The Definition

The following definition allows the same range of years that the Apple IIgs Control Panel CDA currently does:

- A seven-bit ProDOS year value is in the range 0 to 99 (100 through 127 are invalid)
- Year values from 40 to 99 represent 1940 through 1999
- Year values from 0 to 39 represent 2000 through 2039

Note: Apple II and Apple IIgs System Software does not currently reflect this definition.

How to Compare Two Years

To compare two dates, you need to adjust the years to allow for the wrap-around effect between 39 and 40. A simple approach is to add 100 to any year less than 40 before doing the comparison, thus comparing two values in the range 40 to 139.

```
    CompareAB    lda YearB
                 cmp #40
                 bcs B_OK
                 adc #100    ;carry is clear
                 sta YearB

B_OK             lda YearA
                 cmp #40
                 bcs A_OK
                 adc #100    ;carry is clear
                 sta YearA

A_OK             cmp YearB
                 bcc A_is_earlier
                 ...
```

What About GS/OS Dates?

This definition affects how the GS/OS ProDOS File System Translator works internally, but it does not affect GS/OS applications. A year value under GS/OS is always a byte offset from 1900, giving a possible range of 1900 to 2155, regardless of the file system involved.

What Do You Do After 2039?

Apple is still working on it. Contact your neighborhood Apple Developer Technical Support office in 2030.

Further Reference

- ProDOS 8 Technical Reference Manual
- Apple IIgs Toolbox Reference Manual, Volume 1
- GS/OS Reference

### END OF FILE TN.PDOS.028

###
### FILE: TN.PDOS.029
###

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#29: Clearing the Backup Needed Bit

Written by: Jim Luther September 1990

This Technical Note shows how to clear the "backup needed bit" in a directory entry's access byte.

If you are writing a file backup utility program, you probably want to clear the backup needed bit in each directory entry's access byte as you make the backup of the file associated with that directory entry. The SET_FILE_INFO MLI call normally sets the backup needed bit of the access byte, but how do you clear it? The answer is at location BUBIT ($BF95) on the ProDOS 8 system global page.

BUBIT normally contains the value $00. When BUBIT contains $00, the SET_FILE_INFO MLI call always sets the backup needed bit in the directory entry's access byte. However, if the value $20 is stored in BUBIT immediately before calling SET_FILE_INFO, the backup needed bit in the directory entry's access byte can be cleared. BUBIT is set back to $00 by the MLI call. The following code example shows how to clear the backup needed bit. Values other than $20 or $00 in BUBIT are not supported.

```
; The pathname of the file should be in ThePathname buffer when this code is
called!

               65816 off
               longa off
               longi off

ClearBackupBit start

; System global page locations

MLI            equ $BF00              ;MLI call entry point
BUBIT          equ $BF95              ;Backup Bit Disable, SET_FILE_INFO only

; MLI call numbers

SET_FILE_INFO  equ $C3
GET_FILE_INFO  equ $C4

; set up FileInfoParms for GET_FILE_INFO MLI call
               lda #$0A
               sta param_count
; then...
               jsr MLI                  ;get the current file info
               dc  I1'GET_FILE_INFO'
               dc  I2'FileInfoParms'
               bne Error

               lda #$20                 ;set the backup bit disable bit
               sta BUBIT
               eor #$FF
               and access               ;clear the backup needed bit
               sta access

; set up FileInfoParms for SET_FILE_INFO MLI call
               lda #$07
               sta param_count
; then...
               jsr MLI                  ;set the file info with the file info
               dc  I1'SET_FILE_INFO'   ;(clearing only the backup needed bit)
               dc  I2'FileInfoParms'
               bne Error
               rts                      ;return to caller

Error          anop                     ;routine to handle MLI errors
               rts

; Parameter block used for GET_FILE_INFO and SET_FILE_INFO MLI calls

FileInfoParms  anop
param_count    ds  1
pathname       dc  i2'ThePathname'
access         ds  1
file_type      ds  1
aux_type       ds  2
storage_type   ds  1
blocks_used    ds  2
mod_date       ds  2
mod_time       ds  2
create_date    ds  2
create_time    ds  2

ThePathname    entry
               ds  65                   ;store the pathname of the file here

               end
```

Further Reference

---

  o ProDOS 8 Technical Reference Manual

### END OF FILE TN.PDOS.029

```
######################################################################
### FILE: TN.PDOS.030
######################################################################
```

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#30: Sparse Station

Written by: Matt Deatherage

May 1992

This Technical Note discusses issues when using sparse files under ProDOS 8.

SPARSE INFORMATION AVAILABLE

The concept of sparse files is introduced in the ProDOS 8 Technical Reference Manual in sometimes confusing language. The concept behind sparse files is pretty simple. If you didn't think it could be explained in two paragraphs, have a seat and learn something.

The ProDOS file system keeps track of where files reside on disk through a series of "index blocks." All index blocks are disk blocks that contain lists of block numbers. They may be organized in several ways (seedling, sapling or tree), depending on how big the file is--one 512-byte block can hold 256 two-byte block numbers. If a file is one block long, it has no index blocks and is a seedling file. If a non-sparse file is between two and 256 blocks long, it has one index block and is a sapling file. If a non-sparse file is longer than 256 blocks, it's a tree file and has a "master index block" that points to other index blocks. This is more than enough to store any ProDOS file--one master index block pointing to 256 other index blocks, each of which points to 256 data blocks on disk would be a 32 MB file--twice the limit of 16 MB imposed by ProDOS's 3-byte storage for file lengths.

What happens if you don't need to use all of those blocks? For example, if you need to store data at file offset $0000 and at file offset $20000, does ProDOS make you waste 256 disk blocks you're not going to use? Fortunately, the answer is "no." ProDOS lets you skip any data block you're not using by recording a pointer to data block $0000 instead of to a regular block on the disk. When ProDOS sees a block pointer of $0000 in an index block, it knows not to read block zero (which contains boot code) but instead to pretend that it read a block of zeroes from the disk. This lets you save lots of space on disk--a file created this way is a sparse file. (See? Two paragraphs.)

Under ProDOS 8, you can create a sparse file by using the SET_EOF MLI command to extend the file's current end-of-file position, and then using SET_MARK to move the mark to the new end-of-file position. If you grow a file by increasing the EOF but not actually writing data, ProDOS 8 makes the blocks you skip sparse. Under GS/OS, the ProDOS FST automatically converts long stretches of zeroes to sparse blocks, making sparse files even more prevalent.

DEALING WITH SPARSITY

Unfortunately, ProDOS 8 does not automatically make sparse files when you write large sections of zeroes. That means if you read a sparse file and write it back out, you "expand" it and it's no longer sparse. The file could balloon to hundreds of times its previous disk space, which is not a good thing.

So how do you recognize a sparse file? You can notice that the length of the file has to be pretty close to 512 bytes multiplied by the number of blocks allocated to data in the file. For example, take a file that's $4068 bytes long. $4068 bytes takes 33 512-byte blocks--32 blocks is $4000 bytes, plus one more block for the last $68 bytes. This is between 2 and 256 blocks, so there's one more block allocated for the index block. If this file is not sparse, it uses 34 blocks on disk. If it uses any less than 34 blocks in reality, it's sparse.

This calculation gets a little tricker for tree files--if the file has more than 256 data blocks, add one master index block plus one index block for each 256 data blocks or portion thereof. To give another example, a file that's $68D3F bytes long takes 839 ($347) data blocks. This file has five additional blocks allocated to it--one master index block and four index blocks. The first three index blocks are full (256 Y 3 = 768) and the fourth contains the remaining 71 data blocks. If this file takes less than 844 blocks on disk, it's sparse.

TOO COMPLICATED?

For all except very speedy utilities to copy files, yes. If you just need an easy way to deal with sparse files that's not so speed-critical, read on.

All you have to do to preserve (or create) sparsity in normal file copying operations is scan the data you've read from disk before you write it back. Suppose your file copying buffer is 10K large. Read 10K of data from your source file, then divide the buffer into 512-byte chunks and scan the data looking for zeroes. If you find a non-zero byte, write the entire 512-byte chunk of data to the target file and proceed to the next 512-byte chunk. If you don't find any non-zero bytes in a 512-byte chunk, just set the mark ahead 512 bytes and don't issue a WRITE call. This is basically how GS/OS's ProDOS FST automatically sparses files, and it can work for you too.

IS IT THAT EASY?

Well, no. There's an important exception--AppleShare.

Most AppleShare servers (including all of Apple's) don't support sparse files--all the logical blocks you use have to be physically allocated on the server's hard disk. The following BASIC.SYSTEM command:

```
     BSAVE SPARSE.FILE,A$300,L$1,B$FFFFFF
```

creates a 16 MB sparse file with one byte of logical data in it. This file only takes 5 blocks on a ProDOS disk (one master index block, two index blocks and two data blocks--it takes two data blocks because ProDOS 8 always allocates the very first block of a file when you create it, even if you don't use the first 512 bytes), but it takes 16 MB of disk space on a server.

That's not all--for speed reasons, AppleShare does not fill the extra, normally-sparsed blocks with zeroes. If you issued the above command to an

AppleShare server under ProDOS 8 and then tried to read the first few bytes of the resulting file, they would be garbage--but not zeroes.
ProDOS 8 Technical Note #21 gives information on identifying AppleShare server volumes--if you're dealing with one, do not use normal sparse file creation techniques. Just write the 512 bytes of zeroes instead of advancing the mark. It doesn't take any more disk space and it achieves the results you want.

ONE MORE THING

In versions of ProDOS 8 up to 1.9, setting the end-of-file position past $200 on a seedling file created a sparse file that confused ProDOS 8 if you ever used SET_EOF on it again. This is fixed in version 2.0.1 and later.

Further Reference

- ProDOS 8 Technical Reference Manual

### END OF FILE TN.PDOS.030

```
#####################################################################
### 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, Second Edition.

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, Second Edition

### 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, Second Edition. 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, Second Edition
- 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, Second Edition

### 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: Matt Deatherage & Llew Roberts November 1990
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 January 1989: Added notes about the Apple II High-Speed SCSI Card.

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 returns 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. The Apple II High-Speed SCSI Card returns version $3000 (3.0).

To maintain compatibility with the Apple II High-Speed SCSI Card and future SCSI products, you should use the following guidelines when programming with the revision C card:

o Avoid access to the hardware or any RAM locations on the SCSI card.
o Do not use the Patch1Call, SetNewSDAT, or SetBlockSize control calls.
o 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.
o Never Reset the SCSI bus.

The Apple II SCSI Card firmware was designed to operate with SCSI CD-ROM and disk drives only; however, the Apple II High-Speed SCSI Card works with most

SCSI devices, including removable drives, scanners, and tape backup units.

Further Reference

- Apple II SCSI Card Technical Reference
- 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

Revised by: Matt Deatherage — November 1990
Written by: Matt Deatherage — November 1988

This Technical Note documents two bugs in the Apple IIgs SmartPort firmware.
Changes since November 1988: Documented corrections in ROM 03 and an additional ROM 03 bug.

Developers should be aware of the following two bugs in the Apple IIgs ROM 01 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.

The ROM 03 firmware fixes these bugs; however, it has a bug in the ResetHook device-specific CONTROL call for the Apple 3.5" Drive. With this bug, hook numbers of nine or greater crash the machine. At present, hook numbers in this range are invalid, so this bug should not be a problem.

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
o   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.SMPT.009
########################################################################
```

Apple II
Technical Notes

Developer Technical Support

SmartPort
#9:   Apple II SCSI Errata

Written by: Llew Roberts   July 1990

This Technical Note documents SCSI-specific anomalies that were discovered in the development of the Apple II High-Speed SCSI card.

During the testing of the Apple II High Speed SCSI Card, several SCSI-related issues of interest to developers were discovered. Most of these issues directly relate to SCSI devices with removable media.

If a CD-ROM has a bad directory, the firmware hangs because it gets a "Read In Progress" error. There is no timeout. The system can be recovered if the CD-ROM is manually ejected.

If the media is ejected while a SmartPort DIB Status call is in progress, the call returns with incorrect results (it reports DISKSW and that the block count of the device is zero). In order to avoid this conflict, bracket the DIB Status call with the SmartPort SCSI specific Control Calls Prevent Removal ($1A) and Allow Removal ($1B).

A SmartPort FORMAT call, in some cases, does not update the DIB correctly to reflect the new or changed partition map. If a SmartPort Init call is always made immediately after the FORMAT, the DIB is rebuilt correctly. Note that after formatting a tape, the tape should be removed from the SCSI tape drive before an Init call is made. An Init call causes the tape to rewind, hanging the system with no time-out until the rewind is complete.

Further Reference

- Apple IIgs Firmware Reference
- Apple II High-Speed SCSI Card Technical Reference

### END OF FILE TN.SMPT.009

##
### 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, Second Edition

### 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.

o 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