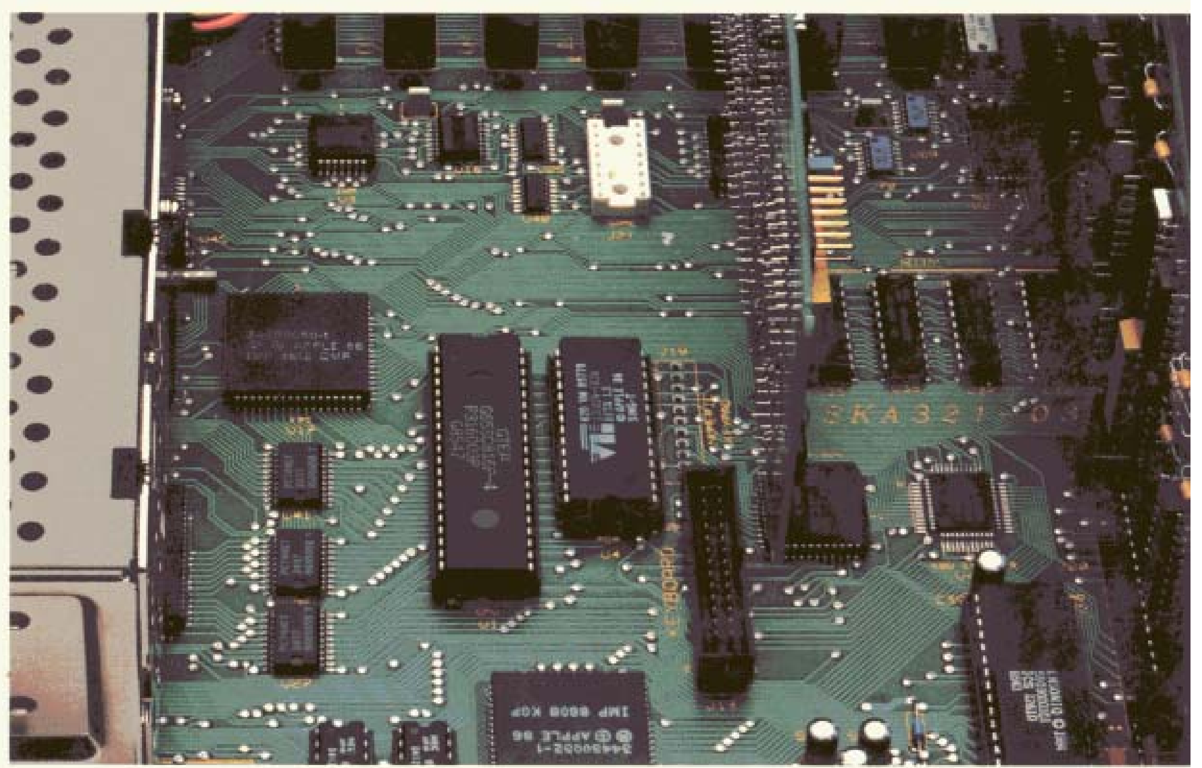
Apple® II

Apple IIGS™ Firmware Reference



> $24.95 FPT
USA

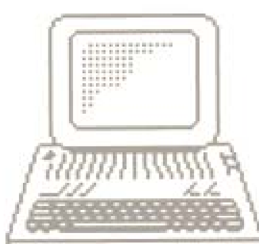
The Apple IIGS Technical Library
The Official Publications from Apple Computer, Inc.

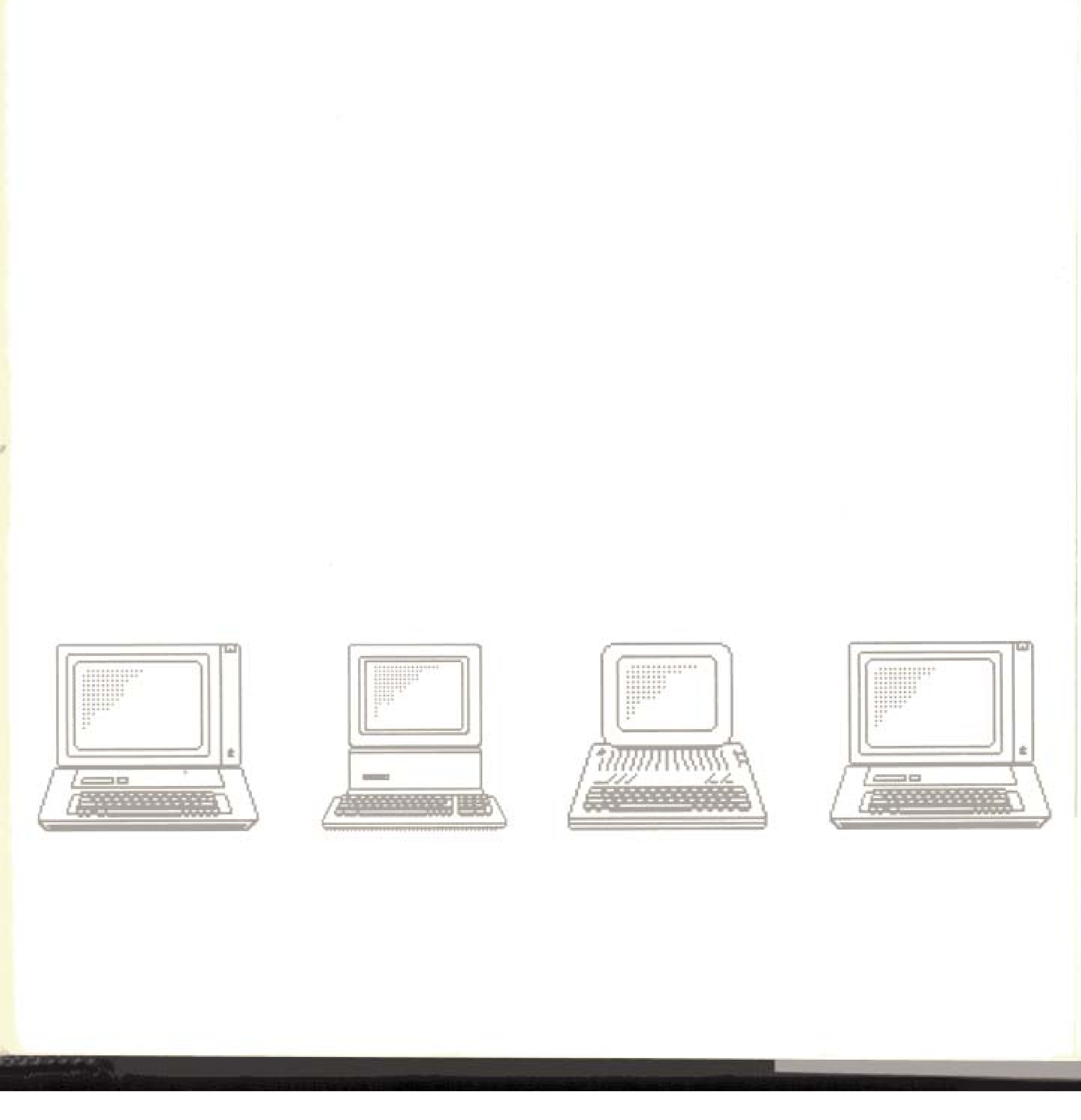
Now Apple IIGS™ programmers and enthusiasts can have comprehensive and definitive information about the Apple IIGS system at their fingertips. The Apple IIGS Technical Library provides much-needed information on all aspects of the Apple IIGS, ranging from details of the new logic board and firmware to a complete description of the more than 800 powerful routines in the Apple IIGS Toolbox. Whether you need a technical overview, details on the new ProDOS® 16 operating system, or information on Macintosh™-style event-driven programming, the Apple IIGS Technical Library has the complete and definitive answers.

These books, written and produced by Apple Computer, Inc., provide authoritative references for those interested in getting the most out of their Apple IIGS.

Titles in the Apple IIGS Technical Library and related titles from the Apple Technical Library include:

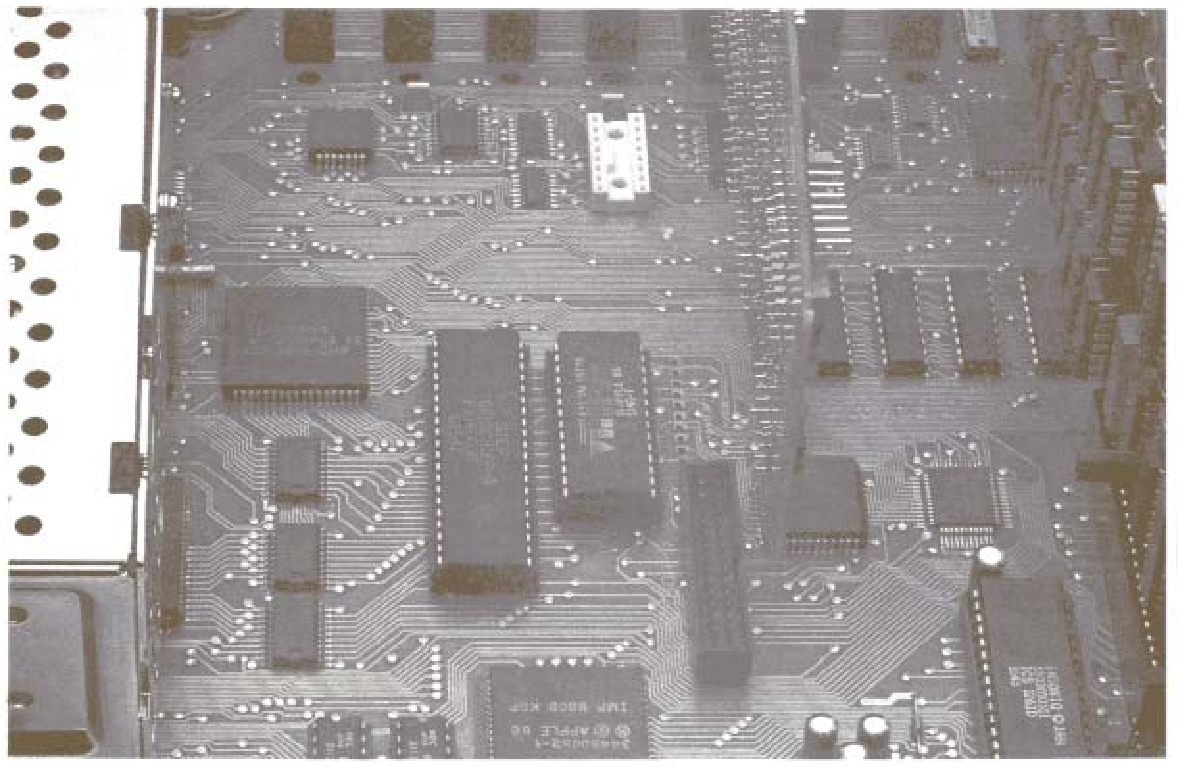
Technical Introduction to the Apple IIGS
Apple IIGS Hardware Reference
Apple IIGS Firmware Reference
Apple IIGS Toolbox Reference, Volume I
Apple IIGS Toolbox Reference, Volume II
Apple IIGS ProDOS 16 Reference
Programmer's Introduction to the Apple IIGS
Apple Numerics Manual
ImageWriter® II Technical Reference Manual





Apple® II

Apple IIGS™ Firmware Reference



Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts Menlo Park, California Don Mills, Ontario
Wokingham, England Amsterdam Bonn Sydney Singapore Tokyo
Madrid Bogotá Santiago San Juan

APPLE COMPUTER, INC.

Copyright © 1987 by Apple Computer, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc. Printed in the United States of America.

Apple, the Apple logo, AppleTalk, Disk II, DuoDisk, LaserWriter, and ProDOS are registered trademarks of Apple Computer, Inc.

Apple DeskTop Bus, AppleMouse, Apple IIGS, Macintosh, SANE, and UniDisk are trademarks of Apple Computer, Inc.

ITC Garamond, ITC Avant Garde Gothic, and ITC Zapf Dingbats are registered trademarks of International Typeface Corporation.

Microsoft is a registered trademark of Microsoft Corporation.

POSTSCRIPT is a trademark of Adobe Systems Incorporated.

Simultaneously published in the United States and Canada.

ISBN 0-201-17744-7
ABCDEFGHIJ-DO-8987
First printing, May 1987

WARRANTY INFORMATION

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Figures and tables xiii

Preface xvii

About this manual xvii

What this manual contains xviii

Chapter 1 **Overview** 1

A word about other Apple IIGS firmware 2
- Apple IIGS Toolbox 2
- Applesoft BASIC 2
- AppleTalk 3
- Diagnostic routines 3

The role of firmware in the Apple IIGS system 3

Levels of program operation 4

Apple IIGS firmware 4
- System Monitor firmware 4
- Video firmware 5
- Serial-port firmware 5
- Disk II support 5
- SmartPort firmware 5
- Interrupt-handler firmware 6
- Apple Desktop Bus microcontroller 6
- Mouse firmware 6

Chapter 2 **Notes for Programmers** 7

Introduction to the Apple IIGS 8
- Microprocessor features 8
  - Microprocessor modes 9
  - Execution speeds 9
  - Expanded memory 9
- Super Hi-Res display 9
- Digital sound synthesizer 10
- Detached keyboard with Apple DeskTop Bus 10
- Built-in I/O 11
- Compatible slots and game I/O connectors 11

Environment for the firmware routines 11
Setting up the system 12
Save your environment 12
Get into bank $00 12
Set the D register to $0000 12
Set the DBR to $00 13
Save the value of the native-mode stack pointer 13
Select emulation mode 14
Returning to native mode 14
Restore the native-mode stack pointer 14
Restore your environment 14
Other requirements for emulation-mode code 15
Cautions about changing the environment 15
Stack and direct page 15
Data bank registers and e, m, and x flags 16
Speed- and Shadow-register changes 16
Language-card changes 16
General information 16
Apple IIGS interrupts 16
Boot/scan sequence 17
Program bank register 17
Exchanging the B and A registers, XBA 18

Chapter 3 **System Monitor Firmware 19**

Invoking the Monitor 20
Monitor command syntax 21
Monitor command types 21
Monitor memory commands 25
Examining memory 26
Examining consecutive memory locations 27
Changing memory contents 28
Changing one byte 28
Changing consecutive memory locations 29
ASCII input mode 30
ASCII filters for stored data 31
Moving data in memory 31
Comparing data in memory 33
Filling a memory range 34
Searching for bytes in memory 34
Registers and flags 35
The environment 36
Examining and changing registers and flags 36
Summary of register- and flag-modification commands 38

Miscellaneous Monitor commands 39
Inverse and normal display 39
Working with time and date 40
Redirecting input and output 40
Changing the cursor character 41
Converting hexadecimal and decimal numbers 41
Hexadecimal math 42
A Tool Locator call 43
Back to BASIC 43
Special tricks with the Monitor 44
Multiple commands 44
Filling memory 45
Repeating commands 46
Creating your own commands 47
Machine-language programs 48
Running a program in bank zero 49
Running a program in other banks of memory 50
Resuming program operation 50
Stepping through or tracing program execution 50
The mini-assembler 51
Starting the mini-assembler 51
Using the mini-assembler 51
Mini-assembler instruction formats 54
The Apple IIGS tools 55
The disassembler 55
Summary of Monitor instructions 57

Chapter 4 **Video Firmware 69**

Standard I/O links 70
Standard input routines 71
RDKEY input subroutine 71
KEYIN and BASICIN input subroutines 71
Escape codes 72
Cursor control 72
GETLN input subroutine 74
Editing with GETLN 75
Keyboard input buffering 75
Standard output routines 76
COUT and BASICOUT subroutines 76
Control characters with COUT1 and C3COUT1 76
Inverse and flashing text 78
Other firmware I/O routines 79
The text window 80

Chapter 5 Serial-Port Firmware 81

Compatibility 82
Operating modes 83
 Printer mode 83
 Communications mode 83
 Terminal mode 83
Handshaking 84
 Hardware, DTR and DSR 84
 Software, XON and XOFF 85
Operating commands 86
 The command character 87
 Command strings 87
 Commands useful in printer and communications modes 88
  Baud rate, nB 88
  Data format, nD 88
  Parity, nP 89
  Line length, nN 89
  Enable line formatting, CE and CD 89
  Handshaking protocol, XE and XD 89
  Keyboard input, FE and FD 90
  Automatic line feed, LE and LD 90
  Reset the serial-port firmware, R 90
  Suppress control characters, Z 90
 Commands useful in communications mode 91
  Echo characters to the screen, EE and ED 91
  Mask line feed in, ME and MD 91
  Input buffering, BE and BD 91
  Terminal mode, T and Q 91
  Tab in BASIC, AE and AD 92
Programming with serial-port firmware 92
 BASIC interface 93
 Pascal protocol for assembly language 93
Error handling 95
Buffering 95
Interrupt notification 96
Background printing 97
 Recharge routine 98
Extended interface 99
 Mode control calls 100
  GetModeBits 100
  SetModeBits 100

Buffer-management calls 101
GetInBuffer 101
GetOutBuffer 101
SetInBuffer 102
SetOutBuffer 102
FlushInQueue 102
FlushOutQueue 102
InQStatus 103
OutQStatus 103
SendQueue 103
Hardware control calls 104
GetPortStat 104
GetSCC 104
SetSCC 105
GetDTR 105
SetDTR 105
GetIntInfo 105
SetIntInfo 106

Chapter 6 **Disk II Support 109**
Startup 112

Chapter 7 **SmartPort Firmware 113**
Locating SmartPort 114
Locating the dispatch address 115
SmartPort call parameters 116
SmartPort assignment of unit numbers 117
Allocation of device unit numbers 117
Issuing a call to SmartPort 120
Generic SmartPort calls 121
Status 121
Required parameters 122
SmartPort driver status 125
Possible errors 125
ReadBlock 126
Required parameters 126
Possible errors 126
WriteBlock 127
Required parameters 127
Possible errors 127
Format 128
Format call implementation 128
Required parameters 128
Possible errors 128

Control 129
Required parameters 129
Possible errors 130
Init 130
Required parameters 130
Possible errors 130
Open 131
Required parameters 131
Possible errors 131
Close 131
Required parameters 132
Possible errors 132
Read 132
Required parameters 133
Possible errors 133
Write 134
Required parameters 134
Possible errors 135
Device-specific SmartPort calls 138
SmartPort calls specific to Apple 3.5 disk drive 138
Eject 138
SetHook 138
Read Address Field 139
Write Data Field 139
Seek 139
Format 139
Write Track 139
Verify 140
ResetHook 140
SetMark 140
ResetMark 141
SetSides 141
SetInterleave 141
SmartPort calls specific to UniDisk 3.5 142
Eject 142
Execute 142
SetAddress 143
Download 143
UniDiskStat 143
UniDisk 3.5 internal functions 144
Mark table 144
Hook table 145

UniDisk 3.5 internal routines 146
RdAddr 146
ReadData 146
WriteData 147
Seek 147
Format 147
WriteTrk 148
Verify 148
Vector 149
Memory allocation 150
ROM disk driver 152
Installing a ROM disk driver 152
Passing parameters to a ROM disk 152
ROM organization 154
Summary of SmartPort error codes 156
The SmartPort bus 157
How SmartPort assigns unit numbers 157
SmartPort–Disk II interaction 158
Other considerations 158
Extended and standard command packets 159
SmartPort bus flow of operations 159

Chapter 8 **Interrupt-Handler Firmware 169**

What is an interrupt? 171
The built-in interrupt handler 172
Summary of system interrupts 175
Interrupt vectors 177
Interrupt priorities 177
RESET 178
NMI 178
ABORT 179
COP 179
BRK 179
IRQ 180
Environment handling for interrupt processing 181
Saving the current environment 181
Going to the interrupt environment 182
Restoring the original environment 182
Handling Break instructions 183
Apple IIGS mouse interrupts 183
Serial-port interrupt notification 183

Chapter 9 **Apple DeskTop Bus Microcontroller 185**

ADB microcontroller commands 188
- Abort, $01 188
- Reset Keyboard Microcontroller, $02 188
- Flush Keyboard Fuffer, $03 188
- Set Modes, $04 189
- Clear Modes, $05 189
- Set Configuration Bytes, $06 190
- Sync, $07 191
- Write Microcontroller Memory, $08 191
- Read Microcontroller Memory, $09 191
- Read Modes Byte, $0A 191
- Read Configuration Bytes, $0B 192
- Read and Clear Error Byte, $0C 192
- Get Version Number, $0D 192
- Read Available Character Sets, $0E 193
- Read Available Keyboard Layouts, $0F 193
- Reset the System, $10 193
- Send ADB Keycode, $11 193
- Reset ADB, $40 194
- Receive Bytes, $48 194
- Transmit num Bytes, $49–$4F 194
- Enable Device SRQ, $50–$5F 194
- Flush Device Buffer, $60–$6F 195
- Disable Device SRQ, $70–$7F 195
- Transmit Two Bytes, $80–$BF 195
- Poll Device, $C0–$FF 195

Microcontroller status byte 196

Chapter10 **Mouse Firmware 197**

Mouse position data 199
- Register addresses—firmware only 200
- Reading mouse position data—firmware only 200
- Position clamps 201

Using the mouse firmware 202
- Firmware entry example using assembly language 202
- Firmware entry example using BASIC 203
- Reading button 1 status 204

Mouse programs in BASIC 206
- Mouse.Move program 206
- Mouse.Draw program 207

Summary of mouse firmware calls 209

Pascal calls 210
PInit 210
PRead 210
PWrite 210
PStatus 210
Assembly-language calls 211
SETMOUSE, $C412 211
SERVEMOUSE, $C413 212
READMOUSE, $C414 212
CLEARMOUSE, $C415 212
POSMOUSE, $C416 213
CLAMPHOUSE, $417 213
HOMEMOUSE, $418 214
INITMOUSE, $419 214

Appendix A Roadmap to the Apple IIGS Technical Manuals 215
The introductory manuals 218
The technical introduction 218
The programmer's introduction 218
The machine reference manuals 219
The hardware reference manual 219
The firmware reference manual 219
The toolbox reference manuals 219
The programmer's workshop reference manual 220
The programming-language reference manuals 220
The operating-system reference manuals 221
The all-Apple manuals 221

Appendix B Firmware ID Bytes 222

Appendix C Firmware Entry Points in Bank $00 224

Appendix D Vectors 258
Bank $00 page 3 vectors 259
Bank $00 page C3 routines 260
Bank $00 page Fx vectors 262
Bank $E1 vectors 264
IRQ.APTALK and IRQ.SERIAL vectors 266
IRQ.SCAN through IRQ.OTHER vectors 267
TOWRITEBR through MSGPOINTER vectors 272

Appendix E **Soft Switches 276**

Appendix F **Disassembler/Mini-Assembler Opcodes 293**

Appendix G **The Control Panel 299**

Control Panel parameters 299
- Printer port 300
- Modem port 301
- Display 302
- Sound 303
- Speed 303
- RAM disk 303
- Slots 304
- Options 304
- Clock 306
- Quit 306

Battery-powered RAM 306
Control Panel at power-up 307

Appendix H **Banks $E0 and $E1 308**

Using banks $E0 and $E1 310
- Free space 310
- Language-card area 310
- Shadowing 310

Glossary 311
Index 321

Figures and tables

Chapter 1 **Overview 1**

Figure 1-1 Levels of program operation 4

Chapter 2 **Notes for Programmers 7**

Figure 2-1 Boot-failure screen 17
Figure 2-2 Accumulator for emulation and native modes 18
Table 2-1 Super Hi-Res graphic modes 10

Chapter 3 **System Monitor Firmware 19**

Table 3-1 Monitor commands grouped by type 23
Table 3-2 Commands for viewing and modifying memory 25
Table 3-3 Registers and flags 35
Table 3-4 Commands for viewing and modifying registers 37
Table 3-5 Miscellaneous Monitor commands 39
Table 3-6 Commands for program execution and debugging 48
Table 3-7 Mini-assembler address formats 54
Table 3-8 Opcodes affected in immediate mode 57

Chapter 4 **Video Firmware 69**

Table 4-1 Escape codes and their functions 73
Table 4-2 Prompt characters 74
Table 4-3 Control characters with 80-column firmware off 77
Table 4-4 Control characters with 80-column firmware on 77
Table 4-5 Text format control values 78
Table 4-6 Partial list of other Monitor firmware I/O routines 79

Chapter 5 **Serial-Port Firmware 81**

Figure 5-1 Handshaking when DTR/DSR option is turned on 84
Figure 5-2 Handshaking when DTR/DSR option is turned off 85
Figure 5-3 Handshaking via XON/XOFF 85
Figure 5-4 Summary of extended serial-port buffer commands 107
Figure 5-5 Summary of extended serial-port mode and hardware control commands 108

Table 5-1 Baud-rate selections 88
Table 5-2 Data-format selections 88
Table 5-3 Parity selections 89
Table 5-4 Terminal-mode command characters 92
Table 5-5 Service routine descriptions and address offsets 93
Table 5-6 I/O routine offsets and registers for Pascal 1.1 firmware protocol 94
Table 5-7 Interrupt setting enable bits 106

Chapter 6 **Disk II Support 109**

Figure 6-1 Order of disk drives on Apple IIGS disk ports 110
Table 6-1 Disk II I/O port characteristics 111

Chapter 7 **SmartPort Firmware 113**

Figure 7-1 SmartPort ID type byte 115
Figure 7-2 Device mapping: configuration 1, derivation 1 118
Figure 7-3 Device mapping: configuration 1, derivation 2 118
Figure 7-4 Device mapping: configuration 2, derivation 1 119
Figure 7-5 Device mapping: configuration 2, derivation 2 119
Figure 7-6 Device mapping: configuration 2, derivation 3 119
Figure 7-7 SmartPort device subtype byte 124
Figure 7-8 Disk-sector format 140
Figure 7-9 UniDisk 3.5 memory map 150
Figure 7-10 The ROM disk 154
Figure 7-11 Block diagram of a 128K ROM disk 155
Figure 7-12 SmartPort control flow 159
Figure 7-13 SmartPort bus communications: read protocol 161
Figure 7-14 SmartPort bus communications: write protocol 162
Figure 7-15 SmartPort bus packet format 163
Figure 7-16 SmartPort bus packet contents 164
Figure 7-17 Bit layout of a 7-byte data packet 165
Figure 7-18 Transmitting a 1-byte data packet 165
Table 7-1 Register status on return from SmartPort 121
Table 7-2 Summary of standard commands and parameter lists 136
Table 7-3 Summary of extended commands and parameter lists 137
Table 7-4 UniDisk 3.5 gate array I/O locations 151
Table 7-5 UniDisk 3.5 IWM locations 151
Table 7-6 SmartPort error codes 156
Table 7-7 Data byte encoding table 164
Table 7-8 Standard command packet contents 166
Table 7-9 Extended command packet contents 167

Chapter 8 **Interrupt-Handler Firmware 169**

Figure 8-1 Built-in interrupt handler 172
Table 8-1 Summary of system interrupts 175
Table 8-2 Interrupt vectors 177

Chapter 9 **Apple DeskTop Bus Microcontroller 185**

Figure 9-1 Apple DeskTop Bus components 186
Table 9-1 Bit functions 189
Table 9-2 Keyboard language codes 190
Table 9-3 Status byte returned by microcontroller 196

Chapter 10 **Mouse Firmware 197**

Figure 10-1 Button interrupt status byte, $77C 205
Figure 10-2 Mode byte, $7FC 205
Table 10-1 Apple IIGS mouse data bits 199
Table 10-2 Apple IIGS mouse register addresses 200
Table 10-3 Position and status information 205
Table 10-4 Mouse firmware calls 209

Appendix A **Roadmap to the Apple IIGS Technical Manuals 215**

Figure A-1 Roadmap to the technical manuals 217
Table A-1 Apple IIGS technical manuals 216

Appendix B **Firmware ID Bytes 222**

Table B-1 ID information locations 222
Table B-2 Register bit information 223

Appendix E **Soft Switches 276**

Table E-1 Symbol table sorted by symbol 291
Table E-2 Symbol table sorted by address 292

Appendix G **The Control Panel 299**

Table G-1 Language options 305

Appendix H **Banks $E0 and $E1 308**

Figure H-1 Memory map of banks $E0 and $E1 309

Preface

This is the firmware reference manual for the Apple® IIGS™ computer. It is for hardware designers and programmers who want to work with the system firmware in lieu of using the Apple IIGS Toolbox routines to accomplish similar goals.

About this manual

As part of the Apple IIGS technical suite of manuals, the *Apple IIGS Firmware Reference* covers the design and function of the firmware that drives the Apple IIGS. It provides information about the entry points for the firmware and describes the firmware functions and limitations.

❖ *Note:* None of the manuals in the technical suite stands alone. Other manuals in the suite describe various tools to accomplish tasks that the firmware can also perform. You should become familiar with the contents of the other Apple IIGS manuals because for most applications, you may not need to directly use the firmware.

The audience for this manual includes programmers who want to work with the firmware and application programmers who wish to convert or upgrade existing applications for the Apple II, II Plus, IIe, or IIc to take advantage of the new functions available on the Apple IIGS.

❖ *Note:* Applications written explicitly for the Apple IIe can be booted on the Apple IIGS, with no discernible difference in their operation.

This manual does not incorporate any descriptions of hardware; see the *Apple IIGS Hardware Reference* for this information.

What this manual contains

Chapter 1, "Overview," provides an overview of the Apple IIGS firmware.

Chapter 2, "Notes for Programmers," provides information for those who are already familiar with other Apple II computers.

Chapter 3, "System Monitor Firmware," shows how to use the system Monitor to examine and change memory or registers and to write and debug small machine-language programs.

Chapter 4, "Video Firmware," describes the text input and output facilities of the Apple IIGS.

Chapter 5, "Serial-Port Firmware," describes the features and functions of the built-in serial port.

Chapter 6, "Disk II Support," describes the firmware support for the Apple Disk II® product.

Chapter 7, "SmartPort Firmware," defines and describes the SmartPort firmware as implemented on the Apple IIGS.

Chapter 8, "Interrupt-Handler Firmware," describes in detail the method by which various kinds of interrupts are processed.

Chapter 9, "Apple DeskTop Bus Microcontroller," describes the firmware portion of the Apple DeskTop Bus™. For a complete picture of this subsystem, you need this manual, the *Apple IIGS Hardware Reference,* and the *Apple IIGS Toolbox Reference.*

Chapter 10, "Mouse Firmware," describes the Apple IIGS mouse interface.

Appendix A contains a roadmap to the Apple IIGS technical manuals. Read this appendix to determine which books you need to learn more about a programming language, the Apple IIGS hardware, or some other aspect of the Apple IIGS computer.

Appendix B contains a list of the firmware ID bytes. The information lets you determine which machine in the Apple II family is running your program. By examining these ID bytes, you can allow your program to take advantage of the features available on a particular member of this family.

Appendix C describes the firmware entry points for the Apple IIGS, as well as the side effects of each routine.

Appendix D describes the firmware vectors. By jumping to vectors instead of directly to particular firmware routines, you can maintain compatibility between your program and future releases of the Apple IIGS firmware.

Appendix E describes the soft switches that control various aspects of system behavior. These switch locations and contents are provided for reference only. The contents of the switches should be modified only by system tools.

Appendix F lists the disassembler/mini-assembler opcodes. These will be useful to the machine-language programmer who uses the system Monitor to enter small programs for quick tests.

Appendix G describes the Control Panel options and defaults.

Appendix H describes the contents of memory banks $E0 and $E1.

A glossary follows the appendixes.

Chapter 1

Overview

This chapter gives a brief overview of the Apple IIGS firmware and how it relates to the rest of the system software. The Apple IIGS firmware is composed of various routines that are stored in the system's read-only memory (ROM). The Apple IIGS firmware routines provide the means to adapt and control the Apple IIGS system.

Routines for the following Apple IIGS firmware are covered in this manual:

- system Monitor firmware
- video firmware (I/O routines)
- serial-port firmware (for character-at-a-time I/O)
- Disk II support (slot 6 support)
- SmartPort firmware (for block device I/O)
- interrupt-handler firmware
- Apple DeskTop Bus (ADB) microcontroller
- mouse firmware

A word about other Apple IIGS firmware

Not all Apple IIGS firmware is discussed in this manual. The Apple IIGS ROM contains other firmware, important enough to warrant separate manuals: the Apple IIGS Toolbox (described in detail in the *Apple IIGS Toolbox Reference*), Applesoft BASIC (described in the *Applesoft BASIC Reference*), and the AppleTalk® Personal Network (described in *Inside AppleTalk*).

Apple IIGS Toolbox

The Apple IIGS Toolbox provides a means of easily constructing application programs without necessarily using the firmware routines described in this manual. Programs that you construct using the tools will conform to the *Apple Human Interface Guidelines*. By offering a common set of routines that every application can call to implement the user interface, the tools not only ensure familiarity and consistency for the user but also help to reduce the application's code size and development time.

Applesoft BASIC

The Apple IIGS also has Applesoft BASIC in ROM so that you can create and run your own programs in BASIC.

AppleTalk

AppleTalk is a local-area network that allows communication and resource sharing by up to 32 computers, disks, printers, modems, and other peripheral devices. AppleTalk consists of communication hardware and a set of communication protocols. This hardware/software package, together with the computers, cables and connectors, shared resource managers (servers), and specialized application software, functions in three major configurations: as a small-area interconnecting system, as a tributary to a larger network, and as a peripheral bus between Apple computers and their dedicated peripheral devices.

Diagnostic routines

The system diagnostic routines are manufacturing test routines. No external entry points are defined for system diagnostic routines at this time. Thus, diagnostic routines are not documented in this manual.

The role of firmware in the Apple IIGS system

The firmware is that set of low-level routines that provides programmers with an interface to the system hardware. The firmware, in turn, controls the display, the mouse, serial input/output (I/O), and disk drives. Firmware programs, such as the Monitor and the Control Panel, work directly with the system memory.

Traditionally, programmers have controlled hardware directly through their application programs, bypassing any system firmware. The disadvantage of this approach is that the programmer has to do a lot more work. More important, bypassing the firmware increases the likelihood that the resulting program will be incompatible either with other programs or with future versions of the computer. By using the firmware interface, a programmer can maintain compatibility with current and future releases of the system.

For most of the functions that the firmware entry points perform, there are equivalent functions provided in the toolbox. The toolbox routines, in addition to performing like functions, also save and restore system registers when they are called. Read Chapter 2, "Notes for Programmers," for more details about system register usage.

Levels of program operation

You can think of the different levels of program operation on the Apple IIGS as a hierarchy, with a hardware layer at the bottom, firmware layers in the middle, and the application at the top. Figure 1-1 shows a hierarchy of command levels; in general, higher-level components call on lower-level ones. (The levels are separated by lines; the hardware components have heavy outlines.)

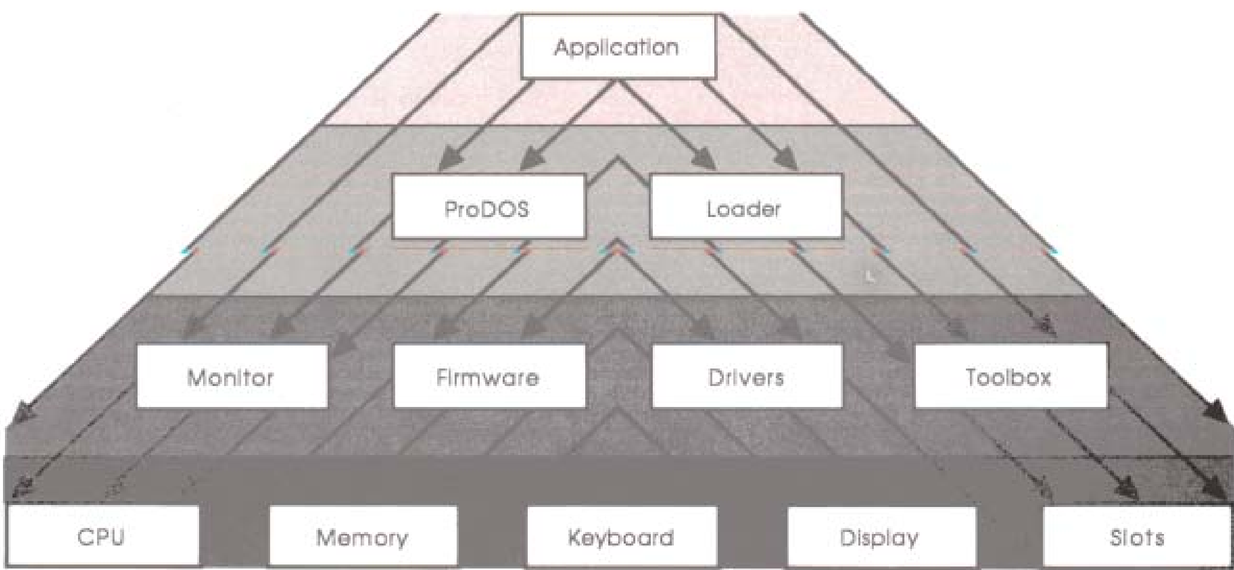


Figure 1-1
Levels of program operation

Apple IIGS firmware

The following sections provide an overview of the Apple IIGS firmware described in this manual.

System Monitor firmware

The system Monitor firmware is a set of routines that you can use to operate the computer at the machine-language level. You can examine and change memory locations, examine and change registers, call system routines, and assemble and disassemble machine-language programs using the system Monitor firmware.

Video firmware

Video firmware allows you to manipulate the screen in low-resolution mode and text mode through your application programs and from the keyboard. Communication between the keyboard and the video screen is controlled by firmware subroutines, escape codes, and control characters. The video firmware provides on-screen editing, keyboard input, output to the screen, and cursor-control facilities.

Serial-port firmware

The Apple IIGS serial-port firmware provides a means to allow serial communication with external devices, such as printers and modems. The serial-port firmware provides support for such options as hardware and software handshaking and background printing. There are two serial ports, either of which can be configured as a printer port or a modem port.

Disk II support

The Apple IIGS Disk II firmware is a disk-support subsystem. It uses a built-in Integrated Woz Machine (IWM) chip and accommodates Disk II (DuoDisk® or UniDisk™) drives. Slot 6 is the standard Disk II support slot. The firmware that communicates with the IWM at boot time provides support for booting Disk II–based software. Other handling of Disk II devices is a function of whichever disk operating system is booted.

SmartPort firmware

Disk II devices are directly manipulated by slot 6 control hardware. Intelligent devices, by contrast, are not directly manipulated by hardware, but rather are controlled by software-driven command streams. Such devices are labeled *intelligent devices* because they have their own controllers, which can interpret these command streams. The SmartPort firmware is a set of assembly-language routines that permit you to attach one or more intelligent devices to the external disk port of the Apple IIGS system. Using the SmartPort firmware, you can control these devices through SmartPort calls, such as Open, Close, Format, ReadBlock, and WriteBlock.

Interrupt-handler firmware

System interrupts halt the execution of a program or the performance of a function or feature. The system contains built-in interrupt-handler firmware, a user's interrupt-handler entry point, and a means to notify the user when an interrupt occurs.

Apple DeskTop Bus microcontroller

The Apple DeskTop Bus (ADB) microcontroller is used to receive information from peripheral units attached to the Apple DeskTop Bus. The ADB microcontroller polls the internal keyboard, sensing key-up and key-down events as well as control keys, and optionally buffers keystrokes for later access by the 65C816. In addition, the ADB microcontroller acts as host for ADB peripheral devices, such as the detachable keyboard and mouse. The ADB microcontroller has its own built-in set of instructions, including Talk, Listen, SendReset, and Flush.

Mouse firmware

The Apple IIGS mouse firmware supplies the communication protocol for sensing the current status of the mouse. The mouse firmware tracks mouse-device position data and button status and provides entry points for assembly-language control.

Chapter 2

Notes for Programmers

This chapter contains information that will be useful to the experienced 6502 programmer as well as someone just beginning to use the Apple IIGS computer.

The Apple IIGS has many new features not found in previous Apple computers. Programs written for the Apple IIc or the Apple IIe will run on the Apple IIGS, but do not take advantage of these new features.

Among the new features of the Apple IIGS is a new set of registers, pseudoregisters, and flags, collectively known as the **environment.** Before you change the environment for the Apple IIGS system, read the following sections, which outline these new features.

Introduction to the Apple IIGS

The Apple IIGS personal computer is a new Apple II with many high-performance features. Highlights include

- □ more powerful microprocessor with faster operation and larger memory
- □ high-resolution RGB video for Super Hi-Res color graphics
- □ multivoice digital sound synthesizer
- □ detached keyboard with Apple DeskTop Bus connector
- □ built-in I/O: clock, disk port, and serial ports with AppleTalk interface
- □ compatible slots and game I/O connectors

This list includes only the main features of the Apple IIGS. For a comprehensive list of features, refer to the *Technical Introduction to the Apple IIGS.*

Microprocessor features

The microprocessor in the Apple IIGS is a 65C816, a 16-bit design based on the 6502. Among the features of the 65C816 are

- □ ability to emulate a 6502 8-bit microprocessor
- □ 16-bit accumulator and index registers
- □ relocatable stack and zero page (direct page)
- □ 24-bit internal address bus for 16-megabyte memory space

Microprocessor modes

The 65C816 microprocessor can operate in two different modes: **native mode,** with all of its new features, and 6502 **emulation mode,** for running programs written for 8-bit Apple II computers.

If you are using emulation mode extensively, you will be using the firmware calls described in this manual. If you are using native mode, you probably will want to use the equivalent toolbox calls instead of directly calling the firmware. The toolbox calls save and restore the environment for you.

Execution speeds

The microprocessor in the Apple IIGS can operate at either of two clock speeds: the standard Apple II speed of 1 MHz and the faster speed of 2.8 MHz. When running programs in RAM, the Apple IIGS uses a few clock cycles for refreshing memory, making the effective processing speed about 2.5 MHz. System firmware, running in ROM, runs at the full 2.8 MHz.

Expanded memory

Thanks to the 24-bit addresses of the 65C816, the Apple IIGS has a memory space totaling 16 megabytes. Of this total, up to 8 megabytes of memory are available for RAM expansion, and 1 megabyte is available for ROM expansion. For additional information about memory, read the *Technical Introduction to the Apple IIGS.*

The minimum memory in the Apple IIGS is 256K. Programs written for the Apple IIGS—that is, programs that run the 65C816 microprocessor in native mode, thereby gaining the ability to address more than 128K of memory—can use up to about 176K of the 256K. The rest is reserved for displays and for use by the system firmware.

The Apple IIGS also has a special card slot dedicated to memory expansion. All of the RAM on a memory-expansion card is available for Apple IIGS application programs that call the Memory Manager. Expansion memory is contiguous: Its address space extends without a break through all of the RAM on the card. Expansion RAM on the Apple IIGS is not limited to use as data storage; program code can run in any part of RAM.

Super Hi-Res display

In addition to all the video display modes of the Apple IIc and Apple IIe, the Apple IIGS has two new Super Hi-Res display modes that look much clearer than standard Hi-Res and Double Hi-Res. Super Hi-Res is also easier to program because it maps entire bytes onto the screen, instead of 7 bits, and its memory map is linear.

Used with an analog RGB video monitor, the new display modes produce high-quality, high-resolution color graphics. Table 2-1 lists the specifications of the two new graphics display modes.

Table 2-1
Super Hi-Res graphics modes

Mode	Resolution		Bits per pixel	Colors per line	Colors on screen	Colors possible
	Horiz.	Vert.				
320	320	200	4	16	256	4096
640	640	200	2	16*	256*	4096

* Different pixels in 640 mode use different parts of the palette.

❖ *Note:* **Pixel** is short for *picture element.* A pixel corresponds to the smallest dot you can draw on the screen.

Each dot on the Super Hi-Res screen corresponds to a pixel. Each pixel has either a 2-bit (640 mode) or a 4-bit (320 mode) value associated with it. The pixel values select colors from programmable color tables called *palettes.* A palette consists of 16 entries, and each entry is a 12-bit value specifying one of 4096 possible colors.

In 320 mode, each pixel consists of 4 bits, so it can select any one of the 16 colors in a palette. In 640 mode, each byte holds four 2-bit pixels. The 16 colors in the palette are divided into four groups of 4 colors each, and successive pixels select from successive groups of 4 colors. Thus, even though a given pixel in 640 mode can be one of only 4 colors, different pixels in a line can take on any of the 16 colors in a palette.

To further increase the number of colors available on the display, there can be as many as 16 different palettes in use at the same time, allowing as many as 256 different colors on the screen.

Digital sound synthesizer

In addition to the single-bit sound output found in other computers in the Apple II family, the Apple IIGS has a new digital sampling sound system built around a special-purpose synthesizer IC called the **Digital Oscillator Chip,** or DOC for short. Using the DOC, the Apple IIGS can produce 15-voice music and other complex sounds without tying up its main processor. Refer to the *Apple IIGS Hardware Reference* for details about the sound system and the DOC.

Detached keyboard with Apple DeskTop Bus

The new detached keyboard includes cursor keys and a numeric keypad. The Apple DeskTop Bus, which supports the keyboard and the Apple mouse, can also handle other input devices such as joysticks and graphics tablets.

Built-in I/O

Like the Apple IIc, the Apple IIGS has two built-in disk ports and two serial I/O ports. Programs can use the built-in ports and peripheral cards in slots. The built-in AppleTalk interface uses one of the serial ports.

The Apple IIGS also has a built-in clock-calendar with a battery for continuous operation.

Compatible slots and game I/O connectors

In addition to the memory-expansion slot, the Apple IIGS has seven I/O expansion slots like those on the Apple IIe. Most peripheral cards designed for the Apple II Plus and the Apple IIe will work in the Apple IIGS slots. The Apple IIGS also has game I/O connectors for existing game hardware.

Environment for the firmware routines

Many useful subroutines are listed in Appendix C, "Firmware Entry Points in Bank $00." All of these routines have one thing in common: To use them, the processor must be set up to look and act exactly like a 6502 in all respects. You must therefore set the operating environment to cause this transformation to happen.

Important

This section contains the specific details about setting and restoring the environment before calling and after returning from calling the firmware routines. You must follow these requirements exactly, or your program will fail.

The specific operating environment requirements for all these routines are as follows:

- d bit = 0 (decimal-mode bit)
- e bit = 1 (emulation-mode bit)
- D register = $0000 (direct-page register)
- DBR register = $00 (data bank register, called B in Chapter 3)
- PBR register = $00 (program bank register, called K in Chapter 3)
- S register = $01xx (stack pointer)

❖ *Note:* If you make tools calls instead of using the firmware directly, you will not have to worry about the operating environment. The tool calls handle the environment for you.

Setting up the system

To correctly prepare the system for calling the firmware routines, you must take several steps:

- Save your environment.
- Get into bank $00: JSL (jump to subroutine long) to a routine in bank $00.
- Set the D register to $0000.
- Set the DBR to $00.
- Save the value of the native-mode stack pointer, and set the stack pointer to the value of the emulation-mode stack pointer.
- Select emulation mode: set the e bit to 1.

These steps make the 65C816 appear to be a 6502 microprocessor operating in its normal environment. Now you can set up the machine registers with the parameters as required by the particular firmware routine and execute a JSR (jump to subroutine). These steps are explained in the sections that follow.

Save your environment

The environment is the complete set of machine registers and flags that your program uses. Besides machine registers, the environment includes such things as processor speed, read-only memory (ROM) bank, language-card bank, and random-access memory (RAM) shadowing.

When you run the various firmware routines, the system will use the machine registers for its own purposes. If you depend on a particular register having a specific value when you finally return to your own code, then save that register's contents on your native-mode stack or wherever else you wish so that you can restore the register's contents before you return to your other program code. To determine which registers each firmware routine uses or affects, see Appendix C, "Firmware Entry Points in Bank $00."

Get into bank $00

If you attempt to run the 65C816 in emulation mode in any bank other than bank $00, no interrupt processing can take place. You enter program bank $00 by executing a JSL (jump to subroutine long) to someplace in bank $00 (if you are not already there), where the next steps are performed. This JSL sets the program bank register (K) to $00, fulfilling that part of the firmware routine requirement. If you did not save your environment before entering bank $00, now would be an equally good time to do so.

Set the D register to $0000

A 6502 expects its **zero page** (called the **direct page** for the 65C816 when operating in native mode) to exist in the microprocessor address range of $00 to $FF. When the D register is set to 0, the zero page gets positioned correctly for a 6502.

Set the DBR to $00

The DBR is the upper 8 bits of the 24-bit data address. The DBR must have a value of $00 for the firmware routines to function.

Save the value of the native-mode stack pointer

When you switch to emulation mode, the upper 8 bits of your stack pointer will be lost. Thus, this value must be saved somewhere so that it can be restored to its original value on exit from this routine. The most common technique is to save the value of the entire native-mode stack pointer on the emulation-mode stack.

❖ *Note:* The main and auxiliary stack-page switches cannot be used in native mode. Thus, when switching to emulation mode, you must use the main stack.

The routine that follows saves the native-mode stack pointer and correctly sets the values for the direct-page register and the data bank register. If your program requires other values for the direct-page and data bank registers, save these environment variables (as well as other register values in your environment) so that you can restore the values after returning from the firmware routine that you call. The EMULSTACK routine can be appended to the beginning of your own firmware calling sequence. A corresponding routine to restore the native-mode stack pointer is given in the section "Returning to Native Mode" later in this chapter.

```
                                  ;Before entry, save YOUR environment!
EMULSTACK   EQU  $010100          ;Emulation stack pointer is saved here
TOEMUL      REP  #$30             ;16-bit m and x
            TSC                   ;Temporary save of native-mode stack pointer
            TAX
            SEP  #$20             ;8-bit m
            XBA                   ;Get stack pointer page
            DEC  A                ;Is stack already in page 1?
            BEQ  ALREADYPG1       ;If so, don't get emulation stack pointer
            LDA  #$01             ;Set stack page to $01
            XBA
            LDA  EMULSTACK        ;Get emulation stack pointer
            TCS                   ;Set emulation stack pointer
ALREADYPG1  PHX                   ;Save native-mode stack pointer
            SEC                   ;Emulation mode
            XCE                   ;Set emulation mode
            PEA  $0000
            PLD                   ;Set direct-page register to $0000
            LDA  #0
            PHA
            PLB                   ;Set data bank register to $00
                                  ;Here continue with YOUR processing
```

Select emulation mode

Setting the e bit to 1 puts the 65C816 into emulation mode and automatically sets the m and x processor status bits to 1. The x bit forces the X and Y registers to be treated as only 8 bits wide. The m bit forces the accumulator to be treated as only 8 bits wide. This step also affects the size of the stack and the contents of the stack register. Specifically, the value of the upper 8 bits of the stack pointer is forced to a value of hexadecimal $01 (the same as the 6502). While you are in emulation mode, these upper 8 bits never change. Thus, the size of the stack is restricted to 256 bytes.

Now you can set up the machine registers as required by the particular firmware routine and JSR.

Returning to native mode

To return to native mode, you must perform a set of steps complementary to the preceding steps that caused your program to enter emulation mode in the first place:

- □ Restore the native-mode stack pointer.
- □ Restore your environment (if you are within the bank $00 entry routine).

Then you can execute an RTL (return from subroutine long) to your point of origin (assuming that you performed a JSL to enter this code in the first place). These two return steps are explained in detail in the next two sections.

Restore the native-mode stack pointer

Return to native mode. The following example is the complement to the preceding example that saved the native-mode stack pointer. Notice that this routine also returns the processor to native mode (it sets the e bit to 0 and then sets the m and x bits to 0).

```
PHP                        ;Preserve firmware's c (carry) status
CLC                        ;Set native mode
XCE                        ;It's still in 8-bit
PLP                        ;Restore the carry flag
REP        #$30            ;Set 16-bit
PLX                        ;Get native stack pointer from emulation stack
TXS                        ;Set the native-mode stack pointer
                           ;Now restore the rest of your environment!
```

Restore your environment

Restore all of your registers and flags to the values that your program expects to find on return.

Assuming that you used a JSL in the code that saved your environment and your native-mode stack pointer, you can now perform an RTL and resume execution of your program.

Other requirements for emulation-mode code

The preceding example showed how to call firmware routines and specified that the processor must be in emulation mode, running in bank $00, to call the firmware routines. There may be other times when you want to use emulation mode from banks other than bank $00, but you must observe other specific requirements.

When you run emulation-mode code in a bank other than bank $00, interrupts *must* be disabled.

❖ *Note:* For AppleTalk applications, you must be sure that interrupts are enabled for at least 20 milliseconds out of every 1.1 seconds. For applications using the tick counter, interrupts must not be disabled for longer than 16.67 milliseconds or ticks will be lost.

When you are in a bank other than bank $00 with interrupts disabled, if you mix 6502 and 65C816 instructions, the 65C816 instructions will still function as documented. But note that all 6502-equivalent instructions behave the same as a 6502 regarding direct-page and stack-page wrapping. The new 65C816 instructions manipulate the stack and direct page, but do *not* wrap on a page boundary. Thus, you must exercise care when using these new stack- or direct-page instructions.

Cautions about changing the environment

If you write your own subroutines (or programs) that change some part of the operating environment, be sure that your code, at exit, puts things back the way it found them at entry. This is especially true of stack- and zero-page changes, data-bank-register changes, m, e, and x changes, speed-register changes, ROM-bank changes, and language-card changes.

Stack and direct page

For Apple II programs, the stack and the direct page (called the *zero page* for a 6502) must be in their proper 6502 locations and the stack must be 256 bytes long. For Apple IIGS programs, stack size and stack- and direct-page locations are at the discretion of the application. (Call the Memory Manager to obtain a new zero-page area).

When you are in native mode, you can locate the stack anywhere within bank $00. If the stack is located in memory at other than page 1 and the processor is switched to emulation mode, the upper half of the stack pointer will be lost (set to $01). When the processor is switched back to native mode, the upper half of the stack pointer will remain set to page $01. To avoid losing the native-mode stack pointer when switching to emulation mode, you must temporarily save the stack pointer so it can be restored. Sample code for saving and restoring the native-mode stack value is shown in the examples.

Data bank registers and e, m, and x flags

If your subroutine changes the contents of the data bank register or the e, m, and x flags, you should restore them to their original values. These registers affect not only the locations to which the index registers X and Y point and the length of the A, X, and Y registers; the contents of these registers also affect how the processor interprets its instructions. One can easily imagine an incorrect flag or register value causing a perfectly good program to fail.

Speed- and Shadow-register changes

Changing any of the bits in the Speed or Shadow register (see Chapter 3, "System Monitor Firmware") also affects how the system runs. (The Shadow-register bits of interest and the speed-change bit are all accessible through the pseudoregister called *Quagmire*. For assembly-language programming, you access these registers directly. See the *Apple IIGS Hardware Reference* for more information.)

Language-card changes

If you change the active bank of the language card without restoring it on exit from your code, you again risk ruining another programmer's code. For example, the other programmer might have executed a JSR or JSL out of some code in a ROM bank or a particular bank of the language card. The return address of that routine is on the stack and points to the return address within that same bank of ROM or the language card. If your routine changes banks without restoring them to the original values upon exit, the system will fail.

General information

This section contains other general information useful in creating 65C816 programs for the Apple IIGS.

Apple IIGS interrupts

The Apple IIGS firmware provides improved interrupt support, very much like the enhanced Apple IIe interrupt support. Neither machine disables interrupts for extended periods.

The main purpose of the interrupt handler is to support interrupts in any memory configuration. This is done by saving the machine's state at the time of the interrupt, placing the Apple IIGS in a standard memory configuration before calling your program's interrupt handler, and then restoring the original state when your program's interrupt handler is finished. (See Chapter 8, "Interrupt-Handler Firmware," for more information.)

Boot/scan sequence

The boot/scan sequence is initiated by selecting Startup: Scan from the Control Panel Slots menu. When the selection is made, the Apple IIGS starts at slot 7 and tests each slot for a boot device; the first device found is booted. The Apple IIGS starts its scan at the slot selected, ignoring all slots with a higher number, and works down to slot 1. If no boot devices are in the slots, the screen displays the message shown in Figure 2-1 (the apple moves back and forth across the screen).

Check Startup Device

Figure 2-1
Boot-failure screen

If slot 7 is enabled for an external device, the scan will proceed as just described. However, if slot 7 is set to AppleTalk and if the startup slot is set to slot 7, the firmware will try to boot AppleTalk. If RAM Disk or ROM Disk is selected, the SmartPort firmware will be activated and the system will attempt to boot from the RAM disk or ROM disk (see Chapter 7, "SmartPort Firmware").

Program bank register

The 65816 program bank register wraps within a 64K bank boundary. Data retrieval and storage, however, do not wrap within a 64K bank. This means that a program that executes at the top of a bank continues to execute at the bottom of the same bank, even between *opcode* and *operand* within a single instruction. Further, data retrieval and storage at the top of a bank simply roll over into the bottom of the next bank and continue as if no bank had been crossed. This same operation also occurs with indexed instructions.

Important

You must exercise care when writing code that deals directly with state-dependent hardware. The cycle-by-cycle operations of the 65C816 emulation mode and the 65C816 native mode differ. This behavior has to do with indexed instructions. In one mode, a false read occurs at a given cycle, and in the other mode, a false write occurs. This difference can cause problems if soft switches and hardware expect one operation and get another.

Exchanging the B and A registers, XBA

The A register (called the C register in native mode) is a 16-bit register used in both native and emulation modes. In native mode, all 16 bits are used; in emulation mode, 8 bits are used for the A register and 8 bits are used for the B register (see Figure 2-2).

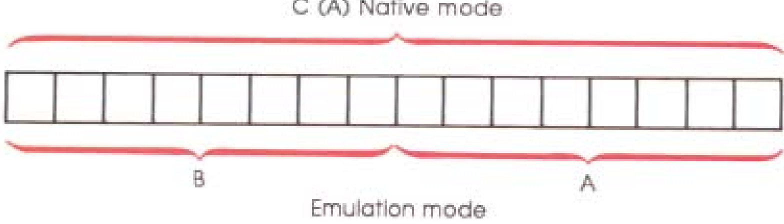


Figure 2-2
Accumulator for emulation and native modes

Some programmers with 6502 experience might see the XBA instruction as a quick way to save the current contents of the A register while running in emulation mode. Then they might assume that it is appropriate to jump to system routines (that have to be executed from emulation mode anyway) and return, restoring the A register from B by another XBA. However, the contents of the B register (the old 8-bit accumulator value) will not be valid on return from any firmware routine. Thus, do not transfer control to any system code prior to restoring the A register with the following XBA. If you do, it is at your own risk. Although current documentation for the firmware entry points occasionally may show that the contents of the B register are preserved, this will not necessarily hold true for later releases of the firmware.

For example, the following code works in 8-bit mode:

```
XBA            ;Preserve A
LDA FLAG       ;Do something with A
LSR            ;Move LSB to carry
XBA            ;Restore A
```

The following code does not work:

```
XBA            ;Preserve A
LDA #A
JSR COUT       ;Control is transferred
XBA            ;Restore A
```

The `A` in the first line is not the same as the `A` in the fourth line.

Chapter 3

System Monitor Firmware

This chapter describes the Apple IIGS system Monitor firmware, a low-level, command-driven program that lets you examine the machine state as well as create and test small machine-language programs. A professional developer will likely use a sophisticated assembler and debugger in addition to the system Monitor firmware.

Note that when you use the Monitor to write machine-language programs, you can use the Monitor entry points listed in Appendix C, "Firmware Entry Points in Bank $00," to make your job easier. Also, if you use the disassembler, you will be interested in the table of disassembler opcodes in Appendix F, "Disassembler/Mini-Assembler Opcodes."

The system Monitor firmware is a program that you can use to create and test your own machine-language programs for the Apple IIGS. From the Monitor, you can create programs that utilize various system-resident subroutines (a summary of which is contained in Appendix C, "Firmware Entry Points in Bank $00"). When you create your own programs or use the Monitor to examine programs that others have created, various features of the Monitor firmware assist you in your task.

The Apple IIGS Monitor provides commands that

- manipulate memory by examining it; by entering changes in either ASCII or hexadecimal form; by moving, comparing, or filling blocks of memory; and by searching for specified patterns
- view and change the execution environment (microprocessor registers and flags)
- execute programs from the Monitor
- step through and trace program execution (hooks only; no code in current ROM)
- perform miscellaneous tasks such as setting the display to inverse or normal video, displaying or setting the time and date, redirecting input and output, performing hexadecimal arithmetic, returning to BASIC via cold or warm start
- invoke the mini-assembler
- invoke the disassembler

Invoking the Monitor

The system Monitor resides in read-only memory (ROM) beginning at location $FF69, or –151. To invoke the Monitor, you issue a Call statement to this location from the keyboard or from a BASIC program. When the Monitor is running, its prompt character (*) appears on the left side of the display screen, followed by a cursor. To use the Monitor, type

`Call -151` Return

The prompt character and the cursor (a flashing blank space) appear:

```
*
```

Monitor command syntax

You enter all Monitor instructions in the same format: Type a line on the keyboard and press Return. The Monitor accepts the line using the I/O subroutine GETLN. A Monitor instruction can be up to 255 characters, followed by a carriage return. (GETLN is described in Chapter 4, "Video Firmware.")

A Monitor command can include four kinds of information: memory-bank number, addresses, data values, and command characters. You type addresses, memory-bank numbers, and data values in **hexadecimal** notation.

The microprocessor in Apple II computers prior to the Apple IIGS could address memory only in an address range from 0 to 65,535. The Apple IIGS, on the other hand, can address up to 256 banks of 65,536 memory locations each. Thus, there is a need for a memory-bank address qualifier for the Monitor commands. You will see the complete address represented as {*bank*/*address*}, where *bank* is to be specified as two hexadecimal digits and *address* as four hexadecimal digits.

When the command you type calls for an address, the Monitor accepts any group of hexadecimal digits, automatically providing leading zeros to fill out the width of the field of digits.

Monitor command types

There are two distinct types of Monitor commands: commands that perform an operation (such as examining or filling memory) and commands that change a register value.

For commands that perform an operation, each command you type consists of one command character, usually the first letter of the command name. When the command is a letter, it can be either uppercase or lowercase. The Monitor recognizes 46 different commands. Some of them are punctuation marks, some are letters, and some are control characters.

❖ *Note:* Although the Monitor recognizes and interprets control characters typed on an input line, control characters do not appear on the screen.

For commands that affect the contents of a register, each command you type consists of a value and a register name. For register names, the Apple IIGS Monitor does require that the register name be entered using the proper case (uppercase or lowercase). The syntax of a register-modifying command is

{*value*}={*register*}

When you use a register-display command, the appropriate case for you to use to modify the register contents is shown in the display for each register. Be certain to note whether the register name is uppercase or lowercase and to use the correct case when setting a register value.

Table 3-1 lists the Monitor commands and their syntax grouped by type. In Table 3-1 and in the rest of this chapter, the command formats often specify addresses from which data is obtained or to which data is sent. The source and target addresses take the form

bank/address

where ***bank*** is an optional bank number (one or two hexadecimal digits) and ***address*** is the address (one to four hexadecimal digits). The bank number, if present, is separated from the address by a forward slash (/) character. To make the command formats more understandable, several terms are introduced here, each of which may be used in lieu of ***bank/address***. Note that each of these terms uses exactly the same format: an optional bank number and the address. The purpose of these substitute forms is to make the command formats (especially within tables) easier to understand at a quick glance.

The following terms may be used:

destination	An address (with optional bank) that serves as a data destination
from_address	An address (with optional bank) at one end of a range of addresses
to_address	An address (with optional bank) at the other end of a range of addresses
start_address	An address (with optional bank) at which the Monitor will start an operation
val	An 8-bit (1-byte) value specified as two hexadecimal digits
val16	A 16-bit (2-byte) value specified as four hexadecimal digits
val64	A value expressed as up to eight hexadecimal digits
val10	A value expressed as decimal digits
mm/dd/yy	Three 8-bit values separated by forward slashes
hh:mm:ss	Three 8-bit values separated by colons

Table 3-1
Monitor commands grouped by type

Command type	Command format
Viewing and modifying memory	
Display single memory location	{*from_address*}
Display multiple memory locations	{*from_address*} . {*to_address*}
Terminate memory-range display	Control-X
Modify consecutive memory	{*destination*} : {*val*} {*val*} {"*literal ASCII*"} {'*flip ASCII*'} {*val*}
Move data in memory	{*destination*}<{*from_address*} . {*to_address*}M
Verify memory contents	{*destination*}<{*from_address*} . {*to_address*}V
Fill memory (zap)	{*val*}<{*from_address*} . {*to_address*}Z
Pattern search (specified in four ways; any or all forms may be combined in a single search request)	\{*val*}\<{*from_address*} . {*to_address*}P \{'*123t*'}\<{*from_address*} . {*to_address*}P \{"*literal ASCII*"}\<{*from_address*} . {*to_address*}P \{*val16*}\<{*from_address*} . {*to_address*}P
Viewing and modifying registers	
Examine registers	Control-E
Modify accumulator	{*val16*}=A
Modify X register	{*val16*}=X
Modify Y register	{*val16*}=Y
Modify D register	{*val16*}=D
Modify DBR register (bank)	{*val*}=B
Modify program bank register	{*val*}=K
Modify stack pointer	{*val16*}=S
Modify processor status	{*val*}=P
Modify machine-state register	{*val*}=M
Modify Quagmire register	{*val*}=Q
Modify 16/8-bit accumulator mode	{*val*}=m
Modify 16/8-bit index mode	{*val*}=x
Modify native/emulation mode	{*val*}=e
Modify language-card bank	{*val*}=L
Modify ASCII filter mask	{*val*}=F

(continued)

Table 3-1 (continued)
Monitor commands grouped by type

Command type	Command format
Miscellaneous	
Begin inverse video	I
Begin normal video	N
Change time and date	=T=*mm/dd/yy hh:mm:ss*
Display time and date	=T
Redirect input links	{*slot*} Control-K
Redirect output links	{*slot*} Control-P
Change screen display to text	Control-T
Change cursor	Control-^ {*new_cursor_character*}
Convert decimal to hexadecimal	={*val10*}
Convert hexadecimal to decimal	{*val64*}=
Perform hexadecimal math	
Add	{*val64*}+{*val64*}
Subtract	{*val64*}-{*val64*}
Multiply	{*val64*}*{*val64*}
Divide	{*val64*}_{*val64*}
Jump to cold-start BASIC	Control-B
Jump to warm-start BASIC	Control-C
Jump to user vector	Control-Y
Quit Monitor	Q
Program execution and debugging	
Go (begin) program in bank $00	{*start_address*}G
Execute from any memory bank	{*start_address*}X
Restore registers and flags	Control-R
Resume execution	{*start_address*}R
Perform a program step	{*start_address*}S
Perform a program trace	{*start_address*}T
Disassemble (list)	{*start_address*}L
Enter mini-assembler	!

Monitor memory commands

The Monitor commands that directly affect memory are discussed in this section. These include commands to examine and change memory locations, search for specific combinations of memory contents, change memory contents individually or in blocks, and compare memory blocks. The Monitor presents memory dumps in both ASCII and hexadecimal formats. You can use either notation to enter your requests for changes to memory.

When you use the Monitor to examine and change the contents of memory, the Monitor keeps track of the address of the last location whose value you inquired about (called the **last-opened location**) and the address of the location that is to have its value changed next (called the **next-changeable location**). In addition, once you have specified a bank number in one of your instructions, the Monitor continues to use that bank number with all other instructions until you explicitly change it.

In the paragraphs that follow, the memory-contents displays are based on what you would see if you were using the display in 80-column mode. When in 40-column mode, the Apple IIGS Monitor dumps memory 8 bytes per line. When in 80-column mode, the Apple IIGS Monitor dumps memory 16 bytes per line.

Table 3-2 lists the Monitor memory commands.

Table 3-2
Commands for viewing and modifying memory

Command type	Command format
Display single memory location	{*from_address*}
Display multiple memory locations	{*from_address*} . {*to_address*}
Terminate memory-range display	Control-X
Modify consecutive memory	{*destination*} : {*val*} {*val*} {"*literal ASCII*"} {'*flip ASCII*'} {*val*}
Move data in memory	{*destination*}<{*from_address*} . {*to_address*}M
Verify memory contents	{*destination*}<{*from_address*} . {*to_address*}V
Fill memory (zap)	{*val*}<{*from_address*} . {*to_address*}Z
Pattern search (specified in four ways; any or all forms may be combined in a single search request)	\{*val*}\<{*from_address*} . {*to_address*}P
	\{'*123t*'}\<{*from_address*} . {*to_address*}P
	\{"*literal ASCII*"}\<{*from_address*} . {*to_address*}P
	\{*val16*}\<{*from_address*} . {*to_address*}P

Examining memory

The syntax required to display a single memory location is

{*bank*/*address*} Return

If the Monitor is already examining the bank desired, you don't have to include the bank number in the instruction. Simply type the address and press Return. However, if you're not sure which bank the Monitor is in, include the bank number as shown in the example. The Monitor responds with the bank and address you typed (*bank*/*address*), a colon, and the hexadecimal contents of the location. For example, to examine memory location hexadecimal $1000, next to the Monitor prompt (*) type

`*00/1000` Return

The bank and address are displayed as well as the contents of address $1000:

```
00/1000:20-
```

❖ *Note:* Dollar signs ($) preceding addresses that appear in running text signify that the addresses are in hexadecimal notation; however, dollar signs are ignored by the Monitor and must be omitted when typing instructions. If location $1000 had contained ASCII code, the ASCII equivalent would be displayed on the far right of the screen, as the following example shows:

`*1000` Return

(Notice that the bank address was not entered because you know that you are in bank $00.) The result is

```
00/1000:41-A
```

❖ *Note:* ASCII codes are decoded in the rightmost 8 spaces of your display. Printable ASCII characters are displayed as normal characters; nonprintable characters are displayed as periods (.). If you are using the Monitor in 80-column mode, the ASCII characters will take up the rightmost 16 spaces instead of 8, and 16 sets of hexadecimal digit pairs corresponding to the byte values stored in the displayed memory range.

When you change the contents of memory, the Monitor saves the address of the last location in which you changed the contents and the address of the next location to be changed—in other words, the last-opened location and the next-changeable location.

Examining consecutive memory locations

You may want to examine a block of memory locations, such as from $1000 to $1007. Simply type the starting address, a period, and the ending address and then press Return:

```
*1000.1007 Return
```

The contents of the memory locations are displayed as follows:

```
00/1000:41 42 43 44 45 55 00 00 -ABCDEU..
*
```

If you type a period (.) followed by an address and then press Return, the Monitor displays a memory dump: the data values stored at all the memory locations from the one following the last-opened location to the location whose address you typed following the period. The Monitor saves the last location displayed as both the last-opened location and the next-changeable location. In these examples, the amount of data displayed by the Monitor depends on the difference between the address of the last-opened location and the address after the period.

```
00/1000:41-A
*.100B Return
00/1001:41 42 43 44 45 55 00 00 -BCDEU..
00/1008:51 52 53 54 -PQRS
*
```

When the Monitor performs a memory dump, it starts at the location immediately following the last-opened location and displays that address and the data value stored there. It then displays the values of successive locations up to and including the location whose address you typed, but shows only up to 8 (or 16) values on a line. When it reaches a location whose address is a multiple of 8 (or 16), that is, one whose address ends with an 8 (or if 16, an address that ends with a 0), it displays that address as the beginning of a new line and then continues displaying more values.

If you have selected a large memory range to display and you wish to halt the display and resume entering other Monitor commands, press Control-X. This terminates the memory-range display.

After the Monitor has displayed the value at the location whose address you specified in the command, it stops the memory dump and sets that location as both the last-opened location and the next-changeable location. If the address specified in the input line is less than the address of the last-opened location, the Monitor displays only the address and the value of the location following the last-opened location.

Changing memory contents

The previous section showed you how to display the values stored in the Apple IIGS memory system; this section shows you how to change those values. You can change any location in RAM and you can also change the soft switches and output devices by changing the contents of the memory locations assigned to them.

Warning

Use these commands carefully. If you change the contents of memory in any area used by the Apple IIGS firmware or Applesoft, you may lose programs or data stored in memory. You can find a map showing the memory use by various parts of the system software in the *Apple IIGS Hardware Reference*.

Changing one byte

Previous commands kept track of the next-changeable memory location; other memory commands make use of that location. In the next example, you open location $1000 and type a colon (:) followed by a value:

```
*1000 Return
00/1000:50 -P
*:54 Return
```

This entry changes the contents of the opened location to the value you requested. To verify the changes, again type

```
*1000 Return
```

The Monitor now displays

```
00/1000:54 -T
*
```

You can combine opening a location and changing its contents into a single operation by specifying the address, a colon, and the contents on a single command line:

```
*1000:41 Return
```

As before, you can verify that the system obeyed your command by typing

```
*1000 Return
```

The Monitor now displays

```
00/1000:41 - A
*
```

You can change a byte to an ASCII code using the character instead of the numeric value. Use the same syntax as before, but enclose the ASCII characters in double quotation marks, as follows:

```
*1000:"a"
```

To verify that the location has been changed, type

`*1000` Return

Again, the *bank/address* and location contents are displayed.

```
00/1000:E1-a
*
```

Note that when you change the contents of a programmable memory location, the new value that you provide entirely replaces the value that was in that location to begin with. This new value will remain there until you replace it with another value or until you turn off the computer. Further information about this operation is provided in the section "ASCII Filters for Stored Data" later in this chapter. (If you are using the ASCII input mode, the filter will affect the data that you have entered.)

Changing consecutive memory locations

You don't have to type a separate command with an address, a colon, a value, and a Return for each location you want to change. You can change the values of many memory locations at the same time by typing only the initial address and a colon, then all the values separated by spaces, and then Return. The only limitation is that the total length of the string, including the address, colon, all of the values and spaces, and the Return, must not exceed 255 characters. Using this method, you could change 100 or more locations in a single entry line. Note that you don't need to type leading zeros, a feature that provides even more possible data entry locations in a single command line.

The Monitor stores the consecutive values in consecutive locations, starting at the location whose address you typed. After it has processed the string of values, it takes the location following the last-changed location as the next-changeable location. Thus, you can continue changing consecutive locations without typing an address on the next input line by simply typing another colon, a space, and more values. In the following examples, you first change some locations and then examine them to verify the changes.

`*1000:56 57 58 59 60 61 62 63 64 65` Return

The contents of locations $1000 through $1009 have been changed, as you can see by examining those locations:

`1000.1009` Return

As before, the memory-bank number and the starting memory address precede the values you typed, and the ASCII values are displayed at the right.

```
00/1000:56 57 58 59 60 61 62 63 64 65-VWXY'abcde
*
```

In the next example, you use the colon to continue a data entry, as noted in the preceding description:

```
*1000:41 42 43 Return
*:3130 32 33 Return
*1000.1006 Return
00/1000:41 42 43 30 31 32 33-ABC0123
```

Note that you can enter data in either single-byte (one or two hex digits) or double-byte (three or four hex digits) or triple-byte (five or six hex digits) or quadruple-byte (seven or eight hex digits) units. When a double-byte quantity is entered, the Monitor stores the bytes in low-byte, high-byte sequence (the reverse of the way you entered them), as demonstrated in the example (3130 entry) above. This is useful when you are specifying address entries for the mini-assembler. You will find more of this kind of entry demonstrated in the section "The Mini-Assembler" later in this chapter.

ASCII input mode

You can enter ASCII data in two different ways. One way is called *literal ASCII;* the other way is called *flip ASCII.*

❖ *Note:* The ASCII filter will affect the final form of your data when ASCII input mode is used. See the section "ASCII Filters for Stored Data" later in this chapter for more information.

To enter data in literal ASCII format, type the character string you wish to enter between a pair of double quotation marks. The characters you enter are stored in ascending order in the same sequence in which you typed them. In some cases, you might want to store the characters in reverse order, with the last character stored at the lowest memory address. You use flip ASCII for this entry mode. Flip ASCII is entered by using single quotation marks in place of double quotation marks. Note, however, that flip ASCII is limited to four characters maximum. The following example demonstrates literal ASCII data entry:

```
1000:"ECHO" Return
1000.1003 Return
00/1000: C5 C3 C8 CF -  ECHO
```

The next example demonstrates flip ASCII data entry:

```
1000:'ECHO' Return
1000.1003 Return
00/1000: CF C8 C3 C5 -  OHCE
```

ASCII filters for stored data

When you perform any manipulation of ASCII code, you must consider the literal ASCII format of the stored data. For example, do you want the data to be stored in ASCII format with the most significant bit set (to be compatible with the I/O firmware for display purposes) or directly in true ASCII format, where what you type exactly follows the ASCII standard? The format can be changed using any filters provided by the Monitor. The filter can be any hex value from $00 (maximum filtering) to $FF (no filtering, that is, all source bits pass through the filter unmodified).

The filter formats are as follows:

Entry	Filter	Format of stored data
"abcdefghijkl"	FF (default filter)	E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC
	7F	61 62 63 64 65 66 67 68 69 6A 6B 6C
	3F	21 22 23 24 25 26 27 28 29 2A 2B 2C

The syntax for changing filters is

{*filter-value*}=F Return

For example, if you type

7F=F Return

the system uses the 7F filter format.

This means that when you search for any pattern in memory, you must know which format is used. If FF is used, abc appears in hex as E1 E2 E3; if 7F is used, abc appears as 61 62 63. Thus, if you perform a pattern search for E1 E2 E3 and the format used was 7F, you will not find the correct pattern.

The input ASCII character is ANDed with the filter value and then stored in the search buffer.

Moving data in memory

You can copy a block of data stored in a range of memory locations from one area in memory to another by using the Monitor's Move (M) command. To move a range of memory, you must tell the Monitor both where the data values are now situated in memory (the source locations) and where the data values are to go (the destination locations). You give this information to the Monitor by providing three addresses: the address of the first location in the destination and the addresses of the starting and ending locations within the source range. You specify the starting and ending addresses of the source range by separating them with a period. You separate the destination address from the range addresses with a less-than character (<), which you may think of as an arrow pointing in the direction of the move. Finally, you tell the Monitor that this is a Move command by typing the letter M (in either lowercase or uppercase).

The format of the complete Move command looks like this:

```
{destination}<{from_address}.{to_address}M
```

To move data from $1000 through $1009 to locations beginning at $2000, type the destination, the starting address, and the ending address followed by the letter `M`. Note that as you type the address values, the words in braces and the braces themselves are replaced by the hexadecimal addresses that you wish to use. The example uses bank $00 as both the source and the destination. You can, however, specify the complete bank address within either of the source addresses or in the destination address, because everywhere that the Monitor requires an address, it will also find the combination of {*bank/address*} acceptable as well.

```
*2000<1000.1009M Return
*
```

Now examine the data you moved by using the examine procedure. Type the starting address and the ending address and press Return:

```
*2000.2009 Return
```

The data returned to the display looks the same as it did when you examined locations $1000 through $1009:

```
00/2000:CF C8 C3 C5 60 61 62 63 64 65-OHCE'abcde
*
```

The Monitor moves a copy of the data stored in the source range of locations to the destination locations. The values in the source range are left unchanged. The Monitor remembers the last location in the source range as the last-opened location and the first location in the source range as the next-changeable location. If the second address in the source range is less than the first, then only one value (that of the first location in the range) will be moved.

If the destination address of the Move instruction is inside the source range of addresses, then strange (and sometimes wonderful) things happen: The locations between the beginning of the source range and the destination address are treated as a subrange, and the values in this subrange are replicated throughout the source range. The section "Special Tricks With the Monitor" later in this chapter provides an interesting application of this feature.

Comparing data in memory

You can use the Verify (V) command to compare two ranges of memory using the same format you use to move a range of memory from one place to another. In fact, a Verify command can be used immediately after a Move command to make sure that the move was successful.

The Verify command, like the Move command, needs a range and a destination. The syntax of the Verify command is identical to the Move command, except that you type a `V` in place of an `M`:

{*destination_address*} < {*starting_address*}.{*ending_address*}V

The Monitor compares the values in the source locations with the values in the locations, beginning with the destination address. If any values don't match, the Monitor displays the first address at which a discrepancy is found and the two values that differ. If you enter the example shown for the Move instruction and then change one byte at the destination, you can use the Verify command to find the discrepancy. Change the first location to hex 41 (it was hex 56) and then use the Verify command:

```
*2000:41 Return
*2000<1000.1009V Return
```

If there are no discrepancies, you will not get a display. In this example, because you will have caused a discrepancy, the following is displayed:

```
00/1000:56 (41) ——$2000
          \
           $1000
```

Location $1000 contains 56; location $2000, however, contains 41.

The Verify command leaves the values in both ranges unchanged. The last-opened location is the last location in the source range, and the next-changeable location is the first location in the source range, just as in the Move command. If the ending address of the range is less than the starting address, the values of only the first locations in the source and destination will be compared. Like the Move command, the Verify command also does strange things if the destination address is within the source range. Again, see the section "Special Tricks With the Monitor" later in this chapter.

Filling a memory range

You can fill a memory range with a specific value by using the Monitor Zap (Z) command. You tell the Monitor where and how to zap memory by providing three pieces of information: the value to fill, the starting address, and the ending address. You separate the value from the starting address by using a less-than character (<). You separate the beginning and ending addresses of the range with a period. The syntax for Zap is

```
{value}<{starting_address}.{ending_address}Z Return
```

When Zap operates, the value you have selected is filled into the entire range, including the starting and ending addresses.

Searching for bytes in memory

The Pattern Search (P) command allows you to search for one or more bytes (hexadecimal values, ASCII characters, or a combination of the two) in a range of memory. The syntax of the pattern search instruction is as follows:

```
*\{value(s) or "literal ASCII" or 'flip ASCII'}\<{starting_address.ending_address}P
```

The byte values are entered end to end with no intervening spaces. This format is required by the Pattern Search command because you are looking for a string of values. Note that you must enter leading zeros. For example, a search for the string of characters 0D followed by 0A between locations 1200 and 1400 would be entered as

```
*\0D 0A\<1200.1400P Return
```

If you are looking for a string of characters, you can enter the characters delimited by double quotation marks as shown here:

```
*\"Mr. Goodbar"\<1200.1400P Return
```

If the pattern is found, the beginning location is displayed. For example, if the pattern is located with its first byte at location $1300, the following is displayed:

```
00/1300:41 -A
*
```

Registers and flags

The Apple IIGS system uses a number of registers and control flags (bits) to perform its various functions. Table 3-3 lists these registers and flags.

Table 3-3
Registers and flags

Register		Flag	
A	Accumulator	M	Machine state
Y	Index register	Q	Quagmire state
X	Index register	m	Accumulator mode
S	Stack pointer	x	Index mode
D	Direct zero page	e	Emulation mode
P	Processor status	L	Language-card bank
B	Data bank		
K	Program bank		

The A, X, and Y registers are the workhorses of the assembly-language programmer. The P register contains all of the system status flags. The D register is the 65816 direct-page register that controls the placement of the zero page of the processor. The S register is the stack pointer. The K register contains the upper 8 bits of the program counter because the 65816 operates anywhere in a 24-bit address space.

In books that describe programming for the 65816, the upper 8 bits of the accumulator are sometimes called the *B register*. These programming books also refer to the 16-bit accumulator as the *C register*, the program bank register as *PBR* (the upper 8 bits of the program counter), and the data bank register as *DBR* (the upper 8 bits applied to the X and Y registers). For convenience, the Monitor renames these registers as follows:

- The Monitor B register display shows the DBR contents.
- The Monitor K register display shows the PBR contents.
- The Monitor A register display shows the 16-bit accumulator contents, whether 8 or 16 bits.
- The Monitor does not separately display the upper 8 bits of the accumulator.

Note that the Monitor does not display the current contents of the program counter register. If you want to step or trace a program, you must create your own separate routine to display the program counter contents along with these other registers.

The M register represents the machine state. The individual bits of this register are described in the summary at the end of this chapter. You can find an in-depth description of the meaning of these bits in the *Apple IIGS Hardware Reference*.

The Q register, also called the **Quagmire register,** is not actually a hardware machine register, but a pseudoregister made up of control bits located elsewhere in the system. One bit (bit 7), selects high-speed operation. (Earlier Apple II series computers operated only at 1 MHz; the Apple IIGS can operate either at 1.0 MHz or 2.8 MHz.) Bits 6 to 0 enable and disable various shadowing options. Shadowing, when enabled, writes the same data to banks $00 (or $01) and $E0 (or $E1) in selected areas, as defined by the individual shadowing bits.

The environment

The complete set of registers and flags is called the *environment*. When your program encounters a break or another kind of interrupt condition, this environment is saved by the Monitor. When you issue a command to resume execution, the environment is restored as it was when the interrupt occurred. Your program resumes as though nothing had happened. If you change the contents of the registers and flags that are displayed, then the changes become the new environment that your program encounters when it again begins to execute. You also change the registers and flags to set up a new environment for a program that you might write and execute using the Go command, discussed later in this chapter.

Examining and changing registers and flags

The microprocessor's register contents change continuously during execution of a program, such as the Monitor firmware. Using the Monitor, you can see what the register contents were when you invoked the Monitor or when a program you were debugging stopped at a Break (BRK) or a COP instruction or as a result of an unserviced hardware abort condition.

Table 3-4 lists the commands that relate to system registers.

Table 3-4
Commands for viewing and modifying registers

Command type	Command format
Examine registers	Control-E
Modify accumulator	{*val16*}=A
Modify X register	{*val16*}=X
Modify Y register	{*val16*}=Y
Modify D register	{*val16*}=D
Modify DBR register (bank)	{*val*}=B
Modify program bank register	{*val*}=K
Modify stack pointer	{*val16*}=S
Modify processor status	{*val*}=P
Modify machine-state register	{*val*}=M
Modify Quagmire register	{*val*}=Q
Modify 16/8-bit accumulator mode	{*val*}=m
Modify 16/8-bit index mode	{*val*}=x
Modify native/emulation mode	{*val*}=e
Modify language-card bank	{*val*}=L
Modify ASCII filter mask	{*val*}=F

When you call the Monitor, it stores the contents of the microprocessor's registers and flags in memory. The registers and flags are stored in the order A, X, Y, S, D, P, B, K, M, Q, L, m, x, and e. When you give the Monitor a G instruction, the Monitor loads the registers in this same sequence before it executes the first instruction in your program. The m, x, and e flags are part of the processor status register (P). However, because the registers and flags are reloaded in the sequence shown, whatever value you have placed in m, x, and e will override any such value you might have placed in P.

❖ *Note:* If you set the value of the e flag to 1, the 65816 automatically sets the value of m and x to 1. This puts the processor into 6502 emulation mode, forcing it to have an 8-bit accumulator and index registers. Additionally, the upper 8 bits of the stack pointer are forced to a value of 01.

Press Control-E and then Return to invoke the Monitor's Examine instruction. This action displays the stored register values and flags and sets the location containing the contents of the A register as the next-changeable location. The example follows:

`*Control-E` Return

The registers and flags are displayed as follows:

You can change the values in any of these locations by typing the new value, an equal sign (=), and the letter for the register or flag to affect and pressing Return. In the following example, the first two locations are changed, and the registers and flag bits are again displayed to verify the change.

Change A to the value 1234:

`*1234=A` Return

Change X to the value 006A:

`*006A=X` Return

Execute the Examine instruction:

*Control-E

The registers and flags are displayed to verify the changes:

```
A=1234 X=006A Y=C3CB S=01F4 D=0000 P=00 B=00 K=00 M=0C Q=80 L=1 m=1 x=1 e=1
```

❖ *Note:* If you are using the Monitor to debug a program running in 6502 emulation mode, the values for the microprocessor registers will revert to their 6502 equivalents. For example, the A, X, Y, and S registers will be able to hold only 8 bits each. Even if you specify (and display) a value that exceeds 8 bits, only the low 8 bits of the value you enter will be used when the system resumes 6502 emulation.

Summary of register- and flag-modification commands

The following commands can be used to modify the registers and flags. Note that all of these are case sensitive. To change the register you want to change, you must use the case (uppercase or lowercase) shown in the registers and flags display. The case of the letters is the only way the Monitor can distinguish between flags and registers in this situation (for example, compare X and x and M and m in the following list).

Change to	Syntax
Accumulator	{*val16*}=A
X register	{*val16*}=X
Y register	{*val16*}=Y
D register	{*val16*}=D
DBR register (bank)	{*val*}=B
Program bank register	{*val*}=K
Stack pointer	{*val16*}=S
Quagmire register	{*val*}=Q
Machine register	{*val*}=M
m flag	{*val*}=m (*val* = 0 for 16-bit accumulator, *val* = 1 for 8-bit accumulator)
x flag	{*val*}=x (*val* = 0 for 16-bit index registers, *val* = 1 for 8-bit index registers)
e flag	{*val*}=e (*val* = 0 for native mode, *val* = 1 for 6502 emulation mode)
Filter value for ASCII modes	{*val*}=FF (*val* = any value from \$00-\$FF; default *val* = FF)
Language-card bank	{*val*}=L (*val* = 0 or 1)

Miscellaneous Monitor commands

Other Monitor commands enable you to change the video display format from normal to inverse and back and to assign input and output to accessories in expansion slots. Table 3-5 lists these miscellaneous commands.

Table 3-5
Miscellaneous Monitor commands

Command type	Command format
Begin inverse video	I
Begin normal video	N
Change time and date	=T=*mm/dd/yy hh:mm:ss*
Display time and date	=T
Redirect input links	{*slot*} Control-K
Redirect output links	{*slot*} Control-P
Change screen display to text	Control-T
Change cursor	Control-^ {*new_cursor_character*}
Convert decimal to hexadecimal	={*val10*}
Convert hexadecimal to decimal	{*val64*}=
Perform hexadecimal math	
Add	{*val64*}+{*val64*}
Subtract	{*val64*}-{*val64*}
Multiply	{*val64*}*{*val64*}
Divide	{*val64*}_{*val64*}
Jump to cold-start BASIC	Control-B
Jump to warm-start BASIC	Control-C
Jump to user vector	Control-Y
Quit Monitor	Q

Inverse and normal display

You can control the setting of the inverse/normal mask location used by the COUT subroutine from the Monitor so that all of the Monitor's output will be in inverse format. The COUT routine is described in Chapter 4, "Video Firmware." The Inverse command (I) sets the mask so that all subsequent input and output are displayed in inverse format.

```
*I  Return
```

To switch the Monitor's output back to normal format, use the Normal command (N).

```
*N  Return
```

Working with time and date

You can display or set the time and date directly from the Monitor. (Normally, time setting is handled through the Control Panel, which is described in Appendix G, "The Control Panel.")

Here is the format for displaying the time and date:

=T Return

If you want to set the time and date, use the following format (for decimal number entry):

=T=*nn*/*dd*/*yy* *hh*:*mm*:*ss*

where *nn* is the month (range 1–12), *dd* is the day (range 1–31), *yy* is the year (range 0–99), *hh* is the hour (range 0–23), *mm* is the minutes (range 0–50), and *ss* is the seconds (range 0–59). The delimiters slash (/) and colon (:) are shown as the suggested format because these delimiters conform to what a user normally expects to see. However, any delimiter other than an apostrophe (') can be used to separate the values entered.

Redirecting input and output

The Printer command, activated by Control-P, diverts all output normally destined for the screen to an interface card in a specified expansion slot, from 1 to 7. There must be an interface card in the specified slot or you will lose control of the computer and your program and variables may be lost. The format of the command is

{*slot-number*} Control-P

A Printer command to slot 0 will switch the stream of output characters back to the Apple IIGS video display.

Don't issue the Printer command using a slot value of 0 to deactivate the 80-column firmware, even though you used this command to activate it in slot 3. The command works, but it just disconnects the firmware, leaving some of the soft switches set for 80-column display.

In much the same way that the Printer command switches the output stream, the Keyboard command substitutes the interface card in a specified expansion slot for the normal Apple IIGS input device, the keyboard. The format for the Keyboard command is

{*slot-number*} Control-K

Specifying slot number 0 for the Keyboard command directs the Monitor to accept input from the Apple IIGS keyboard.

The Printer and Keyboard commands are the equivalents of BASIC commands PR# and IN#.

Changing the cursor character

You can change the Monitor cursor from a flashing blank space to whichever character you wish. Here is the format for changing the cursor:

Control-^ {*new_cursor_character*}

Here is an example that sets an underscore (_) as your new cursor character:

```
*Control-^_ Return
*_
```

The underscore now appears as the cursor character. To restore the original cursor, specify that the new cursor is a delete character.

Converting hexadecimal and decimal numbers

You can convert up to 8-digit hexadecimal numbers to decimal values. The syntax is

{*value*}={Return}

For example, type

```
*000F= Return
```

Hexadecimal $000F is converted to decimal 15:

```
15 {+15}
*
```

You can also convert a decimal number to a hexadecimal number. The syntax is as follows:

={*value*} Return

For example, type

```
*=0015 Return
```

Decimal 0015 is converted to hexadecimal $0000000F:

```
$0000000F
*
```

Hexadecimal math

You can use the Monitor to perform hexadecimal math. The Apple IIGS Monitor can handle 32-bit addition, subtraction, multiplication, and division operations. The syntax for these operations is shown below. Note that multiplication shows a 64-bit result, and division displays both the remainder and the quotient. Notice also that bank-address information provided in the entry of the data is ignored during the calculations. If you wish to actually perform address calculations, you can convert your bank and address into a 6-digit hexadecimal quantity and use that for the calculations (just leave out the forward slash).

Operation	Syntax
Addition	{*val64*}+{*val64*} Return
Subtraction	{*val64*}-{*val64*} Return
Multiplication	{*val64*}*{*val64*} Return
Division	{*val64*}_{*val64*} Return (An underscore character rather than the traditional forward slash is used to specify division.)

Here are a few examples:

```
*1234+1234 Return
-> $00002468
*1234+34 Return
-> $00001268
*34+1 Return
-> $00000035
*1112-2222 Return
-> $FFFFEEF0
*12*3456789 Return
-> $000000003AE147A2
*12345678_120 Return
R-> $000000D8  Q-> $00102E85
*0/23+1/23 Return
-> $00000046    (Bank-address information was ignored.)
```

A Tool Locator call

From the Monitor, it is possible to call the toolbox routines. However, the toolbox routines will most often be used by programs rather than by keyboard access through the Monitor. The syntax for the Tool Locator call is listed in detail in the summary at the end of this chapter. If you wish to use tool calls from the Monitor, see the *Apple IIGS Toolbox Reference* for details about the tool numbers and parameter requirements for the tool of your choice.

As an example of a possible use, here are two sample tool calls. The first call, once entered, allows you to type a line of text, followed by a carriage return. This first call returns a count, in hexadecimal, of the number of characters you typed. You will then store the number you receive into a memory location and call another tool that will retrieve and type the characters to the display.

This first tool call reads the keyboard, storing successive characters in locations beginning in memory location $012080 until you type a carriage return character.

`\C 2 0 0 0 1 20 81 OFF 0 8D 0 1 24 C\U` Return

After you input some text and press Return, the Monitor responds with a hex count of the number of characters you typed. If you typed

`THESE ARE MY LETTERS.` Return

the Monitor responds

```
*15
*
```

Now type the following line after the Monitor prompt to store that number you received into memory to set up for the tool to type the text. The hex value that you enter in this memory-modification command is the same value that the tool returned as your character count.

`01/2080:15` Return

The following command asks a tool to type the text:

`\4 0 0 1 20 80 1C C\U` Return

Back to BASIC

Use the BASIC command, Control-B, to leave the Monitor and enter the BASIC that was active when you entered the Monitor. Normally, this is Applesoft BASIC, unless you deliberately switched to Integer BASIC. Note that if you use this command, any program or variables that you had previously entered in BASIC will be lost. If you want to reenter BASIC with your previous program and variables intact, use the Continue BASIC command, Control-C.

If you are using DOS 3.3 or ProDOS®, press Control-Reset or use the Monitor Q (Quit) command to return to the language you were using with your program and variables intact.

Special tricks with the Monitor

This section describes some more complex ways of using the Monitor commands, including

- □ placing multiple commands on a single command line
- □ filling memory with a multiple-byte pattern
- □ repeating commands
- □ creating your own commands

Multiple commands

You can put as many Monitor commands on a single line as you like, so long as you separate them with spaces and the total number of characters in the line is less than 254. Adjacent single-letter commands such as L, S, I, and N need not be separated by spaces.

You can freely mix all of the commands except the Store (:) command. Because the Monitor takes all values following a colon and places them in consecutive memory locations, the last value in a Store command must be followed by a letter command before another address is entered. You can use the Normal command as the letter command in such cases; it usually has no effect on a program and can be used anywhere.

In the following example, you display a range of memory, change it, and display it again, all with one line of commands:

```
*1300.1307 1300:38 39 1 N 1300.1302 Return
00/1300 - 00 00 00 00 00 00 00 00 38 39 01-........89
*
```

If the Monitor encounters a character in the input line that it does not recognize as either a hexadecimal digit or a valid command character, it executes all the commands on the input line up to that character. It then grinds to a halt with a beep and ignores the remainder of the input line.

Filling memory

The Move command can be used to replicate a pattern of values throughout a range of memory. To do this, first store the pattern in the first locations in the range:

```
*1300:11 22 33-."3
*
```

Remember the number of values in the pattern; in this case, it is 3. Use this number to compute addresses for the Move command, like this:

{*start-number*}<{*start*}.{*end-number*}M

This Move command first replicates the pattern at the locations immediately following the original pattern, then replicates that pattern following the first replication, and so on until it fills the entire range:

```
*1303<1300.1334M
*1300.1317 Return
00/1300 -      11 22 33 11 22 33 11 22 33 11 22 33 11 22 33 11-."3."3."3."3."3.
00/1310 -      22 33 11 22 33 11 22 33-"3."3."3
*
```

You can perform a similar trick with the Verify command to check whether a pattern repeats itself through memory. Verify is especially useful for verifying that a given range of memory locations all contain the same value. In the following example, you first fill the memory range from $1300 to $1320 with zeros and verify it; you then change one location and verify it again:

```
*1300:0
*1301<1300.1320M
*1301<1300.1320V
*1304:02
*1301<1300.1320V
1303 -     00     (02)
1304 -     02     (00)
*
```

The Verify command detects the discrepancy.

Repeating commands

You can create a command line that continuously repeats one or more commands. You do this by beginning the part of the command line that you want to repeat with a letter command, such as N, and ending it with the sequence 34:n, where n is a hexadecimal number that specifies the position in the line of the command where you want to start repeating. For the first character in the line, n = 0. The value for n must be followed by a space for the loop to work properly.

This trick takes advantage of the fact that the Monitor uses an index register to step through the input buffer, starting at location $0200. Each time the Monitor executes a command, it stores the value of the index at location $34; when that command is finished, the Monitor reloads the index register with the value at location $34. By making the last command change the value at location $34, you change this index so that the Monitor picks up the next command character from an earlier point in the buffer.

The only way to stop a loop such as this is to press Control-Reset; that is how the following example ends:

```
*N 1300 1302 34:0 Return
1300 -      11
1302 -      33
1300 -      11
1302 -      33
1300 -      11
1302 -      33
1300 -      11
1302 -      33
1300 -      11
1302 -      33
1300 -      11
1302 -      33
130             (Control-Reset is pressed here; the Monitor jumps to Applesoft.)
]
```

Creating your own commands

The User command, Control-Y, forces the Monitor to jump to memory location $03F8. You can put a JMP instruction there that jumps to your own machine-language program. Your program can then examine the Monitor's registers and pointers or the input buffer itself to obtain its data. For example, the following program displays everything on the input line after Control-Y. The program starts at location $0300; the command line that starts with $03F8 stores a jump to $0300 at location $03F8. Here is the program, followed by a listing of the method by which it is entered into the Monitor.

The program:

```
      LDX  34      ;Get the index from location $34
                   ;Points to next character position in input line
MORE  LDA  200,x   ;Get that character into accumulator
      JSR  COUT    ;Output the character
      INX          ;Point to the next character
      CMP  #8D     ;See if it is a carriage return
      BNE  MORE    ;If not, go get more
      JMP  MONZ    ;Jump to standard monitor entry point (Call -151)
```

Entering the program into the Monitor:

```
*300:A4 34 B9 200 20 FDED C8 C9 8D DO F5 4C FF69
*3F8:4C 300
*Control-Y THIS IS A TEST
THIS IS A TEST
*
```

Notice that the target addresses for the JSR (jump to subroutine) instructions (value of hex 20) are entered directly as their 4-digit hexadecimal values rather than as separate byte pairs in reverse order as would normally have been required for the system Monitor in machines prior to the Apple IIGS. You can enter full 32-bit addresses in this manner if you wish (up to 8 hexadecimal digits, forming a 32-bit quantity).

Machine-language programs

The main reason to program in machine language is to get more speed. A program in machine language can run much faster than the same program written in high-level languages such as BASIC or Pascal, but the machine-language version usually takes a lot longer to write. There are other reasons to use machine language: You might want your program to do something that isn't included in your high-level language, or you might just enjoy the challenge of using machine language to work directly on the bits and bytes. It is highly unlikely that a serious software developer will use the mini-assembler to produce large programs. However, the mini-assembler is a useful tool for quickly checking various basic concepts. Sometimes just the ability to examine memory is very handy.

❖ *Note:* If you have never used machine language before, you'll need to learn the language of the 65C816. To become proficient in machine-language programming, you'll have to spend some time working with it and study at least one book on 65C816 and perhaps also 6502 or 65C02 programming.

You can get a hexadecimal dump of your program, move your program around in memory, examine and change register contents, and so on using the commands described in the previous sections. The Monitor commands in this section are intended specifically for you to use in creating, writing, and debugging machine-language programs. Table 3-6 lists the commands that relate to program creation and debugging.

Table 3-6
Commands for program execution and debugging

Command type	Command format
Go (begin) program in bank $00	{*start_address*}G
Execute from any memory bank	{*start_address*}X
Restore registers and flags	Control-R
Resume execution	{*start_address*}R
Perform a program step	{*start_address*}S
Perform a program trace	{*start_address*}T
Disassemble (list)	{*start_address*}L
Enter mini-assembler	!

Running a program in bank zero

The Monitor command you use to start execution of your machine-language program is the Go command. When you type an address and the letter G, the Apple IIGS restores all of the machine registers from their stored locations and begins executing machine-language instructions starting at the specified location. If you type only G, execution starts at the last-opened location. The syntax of the Go command is

{*start_address*}G Return

The Monitor treats this program as a subroutine and executes a JSR to the program. If you want the routine to end by returning control to the Monitor, your program must end with an RTS (return from subroutine) instruction to transfer control back to the Monitor.

The Monitor has some special features that make it easier for you to write and debug machine-language programs; but before you learn about these, here is a small machine-language program that you can run using only the simple Monitor commands already described. The program in the example displays the letters A through Z. Store it starting at location $0300, examine it to be sure you typed it correctly, and then type `300G` to start it running.

```
*300:A9 C1 20 FDED 18 69 1 c9 DB DO F6 60 Return
*300G Return
abcdefghijklmnopqrstuvwxyz
*
```

This is the assembly code that represents the preceding hand-assembled program:

```
      LDA #C1    ;Place ASCII for "A" into accumulator
OUT   JSR COUT   ;Note: Mini-assembler does not use labels
      CLC
      ADC #1     ;Add 1 to contents of accumulator
      CMP #DB    ;Compare contents to a value of ASCII ("Z"+1)
      BNE OUT    ;If not, go back and output accum value again
```

The G instruction works only for code in bank $00. The system beeps if the user specifies any bank other than $00. The G instruction sets up a JSR to the code and expects this code to end in an RTS.

Running a program in other banks of memory

You can run programs in banks other than bank $00 by using the X command instead of the G command. The X command restores all of the machine registers from their stored locations and begins executing at the specified location. A JSL instruction (jump to subroutine long) is performed instead of a JSR, and the user's code is expected to end with an RTL (return from subroutine long). The syntax of the X command is

{*start_address*}X Return

Resuming program execution

You can resume execution of programs halted by a deliberate BRK (Break) instruction or Trace command by using the R command (Resume). Run programs in banks other than bank $00 by using the X command instead of the G command. The R command restores all of the machine registers from their stored locations and begins executing at the location you specify. A JMP instruction is performed instead of a JSR or JSL because the Resume command assumes that you do not intend to return to the Monitor.

Stepping through or tracing program execution

The Apple IIGS Monitor includes two commands for stepping through a program one instruction at a time and for tracing program execution (performing multiple steps). You put the Monitor into Step mode by using the S command. You put the Monitor into Trace mode by using the T command. (These commands, though present, are not fully implemented.) The Step command prints `"STEP"` and returns control to the Monitor. The Trace command prints `"TRACE"` and returns control to the Monitor. If you want to implement your own Step and Trace functions, simply modify the Step and Trace vector locations to point to your own custom version of each routine. These vectors are shown in Appendix D, "Vectors." The formats for Step and Trace are shown in the summary at the end of this chapter.

The mini-assembler

The Apple IIGS mini-assembler included in the Monitor program allows you to enter machine-language programs directly from the keyboard. ASCII characters or hex values can be entered into a mini-assembler program exactly as you enter them in the Monitor. The mini-assembler doesn't accept labels; you must use actual values and addresses.

When you enter the mini-assembler, the Monitor prompt character changes from * to ! (the mini-assembler prompt) and assembles the first line of code (if a line of code is typed on the same line as the exclamation point that caused the mini-assembler to be entered).

Starting the mini-assembler

To start the mini-assembler, first invoke the Monitor from BASIC by typing

`Call -151` Return

Then, from the Monitor, type

`!` Return

or

`!` ⟨*bb/addr*⟩ `:` ⟨*opcode*⟩ ⟨*operand*⟩ Return

Using the mini-assembler

The mini-assembler saves one address, that of the program counter. Before you start typing a program, you must set the program counter to point to the location where you want the mini-assembler to store your program. Do this by typing the address followed by a colon. Then type the mnemonic for the first instruction in your program, followed by a space and the operand of the instruction.

`!300:LDX #02` Return

The mini-assembler converts the line you typed into hexadecimal format, stores it in memory beginning at the location of the program counter, and then disassembles it again and displays the disassembled line. The prompt is then displayed on the next line.

```
00/0300- A2  02     LDX     #02
!
```

The mini-assembler is now ready to accept the second instruction in your program. To tell it that you want the next instruction to follow the first, don't type an address or a colon; type a space and the next instruction's mnemonic and operand and then press Return.

The first space after the exclamation point (!) controls the nature of the digits that follow:

- □ A space means you want the next instruction to follow the first.
- □ A colon (:) means hexadecimal information follows.
- □ A double quotation mark (") means ASCII information follows.
- □ A number means an address follows.

The first instruction is as follows:

```
! LDA $0,X Return
```

The mini-assembler assembles that line and is then ready for the next instruction.

```
00/0302-  B5  00     LDA    00,X
!
```

The following example shows the procedure for entering a program using the mini-assembler. The instructions you type are shown on a line with the prompt character (!); the assembled display is shown, in each case, on a line without a prompt character.

```
!300:LDX  #02
00/0300-  A2 02     LDX     #02
! LDA 0,X
00/0302-  B5 00     LDA     00,X
! STA $10,X
00/0304-  95 10     STA     10,X
! DEX
00/0306-  CA        DEX
! STA $C030
00/0307-  8D 30 C0 STA     C030
! BPL $302
00/030A-  10 F6     BPL     0302
! BRK 00
00/030C-  00 00     BRK     00
```

❖ *Note:* Don't forget the space after the exclamation point. The program needs the space after the exlamation point to follow the address precedent set by the initial instruction.

If you want to enter a program in hexadecimal notation, you must start in the hex mode, as the following example indicates:

```
!1000::23 24 25
:60 61 C1
```

If an instruction line has an error in it, the mini-assembler beeps loudly and displays a caret (^) under or near the offending character in the input line. If you forget the space before or after a mnemonic or include an extraneous character in the hexadecimal value or address, the mini-assembler rejects the input line. If the destination address of a branch instruction is out of the range of the branch (more than 127 locations distant from the address of the instruction), the mini-assembler flags this as an error.

To leave the mini-assembler and reenter the Monitor, press Return immediately after the ! prompt.

Your assembly-language program is now stored in memory. You can display it with the List (L) instruction as follows:

```
*300L
1=m   1=x    1=LCBank(0/1)
00/0300- A2 02       LDX      #02
00/0302- B5 00       LDA      00,X
00/0304- 95 10       STA      10,X
00/0306- CA          DEX
00/0307- 8D 30 C0    STA      C030
00/030A- 10 F6       BPL      0302
00/030C- 00 00       BRK      00
00/030E- 00 00       BRK      00
00/0310- 00 00       BRK      00
00/0312- 00 00       BRK      00
00/0314- 00 00       BRK      00
00/0316- 00 00       BRK      00
00/0318- 00 00       BRK      00
00/031A- 00 00       BRK      00
*
```

(After the program is displayed, the List instruction displays enough lines of code to fill the screen.)

Mini-assembler instruction formats

The mini-assembler recognizes 256 mnemonics and 24 addressing formats. Table 3-7 shows the address formats for the 65C816 assembly language. (Mini-assembler opcodes are listed in Appendix F, "Disassembler/Mini-Assembler Opcodes.")

Table 3-7
Mini-assembler address formats

Mode	Name	Format
a	Absolute	1234
a,x	Absolute indexed (with x)	1234,X
a,y	Absolute indexed (with y)	1234,Y
(a,x)	Absolute indexed indirect	(1234,X)
al,x	Absolute indexed long	081234,X
(a)	Absolute indirect	(1234)
al	Absolute long	081234
Acc	Accumulator	Blank
xya	Block move	01,02
d	Direct	45
d,x	Direct indexed (with x)	45,X
d,y	Direct indexed (with y)	45,Y
(d,x)	Direct indexed indirect	(45,X)
(d)	Direct indirect	(45)
(d),y	Direct indirect indexed	(45),Y
[d],y	Direct indirect indexed long	[45],Y
[d]	Direct indirect long	[45]
#	Immediate	#23 or #2345
i	Implied	Blank
r	Program counter relative	1000 {+50}
rl	Program counter relative long	1000 {–0200}
s	Stack	Blank
r,s	Stack relative	10,S
(r,s),y	Stack relative indirect indexed	(10,S),Y

An address consists of one or more hexadecimal digits. The mini-assembler interprets addresses the same way the Monitor does: If one, three, or five digits are entered, a preceding zero is automatically entered as well. For example, the instruction LDA #1 is assembled as A9 01.

❖ *Note:* The dollar signs ($) used in this manual to signify hexadecimal notation are ignored by the mini-assembler and may be omitted when typing programs.

Branch instructions, which use the relative addressing mode, require the target address of the branch. The mini-assembler automatically calculates the relative distance to use in the instruction. If the target address is more than the allowable distance from the current program counter, the mini-assembler sounds a beep, displays a caret (^) under the target address, and does not assemble the line.

If you give the mini-assembler the mnemonic for an instruction and an operand and the addressing mode of the operand cannot be used with the instruction you entered, the mini-assembler will not accept the line.

The Apple IIGS tools

As you are creating a program, you will very likely want to incorporate calls to various Apple IIGS tools into your program. To use the tools, you need an intimate knowledge of the tools themselves. You should therefore consult the appropriate *Apple IIGS Toolbox Reference* manual for information about each tool. The Monitor includes a Tool Locator call as one of the commands. The format and details are given in the command summary at the end of this chapter.

The Tool Locator command actually performs a call to the selected tool, performs the desired function, and provides you with debug information about the data that the tool provides as return values.

The Tool Locator call lets you type a one-line command instead of requiring that you create a program to test the tool. See the *Apple IIGS Toolbox Reference* for more information.

The disassembler

Because hexadecimal code is so difficult to read and understand, you may want to translate machine language back into assembly language. You can use the List instruction as a disassembler for this purpose.

The Monitor List instruction has the format

{*start_address*}L

The List instruction starts at the specified location and displays a full screen (20 lines) of instructions. For example, if you want to display a list of instructions starting at location $1000 in bank 12, type

*12/1000L Return

The following list is displayed:

```
0=m   0=x   1=LCBank (0/1)
12/1000:  AD 15 18     LDA  1815
12/1003:  9D 50 10     STA  1050,X
12/1006:  9F 50 52 05  STA  055250
12/100A:  A9 77 66     LDA  #6677
12/100D:  82 20 10     BRL  2030 {+1020}
12/1010:  80 20        BRA  1032 {+20}
12/1012:  F4 12 34     PEA  3412
12/1015:  62 10 10     PER  2028
12/1018:  87 45        STA  [45]
12/101A:  62 00 F0     PER  001D {-1000}
12/101D:  A9 23        LDA  #0023
12/101F:  A2 45 67     LDX  #6745
12/1022:  4F 54 46 02  EOR  0244654
12/1026:  DC 89 23     JML  (2389)
12/1029:  7C BE F2     JSR  (F2BE,X)
12/102C:  73 40        ADC  (40,S),Y
12/102E:  C1 06        CMP  (06),Y
12/1030:  0A           ASL
12/1031:  00 23        BRK  23
12/1033:  B8           CLV
*
```

The top line of the disassembly shows you the current settings of the m and x bits of the 65C816 status register. Recall that you set these bits by using the {*val*}=m and {*val*}=x Monitor commands. Both affect the way the disassembly is performed by the Monitor. The LC (language-card) bank information shows you which of the two available language-card banks is currently active. You change the language-card bank by using the {*val*}=L command.

The disassembler can disassemble all 65C816 opcodes in emulation and native modes (both 8-bit and 16-bit native mode). In either native or emulation mode, the sizes of the accumulator and index registers are significant. In immediate mode, the sizes are important for the opcodes listed in Table 3-8.

Display Memory Location

{*from_address*}

Displays contents of memory location as

{*from_address*}: {*val*}-{*ASCII*}

Display a Range of Memory Locations

{*from_address*}.{*to_address*}

Displays memory.

In 40-column mode, type

```
*20/401.413
```

Memory contents from $0401 in bank 20 to $0413 in bank 20 are displayed in 40-column mode:

```
20/0401:C1 C2 C3 C4 C5 C6 C7-ABCDEFG
20/0408:C8 C9 CA CB CC CD CE CF-HIJKLMNO
20/0410:D0 D1 02 03-PQ.
```

In 80-column mode, type

```
*20/401.42
```

Memory contents from $0401 in bank 20 to $0413 in bank 20 are displayed in 80-column mode:

```
20/0401:C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CD CF-ABCDEFGHIJKOLMNO
20/0410:D0 D1 02 03 04 05 06 07 D2 D3 D4 D5 D6 D7 D8 D9-PQ......RSTUVWXY
20/0420:E1 E2-abcd
```

❖ *Note:* Printable ASCII characters are output as normal ASCII characters. Nonprintable characters are output as periods. In 40-column mode, half a page of memory can be displayed; in 80-column mode, a full page of memory can be displayed.

Terminate Memory Range

Control-X

Terminates Display Range of Memory Locations command.

Carriage Return

Return

Performs a carriage return with no preceding entry.

In 40-column mode, displays the contents of up to the next 8 locations in hexadecimal and ASCII formats. (Location starts at last {*bank/address*} entered and continues until the low nibble of the addresses being displayed equals 0 or 8.) See format of Display Memory Location command.

In 80-column mode, displays the contents of up to the next 16 locations in hexadecimal and ASCII formats. (Location starts at the last {*bank/address*} entered and continues until the low nibble of the addresses being displayed equals 0.) See format of Display Memory Location command.

Move, M

{*destination*}<{*from_address*}.{*to_address*}M

Moves data from {*from_address*} through {*to_address*} to locations starting at {*destination*}.

Verify, V

{*destination*}<{*from_address*}.{*to_address*}V

Compares the memory contents starting at {*destination*} through {*destination*}+({*to_address* − {*from_address*}) with the memory contents starting at {*from_address*} through {*to_address*} and verifies that they are the same.

Fill Memory, Z

{*val*}<{*from_address*}.{*to_address*}Z

Fills memory in the range {*from_address*} through {*to_address*} with the 1-byte value {*val*}.

Pattern Search, P

\{*val1*} {"*literal ASCII*"} {'*123t*'} {*val8*}\<{*from_address*}.{*to_address*}P

Searches for any length pattern up to 236 bytes in memory ranging from {*from_address*} through {*to_address*}; {*val*} can be hexadecimal, literal ASCII, or flipped ASCII. The address of each location where the pattern is found is output to the screen followed by a carriage return. The pattern search continues until the entire range of addresses has been examined.

Examine Registers

Control-E

Examines 65C816 registers and flags.

The screen displays

```
A=aaaa X=xxxx Y=yyyy S=ssss D=dddd P=pp
B=bb K=kk M=mm Q=qq L=1 m=m x=x e=e
```

On a 40-column screen, two lines are displayed automatically; on an 80-column screen, only one line is displayed.

Change the A Register, A

{*val16*}=A

Changes A register value to {*val16*} for Resume/Go/Execute/Step/Trace commands. *Note:* `A` must be uppercase.

Change X Register, X

{*val16*}=X

Changes X register value to {*val16*} for Resume/Go/Execute/Step/Trace commands. *Note:* `X` must be uppercase.

Change Y Register, Y

{*val16*}=Y

Changes Y register value to {*val16*} for Resume/Go/Execute/Step/Trace commands. *Note:* `Y` must be uppercase.

Change D Register, D

{*val16*}=D

Changes direct-page/zero-page register value to {*val16*} for Resume/Go/Execute/Step/Trace commands. *Note:* `D` must be uppercase.

Change Data Bank Register, B

`{val}=B`

Changes data bank register value to {*val*} for Resume/Go/Execute/Step/Trace commands. *Note:* `B` must be uppercase.

Change Program Register, K

`{val}=K`

Changes program register value to {*val*} for Resume/Go/Execute/Step/Trace commands. *Note:* `K` must be uppercase.

Change Stack Pointer, S

`{val16}=S`

Changes stack pointer value to {*val16*} for Resume/Go/Execute/Step/Trace commands. *Note:* `S` must be uppercase.

Change Processor Status, P

`{val}=P`

Changes processor status value to {*val*} for Resume/Go/Execute/Step/Trace commands. *Note:* `P` must be uppercase.

Change Machine State, M

`{val}=M`

Changes machine-state value to {*val*} for Resume/Go/Execute/Step/Trace commands. *Note:* `M` must be uppercase.

The M bits are as follows:

Bit 7 = 1	Makes alternate zero page/LC active
Bit 6 = 1	Makes Page 2 active
Bit 5 = 1	Makes RAMRD active
Bit 4 = 1	Makes RAMWRT active
Bit 3 = 1	Makes RDLCROM active, not read/write—read only
Bit 2 = 1	Makes LC bank 2 active
Bit 1 = 1	Makes alternate ROMBANK active
Bit 0 = 1	Makes INTCXROM active

Change Quagmire State, Q

{*val*}=Q

Changes Quagmire state value to {*val*} for Resume/Go/Execute/Step/Trace commands. (The Quagmire value controls shadowing and system speed). *Note:* Q must be uppercase.

The Q bits are as follows:

Bit 7 = 1 High speed
Bit 6 = 1 Stops IOLC shadowing
Bit 5 = 0 *Always* must be 0
Bit 4 = 1 Stops auxiliary-memory Hi-Res shadowing
Bit 3 = 1 Stops Super Hi-Res shadowing
Bit 2 = 1 Stops Hi-Res Page 2 shadowing
Bit 1 = 1 Stops Hi-Res Page 1 shadowing
Bit 0 = 1 Stops text Page 1 shadowing

Change Accumulator Mode, m

{*val*}=m

Changes accumulator mode value to {*val*} for Resume/Go/Execute/Step/Trace/List commands. *Note:* m must be lowercase.

0 = 16-bit mode
1 = 8-bit mode

Change Index Mode, x

{*val*}=x

Changes index mode value to {*val*} for Resume/Go/Execute/Step/Trace/List commands. *Note:* x must be lowercase.

0 = 16-bit mode
1 = 8-bit mode

Change Emulation Mode, e

{*val*}=e

Changes emulation-mode value to {*val*} for Resume/Go/Execute/Step/Trace/List commands. *Note:* e must be lowercase.

Change Language-Card Bank, L

`{val}=L`

Changes language-card bank value to `{val}` for Resume/Go/Execute/Step/Trace/List commands. *Note:* `L` must be uppercase.

0 = First bank of language card
1 = Second bank of language card

Change Filter Mask, F

`{val}=F`

Changes the ASCII filter mask value to `{val}` for mini-assembler ASCII entry and Monitor ASCII immediate-mode commands. The ASCII filter is ANDed with all ASCII characters entered in the Monitor. Affects both data entry and search conditions. Any value from $00 to $FF is valid. *Note:* `F` must be uppercase. The default value is `FF`.

Change Text Display, I (Inverse)

`I`

Switches to inverse video text display. *Note:* `I` must be uppercase.

Change Text Display, N (Normal)

`N`

Switches to normal video text display. *Note:* `N` must be uppercase.

Display Time and Date, T

`=T`

Displays current time and date. *Note:* `T` must be uppercase.

Change Time and Date

=T=*nn*/*dd*/*yy* *hh*:*mm*:*ss*

Changes time. *Note:* T must be uppercase. Any delimiter except an apostrophe (') may be used between values entered.

Enter

hh	= hours	0–23
mm	= minutes	0–59
ss	= seconds	0–59
nn	= month	1–12
dd	= day	1–31
yy	= year	0–99

Redirect Input Links, K

{*slot*} Control-K

Redirects input links to {*slot*}.

Redirect Output Links, P

{*slot*} Control-P

Redirects output links to {*slot*}.

Change Consecutive Memory

{*bank*/*address*}:{*val*} {*val*} {*val*} {"*literal ASCII*"} {'*flip ASCII*'} {*val*}

Changes consecutive memory locations starting at {*bank*/*address*} to the values after the colon (:). Values can be in hex, literal ASCII, or flip ASCII format.

Change Screen Display, T

Control-T

Changes screen display to text Page 1, regardless of current soft-switch settings.

Change Cursor

Control-^ {*character*}

Changes the cursor to a {*character*} symbol. This command is implemented through COUT1 and C3COUT1. It is not an input command; it works only through the BASIC output links. If {*character*} is the Delete character, the original cursor is restored.

Convert Hexadecimal to Decimal Format

{*val64*}= Return

Converts hexadecimal number entered to decimal number (8-digit hex number maximum). Result is printed starting at first column on next line.

Convert Decimal to Hexadecimal Format

={*val10*} Return

Converts decimal number entered to hexadecimal number (10-digit decimal number maximum). Result is printed starting at first column on next line. Entries may be signed (+/−) or unsigned.

Jump to Cold Start

Control-B

Unconditionally jumps to BASIC's cold-start routine at ROM location $E000.

Jump to Warm Start

Control-C

Unconditionally jumps to BASIC's warm-start routine at ROM location $E003.

Jump to User Vector

Control-Y

Unconditionally jumps to user vector at $03F8.

Quit Monitor, Q

Q

Discontinues Monitor operation. Unconditionally jumps to $3D0 to warm-start the operating system.

Run a Program in Bank $00, G

{*start_address*}G

Transfers control to the machine-language program beginning at {*start_address*}. Sets the environment from stored locations A/X/Y/S/D/P/B/K/M/Q/L/m/x/e; pushes RTS information on the user's stack and performs a JMP to {*start_address*} with RTS information left on the stack (only works for code in bank $00 because it assumes user's routine ends in an RTS).

Reset the Environment and Transfer Control, X (Execute)

{*start_address*}X

Retrieves A/X/Y/S/D/P/B/K/M/Q/L/m/x/e data from stored locations, sets those data as the environment, pushes RTL information on the user's stack, and performs a JMP to {*start_address*} with RTL information on the stack (works for code in any bank; assumes user's code ends in an RTL).

Restore Registers and Flags

Control-R

Restores registers and flags to the normal Monitor configuration mode. Changes A/X/Y/S/D/P/B/K/M/Q/L/m/x/e.

Reset the Environment and Transfer Control, R (Resume)

{*start_address*}R

Sets the environment from stored locations A/X/Y/S/D/P/B/K/M/Q/L/m/x/e and JMPs to {*start_address*}.

Perform a Program Step, S

{*start_address*}S

Not implemented in current version.

Perform a Program Trace, T

{*start_address*}T

Not implemented in current version.

Disassemble, L (List)

{*start_address*}L

Disassembles up to 20 instructions starting at location {*start_address*}.

Tool Locator, U

\#*bytes to stk*_#*bytes frm stk*_*parm1*_. . .*parmz*_*function#*_*tool#*\U

The underline character (_) indicates where spaces must be placed.

#*bytes to stk* indicates the number of parameters that need to be pushed onto the stack to make the utility call to the specified tool.

#*bytes frm stk* indicates the number of parameters the function pushes onto the stack. That many bytes will be pulled from the stack and displayed at the end of the call.

*parm1*_. . .*parmz* indicates the parameters to push onto the stack before making the Tool Locator call. Parameters must be single-byte values. For example, to enter a 4-byte address, type `00` *bb hh* `11`, where

`00` = null byte of address (space required after byte)
bb = bank number of address (space required after byte)
hh = high byte of address (space required after byte)
`11` = low byte of address space (space required after byte, before next parameter)

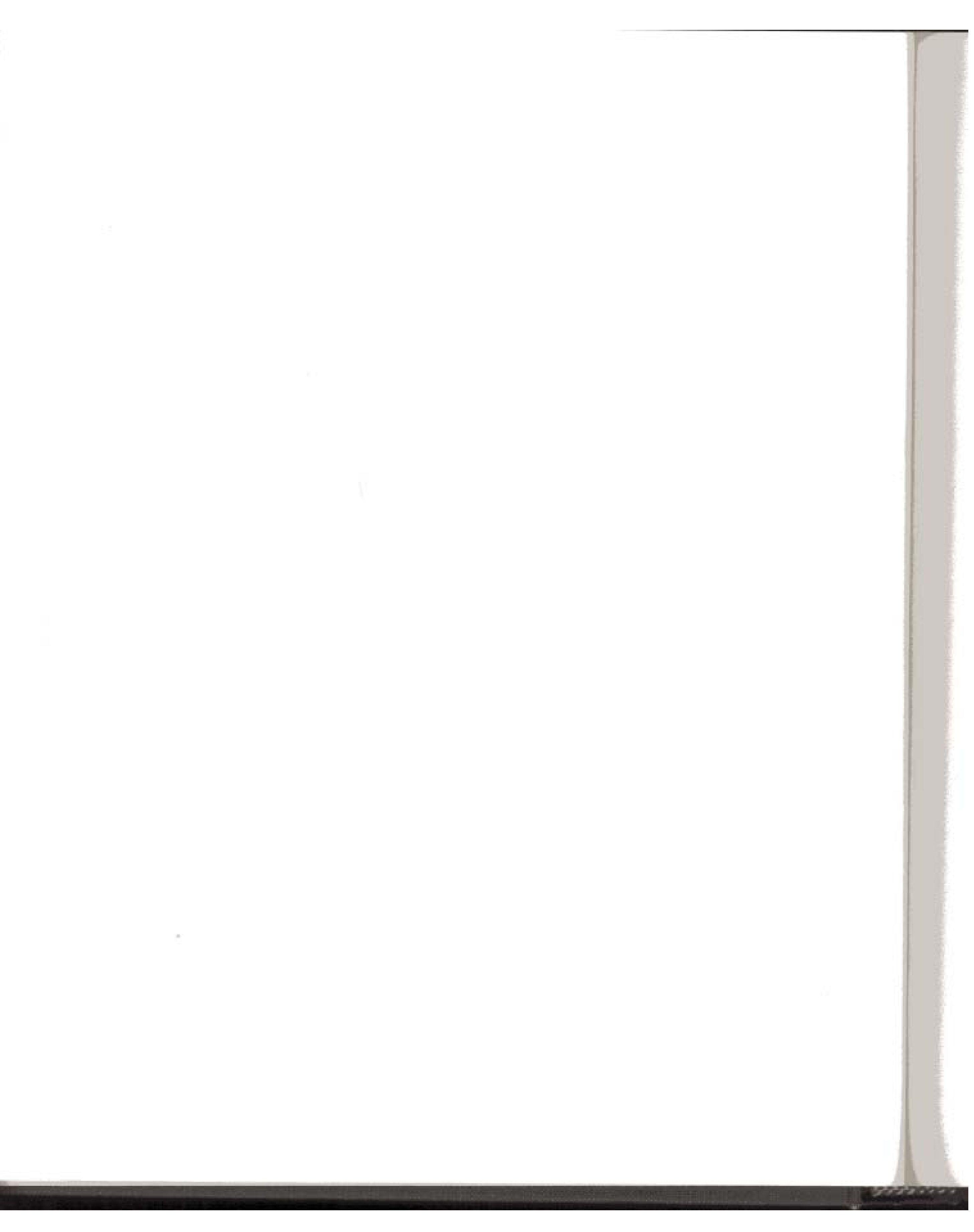
To enter multiple ASCII bytes, type `'W'`, `'X'` or `"W"`, `"X"`, using either single or double quotation marks. Each ASCII byte is a parameter and so must be separated with a space.

function# indicates the function number to be called in the specified tool.

tool# indicates the tool number to be called by utility call.

The function numbers and tool numbers are listed in the *Apple IIGS Toolbox Reference*.

A tool error number is always printed along with parameters left on the stack after the tool is called. The format of the error printout is `Tool error =` *eeee*, where *eeee* is the value of the accumulator (error) after the tool call. On errors $0001–$000F, the U command removes and displays exactly the number of bytes it pushed onto the stack before the call. For errors >$000F, no parameters are left on the stack, so none are displayed.



Chapter 4

Video Firmware

This chapter describes the routines and command sequences that you use to control the video output of text to the Apple IIGS video screen. The Apple IIGS video firmware includes routines for text input and output. These routines are used by high-level languages, but can just as easily be called directly from a routine that you have written using the mini-assembler. Almost every program on the Apple IIGS takes input from the keyboard or mouse and sends output to the display. The Monitor and BASIC accept keyboard input and produce screen output by using standard input/output (I/O) subroutines built into the Apple IIGS firmware.

Using the video firmware I/O routines, you can

- read keys individually from the keyboard
- read an entire line of key entries
- send characters to the firmware output routines
- call built-in routines that control the video display

When you call a routine to get an entire line, the user has the opportunity to use the Backspace key and other onscreen editing facilities before your routine sees the line. When you send characters to the firmware output routines, most of the characters are transmitted to the display. However, some of the characters control the display subsystem. These special characters are listed in Tables 4-1, 4-3, and 4-4.

Standard I/O links

When you call one of the character I/O subroutines (COUT and RDKEY), the video firmware performs an indirect jump to an address stored in programmable memory. Memory locations used for transferring control to other subroutines are sometimes called *vectors;* in this manual, the locations used for transferring control to the I/O subroutines are called *I/O links*. In an Apple IIGS running without a disk, each I/O link normally contains the address of the body of the subroutine (COUT1 or KEYIN) that the firmware calls for that specific form of I/O. If a disk operating system is running, one or both of these links holds the address of the corresponding DOS or ProDOS I/O routines instead of the firmware default values. (DOS and ProDOS maintain their own links to the standard I/O subroutines.)

By calling the I/O subroutines that jump to the link addresses instead of calling the standard subroutines directly, you ensure that your program will work properly with other software, such as DOS or a printer driver, that changes one or both of the I/O links.

For the purposes of this chapter, we shall assume that the I/O links contain the addresses of the standard I/O subroutines: COUT1 and KEYIN if the 80-column firmware is disabled, and BASICOUT (also called C3COUT1) and BASICIN if the 80-column firmware is enabled.

Standard input routines

The Apple IIGS firmware includes three different subroutines for reading from the keyboard. These subroutines are written to function at different levels. The character input subroutine KEYIN (or BASICIN when the 80-column firmware is active) accepts one character at a time from the keyboard. The RDKEY subroutine (short for *readkey*) calls KEYIN or BASICIN and handles the onscreen cursor. The third subroutine is named *GETLN*, which stands for *get line*. By making repeated calls to RDKEY, GETLN accepts a sequence of characters terminated with a carriage return. GETLN also provides onscreen editing features.

RDKEY input subroutine

Your program gets a character from the keyboard by making a subroutine call to RDKEY at memory location $FD0C. RDKEY sets the character at the cursor position to flash and then passes control through the input link KSW to the current input subroutine, which is normally KEYIN or BASICIN.

RDKEY produces a cursor at the current cursor position, immediately to the right of the character you last sent to the display (normally by using the COUT routine). The cursor displayed by RDKEY is a flashing version of the character that happens to be at that position on the screen. Usually, a user types new characters on a blank line, so the next character will normally be a space. Thus, the cursor appears as a blinking rectangle.

KEYIN and BASICIN input subroutines

Apple IIGS supports 40- and 80-column video displays by using input subroutines KEYIN and BASICIN. The KEYIN subroutine is used when the 80-column firmware is inactive; BASICIN is used when the 80-column firmware is active. When called, the subroutine waits until the user presses a key and then returns with the key code in the accumulator.

If the 80-column firmware is inactive, KEYIN displays a cursor by storing a checkerboard block in the cursor location, then storing the original character, and then storing the checkerboard again. If the 80-column firmware is active, BASICIN displays a steady inverse space (rectangle) as a cursor. In an additional operating mode, escape mode, the cursor displayed is an inverse video plus sign (+). This indicates that escape mode is active. See the section "Cursor Control" later in this chapter for more information about the escape mode.

Subroutine KEYIN also generates a random number. While it is waiting for the user to press a key, KEYIN repeatedly increments the 16-bit number in memory locations 78 and 79 (hexadecimal $4E and $4F). This number continues to increase from 0 to 65535 and then starts over again at 0. The value of this number changes so rapidly that there is no way to predict what it will be after a key is pressed. A program that reads from the keyboard can use this value as a random number or as a seed for a random-number generator.

When the user presses a key, KEYIN accepts the character, stops displaying the cursor, and returns to the calling program with the character in the accumulator.

Escape codes

Subroutine KEYIN has special functions that you invoke by typing escape codes at the keyboard. An escape code is obtained by pressing the Esc (Escape) key, releasing it, and then pressing another key. The key sequences shown are not case sensitive. That is, Esc followed by *A* (uppercase) is equivalent to Esc followed by *a* (lowercase).

Escape codes are used to clear the current line, the rest of the screen, or the whole screen; to switch from 40-column to 80-column mode and vice versa; and to move the cursor on the screen. The escape codes that KEYIN follows are listed in Table 4-1.

Cursor control

The Apple IIGS is equipped with four arrow keys. However, these keys do not perform cursor-movement functions unless the system is specifically told to give them such functions. The Apple IIGS firmware provides what is called the *escape mode*, which activates the arrow keys for cursor moves. One of eight possible escape sequences can be used to activate the escape mode. As Table 4-1 shows, you can enter escape mode by pressing Esc followed by an alphabetic key or by pressing Esc followed by one of the four arrow keys. Recall also that when the 80-column firmware is active, the cursor display changes to a plus sign (+) when the system is operating in escape mode.

You can continue to use the arrow keys to move around on the screen. As noted in the table, escape mode terminates when anything other than an arrow key is pressed.

Table 4-1
Escape codes and their functions

Escape code	Function
Cursor control	
Esc A	Moves cursor right one space; exits from escape mode
Esc B	Moves cursor left one space; exits from escape mode
Esc C	Moves cursor down one line; exits from escape mode
Esc D	Moves cursor up one line; exits from escape mode
Cursor control/ entering escape mode	
Esc I (or Esc Up Arrow)	Moves cursor up one line and remains in escape mode
Esc J (or Esc Left Arrow)	Moves cursor left one space and remains in escape mode
Esc K (or Esc Right Arrow)	Moves cursor right one space and remains in escape mode
Esc M (or Esc Down Arrow)	Moves cursor down one line and remains in escape mode
Screen/line clearing	
Esc @	Clears window and moves cursor to its home position (upper-left corner of screen); exits from escape mode
Esc E	Clears to end of line; exits from escape mode
Esc F	Clears to bottom of window; exits from escape mode
Screen format control	
Esc 4	Switches from 80-column display to 40-column display if 80-column firmware is active, sets links to BASICIN and BASICOUT, restores normal window size; exits from escape mode
Esc 8	Switches from 40-column display to 80-column display by enabling 80-column firmware, sets links to BASICIN and BASICOUT, restores normal window size; exits from escape mode
Esc-Control-D	Disables control characters; only carriage returns, line feeds, bells, and backspaces have effects when printing is performed
Esc-Control-E	Reactivates control characters
Esc-Control-Q	If 80-column firmware is active, deactivates 80-column firmware, sets links to KEYIN and COUT1, restores normal window size, exits from escape mode

GETLN input subroutine

Programs often need strings of characters as input. Although you can call RDKEY repeatedly to get several characters from the keyboard, there is a more powerful subroutine you can use to get an edited line of characters. This routine is named *GETLN*, which stands for *get line*; GETLN starts at location $FD6A. Using repeated calls to RDKEY, GETLN accepts characters from the standard input subroutine—usually KEYIN—and puts them into the input buffer located in the memory page from $200 to $2FF. GETLN also provides the user with onscreen editing and control features. These are described in the next section, "Editing With GETLN."

GETLN displays a prompting character, called a *prompt*. The prompt indicates to the user that the program is waiting for input. Different programs use different prompt characters to help remind the user which program is requesting input. For example, an INPUT statement in a BASIC program displays a question mark (?) as a prompt. The prompt characters used by Apple IIGS programs are shown in Table 4-2.

GETLN uses the character stored at location 51 (hexadecimal $33) as the prompt character. In an assembly-language program, you can change the prompt to any character that you wish. In BASIC or in the Monitor, changing the prompt character has no effect because both BASIC and the Monitor restore the prompt to their original choices each time they request user input.

Table 4-2
Prompt characters

Prompt character	Program requesting input
?	User's BASIC program (INPUT statement)
]	Applesoft BASIC
>	Integer BASIC
*	Monitor
!	Mini-assembler

As you type an input character string, GETLN sends each character to the standard output routine, normally COUT1, which displays the character at the previous cursor position and puts the cursor at the next available position on the display, usually immediately to the right of the original position. As the cursor travels across the display, it indicates the position where the next character will be displayed.

GETLN stores the characters in its buffer, starting at memory location $200 and using the X register to index the buffer. GETLN continues to accept and display characters until you press Return. Then it clears the remainder of the line the cursor is on, stores the carriage return code in the buffer, sends the carriage return code to the display, and returns to the calling program.

The maximum line length that GETLN can handle is 255 characters. If the user types more than 255 characters, GETLN sends a backslash (\) and a carriage return to the display, cancels the line it has accepted so far, and starts over. To warn the user that the line is getting full, GETLN sounds a bell (tone) at every keypress after the 248th.

Editing with GETLN

The subroutine GETLN provides the standard onscreen editing features used with BASIC interpreters and the Monitor. Any program that uses GETLN for reading the keyboard offers these features. For an introduction to editing with GETLN, refer to the *Applesoft Tutorial*.

Cancel line: Any time you are typing a line, pressing Control-X causes GETLN to cancel the line. GETLN displays a backslash (\) and issues a carriage return and then displays the prompt and waits for you to type a new line. GETLN automatically cancels the line when you type more than 255 characters, as described earlier.

Backspace: When you press the Backspace key, the Back Arrow key (←), or the Delete key, GETLN moves its buffer pointer back one space, deleting the last character in its buffer. It also sends a backspace character to the routine COUT, which moves the display position back one space. If you type another character now, it will replace the character you backspaced over, both on the display and in the line buffer. Each time you press the Backspace key, the cursor moves left and deletes another character until you reach the beginning of the line. If you then press Backspace one more time, you cancel the line. If the line is canceled this way, GETLN issues a carriage return and displays the prompt.

Retype: The function of the Retype key (→) is complementary to the function of the Backspace key. When you press Retype, GETLN picks up the character at the display position just as if it had been typed on the keyboard. You can use this procedure to pick up characters that you have just deleted by backspacing across them. You can use the backspace and retype functions with the cursor-motion functions to edit data on the display. For more information about cursor motion, see the section "Cursor Control" earlier in this chapter.

Keyboard input buffering

In versions of the Apple II prior to the Apple IIGS, if a user pressed a key while a program was processing the previous keystroke, characters that the user was typing into the program were in danger of being lost. The Apple IIGS allows you to use keyboard input buffering to prevent the loss of keystrokes.

The user can select keyboard input buffering through the Control Panel program. If the Event Manager is enabled, the type-ahead buffer can process an unlimited number of key presses.

Standard output routines

The Monitor firmware output routine is named *COUT* (pronounced *C-out*), which stands for *character out*. The COUT routine normally calls COUT1, which sends one character to the display, advances the cursor position, and scrolls the display when necessary. The COUT1 routine restricts its use of the display to an active area called the *text window*, described later in this chapter.

BASICOUT is used instead of COUT1 when the 80-column firmware is active. Subroutine BASICOUT is essentially the same as COUT1: BASICOUT displays the character in the accumulator on the display screen at the current cursor position and advances the cursor. When BASICOUT returns control to the calling program, all registers are intact.

COUT and BASICOUT subroutines

When you call COUT (or BASICOUT) and send a character to COUT1, the character is displayed at the current cursor position, replacing whatever was there. COUT1 then advances the cursor position one space to the right. If the cursor position is at the right edge of the window, COUT1 moves the cursor to the leftmost position on the next line down. If this moves the cursor past the end of the last line in the window, COUT1 scrolls the display up one line and sets the cursor position at the left end of the new bottom line.

The cursor position is controlled by the values in memory locations 36 and 37 (hexadecimal $24 and $25). Subroutine COUT1 does not display a cursor, but the input routines COUT1 and C3COUT1, described in the next section, do display and use a cursor. If another routine displays a cursor, that routine will not necessarily put the character in the cursor position used by COUT1.

Control characters with COUT1 and C3COUT1

Subroutine COUT1 is the entry point that is active for character output in 40-column mode. Entry point C3COUT1 is active when the system is in 80-column mode. Subroutines COUT1 and C3COUT1 do not display control characters. Instead, the control characters listed in Tables 4-3 and 4-4 are used to initiate action by the firmware. Other control characters are ignored. Most of the functions listed here can also be invoked from the keyboard, either by typing the control character listed or by using the appropriate escape code, as described in the section "Escape Codes" earlier in this chapter.

Table 4-3
Control characters with 80-column firmware off

Control character	Action taken by COUT1
Control-G	Produces user-defined tone (Control Panel menu)
Control-H	Causes backspace
Control-J	Causes line feed
Control-M	Causes carriage return
Control-^ {*char*}	First character output after Control-^ becomes new cursor. If Delete key is first character, default prompt is restored.

Table 4-4
Control characters with 80-column firmware on

Control character	Action taken by C3COUT1
Control-E	Turns cursor off
Control-F	Turns cursor on
Control-G	Produces user-defined tone (Control Panel menu)
Control-H	Causes backspace
Control-J	Causes line feed
Control-K	Clears from cursor position to end of screen
Control-L	Causes form feed
Control-M	Causes carriage return
Control-N	Changes to normal display format
Control-O	Changes to inverse display format
Control-Q	Sets 40-column display
Control-R	Sets 80-column display
Control-S	Stops listing of characters until another key is pressed
Control-U	Deactivates enhanced video firmware
Control-V	Scrolls display down one line, leaving cursor in current position
Control-W	Scrolls display up one line, leaving cursor in current position
Control-X	Disables MouseText character display and uses inverse uppercase characters
Control-Y	Homes cursor to upper-left corner
Control-Z	Clears line on which cursor resides
Control-[	Enables MouseText character display by mapping inverse uppercase characters to MouseText characters
Control-\	Moves cursor position one space to right; from edge of window, moves to left end of next line
Control-]	Clears from cursor position to right end of line
Control-_	Moves cursor up one line with no scroll
Control-^	Goes to XY; using next two characters minus 32 as 1-byte X and Y values, moves cursor to CH=X, CV=Y (Pascal)
Control-^ {*char*}	First character output after Control-^ becomes new cursor. If Delete key is first character, default prompt is restored. This works only when using BASIC links, not Pascal output links.

Inverse and flashing text

Subroutine COUT1 can display text in normal format, inverse format, or with some restrictions flashing format. The display format for any character in the display depends on two factors: the character set currently being used and the setting of the two high-order bits of the character's byte in the display memory.

As it sends your text characters to the display, COUT1 sets the high-order bits according to the value stored at memory location 50 (hexadecimal $32). If that value is 255 (hexadecimal $FF), COUT1 sets the character display to normal format. If that value is 63 (hexadecimal $3F), COUT1 sets the character display to inverse format. If the value is 127 (hexadecimal $7F) and if you have selected the primary character set, the characters will be displayed in flashing format. Note that the flashing format is not available in the alternate character set. Table 4-5 shows the effect of the mask value on particular parts of the character set.

Table 4-5
Text format control values

Mask (dec)	Value (hex)	Display format
255	$FF	Normal, uppercase, and lowercase
127	$7F	Flashing, uppercase, and symbols
63	$3F	Inverse, uppercase, and lowercase

To control the display format of the characters, routine COUT1 uses the value at location 50 as a logical mask to force the setting of the two high-order bits of each character byte it puts into the display page. It does this by performing a logical AND operation on the data byte and mask byte. The resulting byte contains a 0 in any bit that was a 0 in the mask. BASICOUT, used when the 80-column firmware is active, changes only the high-order data bit.

❖ *Note:* If the 80-column firmware is inactive and you store a mask value at location 50 with zeros in its low-order bits, COUT1 will mask those bits in your text. As a result, some characters will be transformed into other characters. You should set the mask values only to those given in Table 4-5.

If you set the mask value at location 50 to 127 (hexadecimal $7F), the high-order bit of each resulting byte will be 0 and the characters will be displayed either as lowercase or flashing, depending on which character set you selected. In the primary character set, the next-highest bit, bit 6, selects flashing format with uppercase characters. With the primary character set, you can display lowercase characters in normal format and uppercase characters in normal, inverse, and flashing formats. In the alternate character set, bit 6 selects lowercase or special characters. With the alternate character set, you can display uppercase and lowercase characters in normal and inverse formats.

Other firmware I/O routines

In addition to the read and write character routines described above, the Apple IIGS firmware also includes several routines that provide convenient screen-oriented I/O functions. These functions are listed in Table 4-6 and are described in detail in Appendix C, "Firmware Entry Points in Bank $00."

Important

Appendix C is the official list of all entry points that are currently valid and for which continued support will be provided in future revisions of this product.

Table 4-6
Partial list of other Monitor firmware I/O routines

Location	Name	Description
$FC9C	CLREOL	Clears to end of line from current cursor position
$FC9E	CLEOLZ	Clears to end of line using contents of Y register as cursor position
$FC42	CLREOP	Clears to bottom of window
$F832	CLRSCR	Clears low-resolution screen
$F836	CLRTOP	Clears top 40 lines of low-resolution screen
$FDED	COUT	Calls output routine whose address is stored in CSW (normally COUT1)
$FDF0	COUT1	Displays character on screen
$FD8E	CROUT	Generates carriage return
$FD8B	CROUT1	Clears to end of line and then generates carriage return
$FD6A	GETLN	Displays prompt character; accepts string of characters by means of RDKEY
$F819	HLINE	Draws horizontal line of blocks
$FC58	HOME	Clears window and puts cursor in upper-left corner of window
$FD1B	KEYIN	With 80-column firmware inactive, displays checkerboard cursor; accepts characters from keyboard
$F800	PLOT	Plots single low-resolution block on screen
$F94A	PRBL2	Sends 1 to 256 blank spaces to output device
$FDDA	PRBYTE	Prints hexadecimal byte
$FDE3	PRHEX	Prints 4 bits as hexadecimal number
$F941	PRNTAX	Prints contents of A and X in hexadecimal format
$FD0C	RDKEY	Displays blinking cursor; goes to standard input routine (normally KEYIN or BASICIN)
$F871	SCRN	Reads color of low-resolution block
$F864	SETCOL	Sets color for plotting in low-resolution block
$FC24	VTABZ	Sets cursor vertical position
$F828	VLINE	Draws vertical line of low-resolution blocks

The text window

After starting the computer or after a reset operation, the firmware uses the entire display for text. However, you can restrict text video activity to any rectangular portion of the display that you wish. The active portion of the display is called the *text window.* COUT1 (or BASICOUT) puts characters into the window only; when it reaches the end of the last line in the window, it scrolls only the contents of the window.

You can control the amount of the screen that the video firmware reserves for text by modifying memory directly. You can set the top, bottom, left side, and width of the text window by storing the appropriate values in four locations in memory. This allows your programs to control the placement of text in the display and to protect other portions of the screen from being overwritten by new text.

Memory location 32 (hexadecimal $20) contains the number of the leftmost column in the text window. This number normally is 0, the number of the leftmost column of the display. In a 40-column display, the maximum value for this number is 39 (hexadecimal $27); in an 80-column display, the maximum value is 79 (hexadecimal $4F).

Memory location 33 (hexadecimal $21) holds the width of the text window. For a 40-column display, the width normally is 40 (hexadecimal $28); for an 80-column display, it normally is 80 (hexadecimal $50).

Memory location 34 (hexadecimal $22) contains the number of the top line of the text window. This normally is 0, the topmost line in the display. Its maximum value is 23 (hexadecimal $17).

Memory location 35 (hexadecimal $23) contains the number of the bottom line of the screen. Its normal value is 24 (hexadecimal $18), the bottom line of the display. Its minimum value is 1.

After you have changed the text window boundaries, the appearance of the screen will not change until you send the next character to the screen.

Chapter 5

Serial-Port Firmware

This chapter covers the features of the serial communications firmware. The Apple IIGS serial-port firmware provides serial communications for external devices, such as printers and modems. The Apple IIGS serial-port firmware uses a two-channel Zilog Serial Communications Controller chip (SCC8530) and RS-422 drivers. The driver firmware emulates the functionality of the Apple Super Serial Card (SSC) and supports input/output buffering as well as background printing. The firmware also implements a number of calls that the application can make to control the new features.

Input/output buffering and background printing are done on an interrupt basis and can use any buffer size up to 64K at any location that the application wishes. I/O buffering is transparent for BASIC and Pascal. An application can make a function call that starts background printing. The function call copies the data into the background printing buffer and then returns control to the application. Data is fed to the printer automatically until the entire contents of the buffer have been sent to the printer.

Note that AppleTalk, when active, requires the use of one of the two available serial channels. Therefore, only two of these three—AppleTalk, serial port 1, and serial port 2—are allowed to be active at any one time. The Control Panel program ensures that at least one serial port is made inactive when AppleTalk has been selected. You can't initialize the serial-port firmware when the channel is being used by AppleTalk. Both port 1 and port 2 can be configured as either printer or communications (modem) ports.

You can set default parameters for the serial ports through the Control Panel firmware. The application program can temporarily change the parameter values by sending control sequences to the serial-port firmware.

Compatibility

The commands used to communicate with the serial-port firmware are essentially the same as those used with the SSC. However, many existing programs using these ports are not compatible with the Apple IIGS. Many programs, particularly communications packages, send their output directly to the hardware; the Apple IIGS hardware no longer uses hardware different from that used on the SSC. Print programs and applications written in BASIC and Pascal are more likely to work.

One other difference between the Apple IIGS serial-port firmware and other serial-port firmware is in error handling. In the SSC, as well as in the Apple IIc firmware, when a character with an error is received, the character in error is not deleted from the input stream. The Apple IIGS firmware does delete the character from the input stream and sets a bit to record the fact that an error was encountered.

Operating modes

The serial-port firmware has three main operating modes: printer mode, communications mode, and terminal mode. You set these modes through the Control Panel. An application program can change these modes by sending command sequences to the serial port.

❖ *Note:* If you are writing software that depends on the serial-port firmware being in a given operating mode, make sure that your documentation tells the user to set up the firmware using the Control Panel in the proper way.

Printer mode

When in printer mode, the serial-port firmware can send data to a printer, a local terminal, or some other serial device.

Communications mode

When in communications mode, the firmware can operate with a modem. From BASIC, while the serial firmware is set for communications mode, the firmware can enter a special mode, called *terminal mode,* in which the Apple IIGS acts like an unintelligent terminal.

Terminal mode

In terminal mode, the Apple IIGS acts like an unintelligent terminal. All the characters typed are passed to the serial output (except the command strings), and all serial input goes directly to the screen.

You enter terminal mode from the BASIC interface by typing `IN#n` and then typing the current command character followed by a `T`. The prompt character changes to a flashing underline (_), indicating that terminal mode is active. You exit terminal mode by typing the current command character followed by a `Q`.

You can use terminal mode with buffering enabled. This minimizes character loss at higher baud rates. Enable buffering with the Buffering Enable (BE) serial command, described below.

Many remote computers send a line feed (LF) after a carriage return (CR). When using terminal mode with such a computer, use the Masking Enable (ME) serial control command to mask any line feeds that immediately follow carriage returns.

Handshaking

Communications-equipment manufacturers have devised a variety of handshaking schemes. Apple IIGS accommodates these various schemes by providing several hardware and software handshaking options.

Hardware, DTR and DSR

When the DTR/DSR option is active, the data terminal ready (DTR) and data set ready (DSR) lines control the data flow into and out of the system. The Apple IIGS transmits characters only when the DSR line is enabled; the DTR line tells the device when the host is ready to accept data. The default Control Panel setting enables hardware handshaking. If this option is disabled, the DSR line is not checked on transmission and the DTR line will not be toggled during reception (see Figures 5-1 and 5-2). The target device's firmware determines whether these lines mean anything during data transfer.

The data carrier detect (DCD) line controls modem communications. If you enable the DCD handshake option, the Apple IIGS serial-port firmware will transmit characters only when the DCD line is enabled. The DCD option has no direct effect on character reception. This mode provides compatibility with the SSC, which uses DCD as a handshake line.

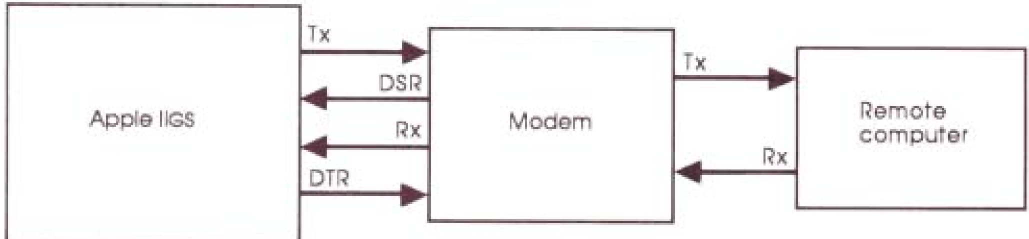


Figure 5-1
Handshaking when DTR/DSR option is turned on

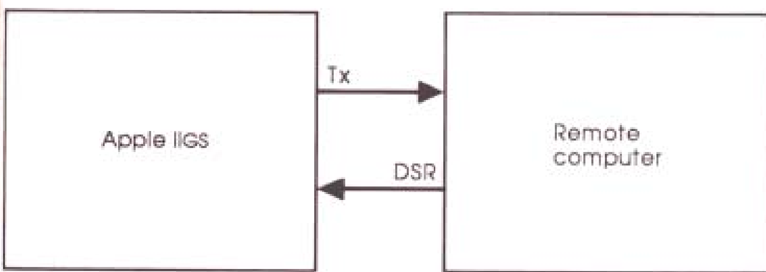


Figure 5-2
Handshaking when DTR/DSR option is turned off

Software, XON and XOFF

If an XOFF ($13) character is received from a device attached to the SCC, the firmware halts character transmission until an XON ($11) character is received. This option works in addition to the hardware handshake. In printer mode, the firmware disables this function.

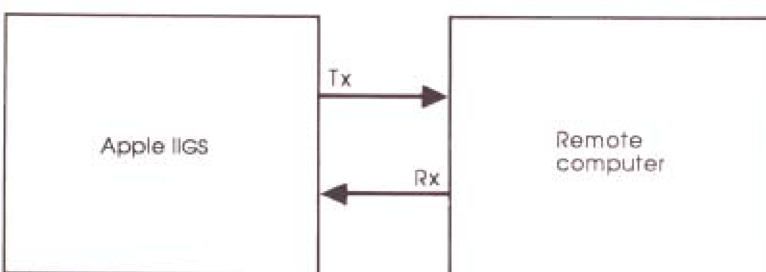


Figure 5-3
Handshaking via XON/XOFF

Operating commands

Apple IIGS control commands, embedded in the serial output flow, are invoked by BASIC or Pascal output routines. For each of the operating modes (printer or communications), you can control many aspects of your data transmissions, such as baud rate, data format, and line-feed generation, by sending control codes as commands to the firmware. All commands are preceded by a command character and optionally followed by a return character ($0D). The carriage return is allowed to maintain compatibility with the SSC. The format of the commands is as follows:

{*command-character*} {*command-string*} Return

The command character usually is Control-I in printer mode and Control-A in communications and terminal modes. In the examples in the following text, Control-I is used unless the command being described is available only in communications mode or terminal mode. A return character is represented by its ASCII symbol, CR.

There are three types of command formats:

- a number, represented by n, followed by an uppercase letter with no space between the characters (for example, `4D` to set data format 4)
- an uppercase letter by itself (for example, `R` to reset the serial-port firmware)
- an uppercase letter followed by either a space or no space and then either `E` to enable or `D` to disable a feature (for example, `LD` to disable automatic insertion of line-feed characters)

The allowable range of n is given in each command description that follows.

❖ *Note:* All options, such as baud rate, parity, and line length, can be configured from the Control Panel (see Chapter 10, "Mouse Firmware").

Serial-port firmware must be reinitialized after changing options from the Control Panel for the new values to take effect.

The command character

The normal command character is Control-I (ASCII $09) in printer mode and Control-A (ASCII $01) in communications mode. If you want to change the command character from Control-I to another command character (for example, Control-W), send Control-W to Control-I. To change back, send Control-I to Control-W. No return character is required after either of these commands.

❖ *Note:* The SSC allows you to send the current command character through the output stream by sending the character twice in a row. The Apple IIGS does not allow this; the character will not be output. To send the command character through the serial port, you must temporarily change to an alternate command character. For example, if the current command character is Control-I and you want to send a Control-I out the serial port, then send

Control-I Control-A Control-I Control-A Control-I

The first two characters change the command character to a Control-A. The third character is the Control-I you wanted to send. The fourth and fifth characters restore the command character to Control-I again. Remember, though, that you can disable all command-character parsing by using the Zap command.

To generate this command character in Applesoft BASIC, enter

```
PRINT CHR$(9); "command-string"
```

For Pascal, enter

```
WRITELN (CHR(9),   'command-string');
```

The following example shows how to generate the command from a BASIC program:

```
10   D$ = CHR$(4):        REM  Sends Control-D
20   A$ = CHR$(9):        REM  Sends Control-I
30   PRINT D$; "PR#1":    REM  Establishes link: BASIC to port 1
40   PRINT A$; "6B":      REM  Changes to 300 baud
50   . . .:               REM  Continue program
```

Command strings

A command string is a letter sometimes with a number prefix and sometimes with an E or a D suffix. Command strings select the option to be used; for instance, they may change the baud rate, select the data format, and set the parity. The preceding example shows commands generated in BASIC; the command strings in the following sections are generated from the keyboard.

Commands useful in printer and communications modes

The following commands are most useful in printer and communications modes.

Baud rate, nB

You can use the nB command to select the baud rate for the serial-port firmware. For example, to change the baud rate to 135, send Control-I 4B CR to the serial-port firmware (see Table 5-1).

Table 5-1
Baud-rate selections

n	Baud rate	n	Baud rate
0	Default*	8	1200
1	50	9	1800
2	75	10	2400
3	110	11	3600
4	134.5	12	4800
5	150	13	7200
6	300	14	9600
7	600	15	19,200

* You set the default by using the Control Panel.

Data format, nD

You can override the Control Panel setting that specifies the data format by using the nD command. Table 5-2 shows how many data bits and stop bits correspond to each value of n. For example, Control-I 2D makes the serial-port firmware transmit each character in the form of 1 start bit (always transmitted), 6 data bits, and 1 stop bit.

Table 5-2
Data-format selections

n	Data bits	Stop bits
0	8	1
1	7	1
2	6	1
3	5	1
4	8	2
5	7	2
6	6	2
7	5	2

Parity, nP

You can use the nP command to set the parity that you want to use for data transmission and reception. Four parity options are available. These are listed in Table 5-3.

Table 5-3
Parity selections

n	Parity value
0	None (default value)
1	Odd
2	None
3	Even

❖ *Note:* The SCC 8530 does not support MARK and SPACE parity.

Line length, nN

The line length is set by sending Control-I nN. The number n can be in the range of 1 to 255 characters. For example, if you send Control-I 75N, the line length is set to 75 characters. (*Note:* Use the C command, discussed next, to enable line formatting.) If you set n to 0, formatting is disabled.

Enable line formatting, CE and CD

A forced carriage return is invoked after a lineful of characters by sending Control-I CE. For example, Control-I 75N (see "Line Length" above) and Control-I CE cause a forced carriage return after 75 characters are typed on a line.

Handshaking protocol, XE and XD

Sending Control-I XE CR or Control-I XD CR to the serial-port firmware determines whether the firmware looks for any XOFF ($13) character coming from a device attached to the SCC. It responds by halting transmission of characters until the serial-port firmware receives an XON ($11) character from the device, signaling the SCC to continue transmission. In printer mode, this function normally is disabled.

XE = Detect XOFF, await XON.
XD = Ignore XOFF.

Keyboard input, FE and FD

The FD command is used to make the serial-port firmware ignore keyboard input. For example, you can include Control-I FD CR in a program, followed by a routine that retrieves data through the serial-port firmware, followed by Control-I FE CR to turn the keyboard back on. As a default, the serial-port firmware keyboard input is enabled.

FE = Insert keystrokes into serial-port firmware input stream.
FD = Disable keyboard input.

Automatic line feed, LE and LD

The automatic line-feed command causes the serial-port firmware to generate and transmit a line-feed character after each return character. For example, Control-I LE CR to print listings or double-spaced text.

LE = Add line feeds after each carriage return output.
LD = Do not add line feeds after carriage return output.

Reset the serial-port firmware, R

The R command resets the serial-port firmware, cancels all previous commands to the serial-port firmware and reinstalls the Control Panel default settings. Sending Control-I R CR to the serial-port firmware has the same effect as sending PR#0 and N#0 to a BASIC program and then resetting the serial-port firmware. This call also relinquishes any memory obtained from the Memory Manager for buffering purposes.

Suppress control characters, Z

The Z command causes all further commands to be ignored. This command is useful when the data you are transmitting (for instance, graphics data) contains bit patterns that the serial-port firmware could mistake for control characters.

Sending Control-I Z CR to the serial-port firmware prevents the firmware from recognizing any further control characters, whether from the keyboard or contained in a stream of characters sent to the serial-port firmware. All tabbing and line formatting are disabled after a Control-I Z command.

Important

The only way to reinstate command recognition after the Z command is either to initialize the serial-port firmware or to use the SetModeBits call described later in this chapter.

Commands useful in communications mode

The following commands are most useful in communications mode.

Echo characters to the screen, EE and ED

The EE and ED commands are used to display (echo) or not to display a character on the video screen during communication. For example, if you send Control-A ED CR, the serial-port firmware disables the forwarding of incoming characters to the screen. This command can be used to hide a password entered at a terminal or to avoid the double display of characters.

EE = Echo input.
ED = Don't echo input.

Mask line feed in, ME and MD

If you send Control-A ME to the serial-port firmware, the firmware will ignore any incoming line-feed character that immediately follows a return character.

Input buffering, BE and BD

The BE and BD commands control input and output communication buffering.

Terminal mode, T and Q

The T command transfers you to terminal mode. In this mode, you can communicate with another computer or a computer time-sharing service. Terminal mode is entered through the BASIC interface. This means that you must initialize the firmware by typing `IN#n` and then sending Control-AT.

❖ *Note:* IN#n sets the port input link, and PR#n sets the port output link. The lowercase n indicates the port number.

To quit terminal mode, send Control-AQ.

Often, when communicating with another computer in terminal mode, you want to send a break character to signal the other computer that you wish to signal the end of the current segment of transmission. To send a break character, send Control-AS CR. This command causes the serial hardware to transmit a 233-millisecond break signal, recognized by most time-sharing systems as a sign-off signal.

Table 5-4 summarizes terminal-mode command characters.

Important

If you enter terminal mode and can't see what you type echoed on the video screen, the modem link may not yet be established or you may need to use the Echo Enable command (Control-A EE).

Table 5-4
Terminal-mode command characters

Character	Description
S	Transmits 233-millisecond break (all zeros)
T	Enters terminal mode
Q	Exits terminal mode

Tab in BASIC, AE and AD

If you send Control-I AE CR to the serial-port firmware, the BASIC horizontal position counter is left equal to the column count. Tabbing initially is disabled. It is up to the program to enable this feature if tabbing is desired.

AE = Implement BASIC tabs.
AD = Do not implement BASIC tabs.

Programming with serial-port firmware

The serial-port firmware provides two interfaces: one for BASIC and one that adheres to the Pascal 1.1 firmware protocol.

❖ *Note:* To use the serial-port firmware, you must set the 65816 data bank register to $00, shift to emulation mode (e bit set to 1), and then issue your call. All entry points are in the $Cn00 space in bank $00. (This applies to all calls to serial-port firmware.)

BASIC interface

The following entry points accommodate the BASIC interface (n is the slot number, which can be 1 or 2):

CnOO BASIC initialization (also outputs character in the accumulator)
$Cn05 BASIC read character (character returned to accumulator; X, Y preserved)
$Cn07 BASIC write character (character passed through accumulator; X, Y preserved)

Although the call to $CN00 coincidentally outputs the character in the accumulator, you should not use this side effect as the standard means of character output. Rather, you should use the $CN07 entry point for output of all but the first character (that is, initialize the serial port only once).

When you type `IN#n` or `PR#n` (set input or output link), BASIC makes a call to $Cn00 after it sets either the KSWL or CSWL link to $Cn00. When the serial-port firmware has control, it alters the links so that they point to the firmware Read and Write routines.

Pascal protocol for assembly language

If you are a machine-language programmer, you should use the Pascal 1.1 protocol to communicate with the serial-port firmware. The Pascal 1.1 protocol interface is more flexible than the BASIC protocol. The Pascal 1.1 protocol uses a branch table in the $Cn00 page to indicate where each of the service routines begins (see Table 5-5).

For example, to reach the Read routine, read the value contained in location $Cn0E (suppose it is $18) and then execute a JSR instruction to the address (for example, $Cn18). Table 5-6 lists the I/O routine offsets and registers.

❖ *Note:* The Pascal interface assumes that the application supplies a line feed after a carriage return, overriding the Control Panel setting. If the application does not supply line feeds, it should send the LE (line-feed generation) call described in the section "Command Strings" earlier in this chapter.

Table 5-5
Service routine descriptions and address offsets

Routine name	Address offset	Description
Initialization	$Cn0D	Reset port, restore Control Panel defaults
Read	$Cn0E	Wait for and get next character
Write	$Cn0F	Send character
Status	$Cn10	Inquire if character has been received
Control	$Cn12	Access extended interface commands

Table 5-6
I/O routine offsets and registers for Pascal 1.1 firmware protocol

Address offset	When used	X register	Y register	A register
$Cn0D	Initialization			
	On entry	$Cn	$n0	
	On exit	Error code	Undefined	Undefined
$Cn0E	Read			
	On entry	$Cn	$n0	
	On exit	Error code	Undefined	Character read
$Cn0F	Write			
	On entry	$Cn	$n0	Character to write
	On exit	Error code	Undefined	Undefined
$Cn10	Status			
	On entry	$Cn	$n0	Request (0 or 1)*
	On exit	Error code	Undefined	Undefined
Extended interface†				
$Cn12	Control	Command list	Command list	Command list
	On entry	address (8..15)	address (16..23)	address (0..7)
	On exit	Undefined	Undefined	Undefined

* Request code 0 means *Are you ready to accept output?* Request code 1 means *Do you have input ready?* On exit, the reply to the status request is in the carry bit, as follows: Carry = 0 means *no;* Carry = 1 means *yes.*

† If the function call returns with the carry bit set, an error is returned in the accumulator. The status call can return a "bad request code" ($40). Result codes returned by the extended interface are as follows:

Error type	Explanation	Error code
No error	No problem detected.	$0000
Bad call Error	Illegal command used.	$0001
Bad parameter count	Parameter count not consistent with command requested.	$0002

Error handling

When the serial-port firmware receives a character from the hardware, it checks the error status register in the SCC. 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 bit-mode bit is set. You can use the GetModeBits call to read these two bits (one for framing errors and the other for parity errors) to determine whether at least one receive error has occurred. After you read these bits, you should clear them (using SetModeBits) so that future errors can be detected. Error checks should be performed periodically so that you will know whether received data is accurate.

Buffering

Input and output communications and background printing can be transparently buffered. Each port has two buffers: one for input and one for output. Default buffers are 2048 characters each. If you wish to use a buffer larger than this, you must pass the address and length to the firmware by way of the extended-interface instruction SetInBuffer or SetOutBuffer. You can allocate up to 64K bytes.

◆ *Note:* In systems with little RAM remaining, you can reduce the size of the I/O buffers to 128 bytes.

You can enable buffering by using the PR#n command from BASIC if the buffering option has been set in the Control Panel. If the buffering option has not been set, you can still enable buffering from the keyboard or by sending the BE command through the output flow. When buffering is enabled for output, characters sent to the firmware are placed in a FIFO (first in, first out) queue in the output buffer. These characters are sent out on an interrupt basis whenever the hardware is ready to send another character.

The XON and XOFF characters are not queued; they are sent directly through the channel so that the data flow to the Apple IIGS may be stopped or restarted immediately. Characters received in the buffering mode are placed in the input queue, and all read calls return characters from the queue. Any XON and XOFF characters received are not queued, so the output flow can be halted or resumed immediately upon reception.

When the input queue becomes more than three-fourths full, the firmware attempts to disable the handshake. The firmware sends an XOFF character (if XON/XOFF handshaking is enabled), or the DTR line is disabled (if DSR/DTR handshaking is enabled). You can determine, through your application program, that the handshake has been disabled by inspecting the input flow mode bit using the GetModeBits call in the extended interface. The firmware reenables the handshake as soon as the receive queue fills less than one-fourth of the input buffer.

You can determine the number of characters in the input queue or the amount of space left in the output queue by using the InQStatus and OutQStatus commands in the extended interface. Also, the InQStatus call returns the amount of time elapsed since the last character was queued. This allows a program to keep track of the input stream activity level even though it is not involved in the interrupt process.

❖ *Note:* The InQStatus elapsed-time counter functions correctly only if the heartbeat interrupt task has been started. The heartbeat interrupt task is a set of functions called by interrupt code that run automatically at one-thirtieth of a second intervals.

Interrupt notification

When a channel has buffering enabled, the firmware services all interrupts that occur on that channel. If an application wishes to service interrupts for a given channel itself, it should disable buffering using the BD command in the output flow. If the buffering mode is off, the serial-port firmware will not process any interrupts. The system interrupt handler will transfer control to the user's interrupt vector as $03FE in bank $00. (This is the ProDOS user's interrupt vector.) The user's interrupt service handler is then completely responsible for all serial-port firmware interrupt service.

If the application does not want to disable buffering, but does wish to be *notified* when a certain type of serial interrupt occurs, it can instruct the firmware to pass control to an application-installed routine after the system has serviced the interrupt. The application tells the firmware when it wishes to be notified and establishes the address of the application's completion routine by using the SetIntInfo routine. (See Chapter 8, "Interrupt-Handler Firmware," for more information about interrupt routines.) This call guarantees that the completion routine will get control when a specific type of interrupt occurs, but only after the serial-port firmware has processed and cleared the interrupt. The application then uses the GetIntInfo routine to determine which interrupt condition occurred.

A terminal emulator offers a typical example of when interrupt notification might be desirable. The emulator usually should perform input and output character buffering, handshaking, and other such operations. The terminal emulator can be designed to allow the firmware to handle all character-buffering details. The designer of the emulator can have the firmware signal the emulator program when the firmware receives a break character. To enable this special-condition notification, the emulator application sets the break interrupt-enable function by using the SetIntInfo routine. Now whenever the firmware receives a break character, the firmware SCC interrupt handler records and clears the interrupt, finally passing control to the emulator's completion routine. This routine calls GetIntInfo, and if the break bit is set, the completion routine knows that a break character has been received.

Note that all interrupt sources (except receive and transmit) cause an interrupt on a *transition* in a given signal. This means that a user's interrupt handler will get control passed to it on both positive and negative transitions in the signals of interest. For example, a break-character sequence causes two interrupts: one at the beginning of the sequence and one at the end. The user's interrupt handler should take this into account. A routine can always determine the current state of the bits of interest by using the GetPortStat routine.

The interrupt completion routine executes as *part of the firmware interrupt handler* and must run in that environment. In addition, the following environment variables must be preserved by your routine:

DBR = $00, e = 0, m = 1, x = 1

Registers A, X, and Y need not be preserved.

Background printing

Apple IIGS allows you to print while running an application program. Printing while another program is running is called **background printing.** Background printing is another example of output buffering, as described in the section on buffering: In background printing, you send a block of characters over a serial channel on an interrupt basis. The major difference is that the firmware is handed a large number of characters to transmit all at once rather than getting them one at a time.

To print in the background, perform the following steps:

1. Issue an Init call through the Pascal interface. This ensures that the firmware and hardware are active. The hardware characteristics (baud rate, data format, and so on) will be as specified in the Control Panel.
2. Disable buffering using the BD serial command in case the Control Panel was set to enable buffering.
3. If you want to change the port characteristics, specify them using either the SetModeBits call or the Send command in the output flow.
4. Set the output buffer using SetOutBuffer. To use the default buffer, make a call to GetOutBuffer to ascertain its location.
5. Load the data into the buffer.
6. Start the printing process with SendQueue, passing the length of the buffer data and the address of the Recharge routine.

Recharge routine

Once you start background printing with a SendQueue call, the firmware sends the characters periodically, in the background, until the buffer is exhausted. When the last character is removed from the buffer, the firmware executes a JSL to the Recharge routine, whose address was passed when the call to the SendQueue routine was made. This application-supplied routine reloads the buffer with the next set of data to be sent, a task that could involve some disk activity if the application is performing background printing from the disk. Finally, the routine loads the number of bytes in the new block of data to be sent to the X and Y registers (these will both be zero in case the background printing is complete) and executes an RTL. Requirements for the Recharge routine are as follows:

On entry	On exit
System speed = fast	System speed = fast
DBR = $00	DBR = $00
Native mode (that is, m = 0, x = 0, e = 0)	Native mode, 8-bit m and x (e = 0)
	X register = data size (low)
	Y register = data size (high)

Note that the Recharge routine is called at interrupt time. Therefore, you should regard it as an interrupt handler, in the sense that anything it changes must be restored. Also note that interrupts are disabled during the time the Recharge routine is running. If too much time is spent in this routine, performance degradation of interrupt-critical processes will occur. An interrupt-critical process is one such as AppleTalk that has stringent interrupt-response requirements.

❖ *Note:* The firmware reserves the last byte in the data buffer for empty buffer detection. Make sure that the buffer's size is 1 byte larger than the amount of data you place in it. For example, if GetOutBuffer reveals an output buffer of 2048 bytes, only data lengths *less than* 2048 should be passed with the background-printing call or Recharge routine.

Extended interface

The Apple IIGS system has extended call features not present in the SSC. These calls are made through the extended interface and are divided into three groups: hardware control, mode control, and buffer-management features. A list of the extended interface calls follows this section.

You can make a call through the extended interface using the following method:

1. Determine the dispatch address by adding the value $CN00 to the value located at $CN12. The byte at $CN12 is called the *optional control routine offset* of the Pascal 1.1 protocol.
2. Perform an emulation-mode JSR (DBR = $00) to this dispatch address, with the address of the command list (CMDLIST) in the appropriate registers as follows:

Register	Register value
A	Address of CMDLIST (low)
X	Address of CMDLIST (medium)
Y	Address of CMDLIST (high)

Every command list starts with a 1-byte *parameter* count (not a *byte* count), a command code, and space for a result code. The possible result codes returned are listed in the section "Error Handling" earlier in this chapter.

❖ *Note:* If you want to ensure that your application will work with future systems, limit the use of hardware control calls, particularly the Get SCC and Set SCC calls. If future systems use hardware other than the current serial chip (SCC 8530), your hardware control calls will most likely have to be changed.

In the extended serial interface descriptions that follow, a DFB is an assembler directive that produces a single byte, a DW is an assembler directive that produces a double byte (16-bits: low byte, high byte), and a DL is an assembler directive that produces a double word (32 bits, that is, 4 bytes).

Important

Different instructions require that a different number of bytes be reserved for the return parameters. Be sure that the CMDLIST buffer area to which you point is large enough to hold all of the bytes of the return parameters for that command. If your buffer area is not large enough, the system may fail.

Mode control calls

GetModeBits

Returns the current mode bit settings.

```
CMDLIST   DFB   $03               ;Parameter count
          DFB   $00               ;Command code
          DW    $00               ;Result code (output)
          DL    $00               ;ModeBitImage (output)
```

This call allows the application to determine the status of various firmware operating modes. Four bytes (32 bits) of mode information are returned. To change any of these bits, use this call to get the current settings, then alter the bits of interest, and then use the SetModeBits call to make the actual modification. (To avoid race conditions in this process, be sure to disable interrupts during the reading, altering, and writing of the bits.) The meaning of each bit is described below.

SetModeBits

Sets the mode bits.

```
CMDLIST   DFB   $03               ;Parameter count
          DFB   $01               ;Command code
          DW    $00               ;Result code (output)
          DL    ModeBitImage      ;(input)
```

Use this call to alter any of the mode bits whose function is described above. First read in the bits using GetModeBits, then alter the bits of interest, and then write the bits by using this call. (Be sure to disable interrupts, as discussed in the GetModeBits description.) The bits marked Preserve should not be changed; they are informational only. Altering these bits will confuse the firmware.

ModeBitImage is 4 bytes, where bit 0 is the least significant bit of the lowest addressed byte and bit 31 is the most significant bit of the highest addressed byte.

[31..24] (Preserve)
[23] 1 = Ignore commands in the output flow
[22] 1 = Framing error has occurred
[21] (Preserve)
[20] 1 = Parity error has occurred
[19..16] (Preserve)
[19..16] (Preserve)
[15] (Preserve)
[14] (Preserve) 1 = I/O buffering enabled
[13] 1 = DCD handshaking enabled
[12] (Preserve)
[11] 1 = Generate CR at end of line

[10]	(Preserve) 1 = Input flow halted
[9]	(Preserve) 1 = Output flow halted
[8]	(Preserve) 1 = Background printing in progress
[7]	1 = Echo input to the video screen
[6]	1 = Generate LF after CR
[5]	1 = XON/XOFF handshaking enabled
[4]	1 = Accept keyboard input
[3]	0 = Delete LF after CR
[2]	1 = DTR/DSR handshaking enabled
[1]	(Preserve) 1 = awaiting XON character
[0]	(Preserve) 1 = communications mode, 0 = printer mode

Buffer-management calls

GetInBuffer

Returns the address and length of the input buffer.

```
CMDLIST   DFB   $04        ;Parameter count
          DFB   $10        ;Command code
          DW    $00        ;Result code (output)
          DL    $00        ;Buffer address (output)
          DW    $00        ;Buffer length (output)
```

This call and the one that follows (GetOutBuffer) are used to determine the addresses and lengths of the current input and output buffers. If background printing is to be invoked and the application wants to use the default buffer, its address can be retrieved by these calls.

GetOutBuffer

Returns the address and length of the output buffer.

```
CMDLIST   DFB   $04        ;Parameter count
          DFB   $11        ;Command code
          DW    $00        ;Result code (output)
          DL    $00        ;Buffer address (output)
          DW    $00        ;Buffer length (output)
```

SetInBuffer

Specifies the buffer to contain the input queue.

```
CMDLIST   DFB   $04              ;Parameter count
          DFB   $12              ;Command code
          DW    $00              ;Result code (output)
          DL    Buffer address   ;(input)
          DW    Buffer length    ;(input)
```

This call and the one following (SetOutBuffer) allow the application to change the location and length of the input or output buffers. A queue buffer can cross bank boundaries but must be fixed in memory while buffering is active.

SetOutBuffer

Specifies the buffer to contain the output queue.

```
CMDLIST   DFB   $04              ;Parameter count
          DFB   $13              ;Command code
          DW    $00              ;Result code (output)
          DL    Buffer address   ;(input)
          DW    Buffer length    ;(input)
```

FlushInQueue

Discards all characters in the input queue.

```
CMDLIST   DFB   $02              ;Parameter count
          DFB   $14              ;Command code
          DW    $00              ;Result code (output)
```

This call and the one following (FlushOutQueue) allow the application to flush unwanted data from the input and output queues.

FlushOutQueue

Discards all the characters in the output queue.

```
CMDLIST   DFB   $02              ;Parameter count
          DFB   $15              ;Command code
          DW    $00              ;Result code (output)
```

InQStatus

Returns information about the input queue.

```
CMDLIST  DFB   $04        ;Parameter Count
         DFB   $16        ;Command Code
         DW    $00        ;Result Code (output)
         DW    $00        ;Number of characters in receive queue
                           (output)
         DW    $00        ;Time since last receive character queued
                           (output)
```

This call and the one following (OutQStatus) call return information about the input and output queues. The InQStatus call additionally returns the number of heartbeat ticks (1 tick = 1/30 second) between the time the last character was queued and the time of the call. Note that for this number to be valid, the application must have turned on the heartbeat system by making a tool call.

OutQStatus

Returns information about the output queue.

```
CMDLIST  DFB   $04        ;Parameter count
         DFB   $17        ;Command code
         DW    $00        ;Result code (output)
         DW    $00        ;Number of characters until transmit
                           queue overflow (output)
         DW    $00        ;Reserved (output)
```

SendQueue

Launches background printing.

```
CMDLIST  DFB   $04               ;Parameter count
         DFB   $18               ;Command code
         DW    $00               ;Result code (output)
         DW    Data length
         DL    Recharge address
```

This call begins the background-printing process. The application must first set the output buffer address (or use the default buffer) to load the data to be output into the buffer starting at the buffer base address. The data then is placed into the buffer. The call to SendQueue then must be made specifying the length of the data in the buffer and the 4-byte address of a subroutine (the Recharge routine), which will be called by the interrupt firmware when all characters have been sent. (See the section earlier in this chapter for further information about Recharge.)

Hardware control calls

Refer to the section "Compatibility" at the beginning of this chapter.

GetPortStat

Returns the port hardware status.

```
CMDLIST   DFB   $03                  ;Parameter count
          DFB   $06                  ;Command code
          DW    $00                  ;Result code (output)
          DW    $00                  ;Port status info (output)
```

This call is used to get the current status of the serial channel at the hardware level. There are 16 bits of result. The meaning of these bits is as follows:

[15..8]		(Reserved)
[7]	Break/Abort	Set to 1 when a break sequence is detected
[6]	Tx Underrun	Set to 1 when a transmit underrun occurs
[5]	DSR	State of the input handshake line
[4]		(Reserved)
[3]	DCD	State of the general-purpose input line
[2]	Tx Buff Empty	Set to 1 when ready to transmit next character
[1]		(Reserved)
[0]	Rx Char Avail	Set to 1 when a character is available to be read

GetSCC

Returns the value of the specified SCC register.

```
CMDLIST   DFB   $04                  ;Parameter count
          DFB   $08                  ;Command code
          DW    $00                  ;Result code (output)
          DFB   Register             ;SCC register number (input)
          DFB   $00                  ;Value of SCC register (output)
```

GetSCC returns the value in a specified SCC register. The GetSCC and SetSCC calls allow direct access to the serial hardware. (See the SCC 8530 technical manual for a description of the registers in the serial controller chip.) The serial-port firmware does not need to be initialized for these calls to work; in fact, these calls should be used only if the application is handling all serial tasks itself and not using the firmware at all.

SetSCC

Writes a value into the SCC.

```
CMDLIST  DFB  $04                 ;Parameter count
         DFB  $09                 ;Command code
         DW   $00                 ;Result code (output)
         DFB  Register            ;SCC register to write (input)
         DFB  Value               ;Value to write to Register (input)
```

This call allows the writing of a register in the SCC.

GetDTR

Returns the value of the output handshake line.

```
CMDLIST  DFB  $03                 ;Parameter count
         DFB  $0A                 ;Command code
         DW   $00                 ;Result code (output)
         DW   $00                 ;Bit 7 is the state of DTR (output)
```

Use this call to find out the current setting of the output handshake line. The state of this line is returned in the most significant bit of the returned byte. The line may be set by the SetDTR call.

SetDTR

Sets the value of the output handshake line.

```
CMDLIST  DFB  $03                 ;Parameter count
         DFB  $0B                 ;Command code
         DW   $00                 ;Result code (output)
         DW   DTR state           ;Bit 7 is the state of DTR (input)
```

Use this call to set the current mode of the output handshake line.

GetIntInfo

Returns the type of interrupt (for use in the interrupt completion routine).

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

This call allows the application to determine which type of interrupt caused the application's completion routine to be called. The meanings of the bits are the same as for SetIntInfo.

SetIntInfo

Sets up informational interrupt handling.

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

This call allows the application to specify the types of interrupts that will be passed to the application's interrupt routine. The firmware should be enabled and buffering turned on when this call is made. The types of interrupts and the bits used to enable them are as shown in Table 5-7.

The extended serial-port commands are summarized in Figures 5-4 and 5-5.

Table 5-7
Interrupt setting enable bits

[15..8]	(Reserved)	Set these to zero
[7]	Break/Abort	Break sequence detect
[6]	Tx Underrun	Transmit underrun detect
[5]	CTS	Transition on input handshake line
[4]	0	(Reserved)
[3]	DCD	Transition on general-purpose line
[2]	Tx	Transmit register empty
[1]	0	(Reserved)
[0]	Rx	Character available

GetInBuffer	
Parameter count = $04	1
Command code = $10	1
Result code	2
Buffer base address	4
Buffer length	2

Return location and length of the receive queue buffer

GetOutBuffer	
Parameter count = $04	1
Command code = $11	1
Result code	2
Buffer base address	4
Buffer length	2

Return location and length of the transmit queue buffer

SetInBuffer	
Parameter count = $04	1
Command code = $12	1
Result code	2
Buffer base address	4
Buffer length	2

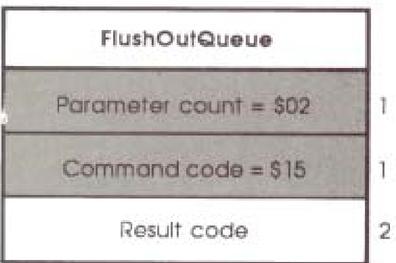
Set location and length of the receive queue buffer

SetOutBuffer	
Parameter count = $04	1
Command code = $13	1
Result code	2
Buffer base address	4
Buffer length	2

Set location and length of the transmit queue buffer

FlushInQueue	
Parameter count = $02	1
Command code = $14	1
Result code	2

Throw away all characters in the receive queue



Throw away all characters in the transmit queue

InQStatus	
Parameter count = $04	1
Command code = $16	1
Result code	2
Number of characters in receive queue	2
Number of ticks since last character arrived	2

Return receive queue information

OutQStatus	
Parameter count = $04	1
Command code = $17	1
Result code	2
Number of character spaces left in transmit queue	2
Reserved	2

Return transmit queue information

SendQueue	
Parameter count = $04	1
Command code = $18	1
Result code	2
Data length	2
Completion address	4

Begin background output

Figure 5-4
Summary of extended serial-port buffer commands

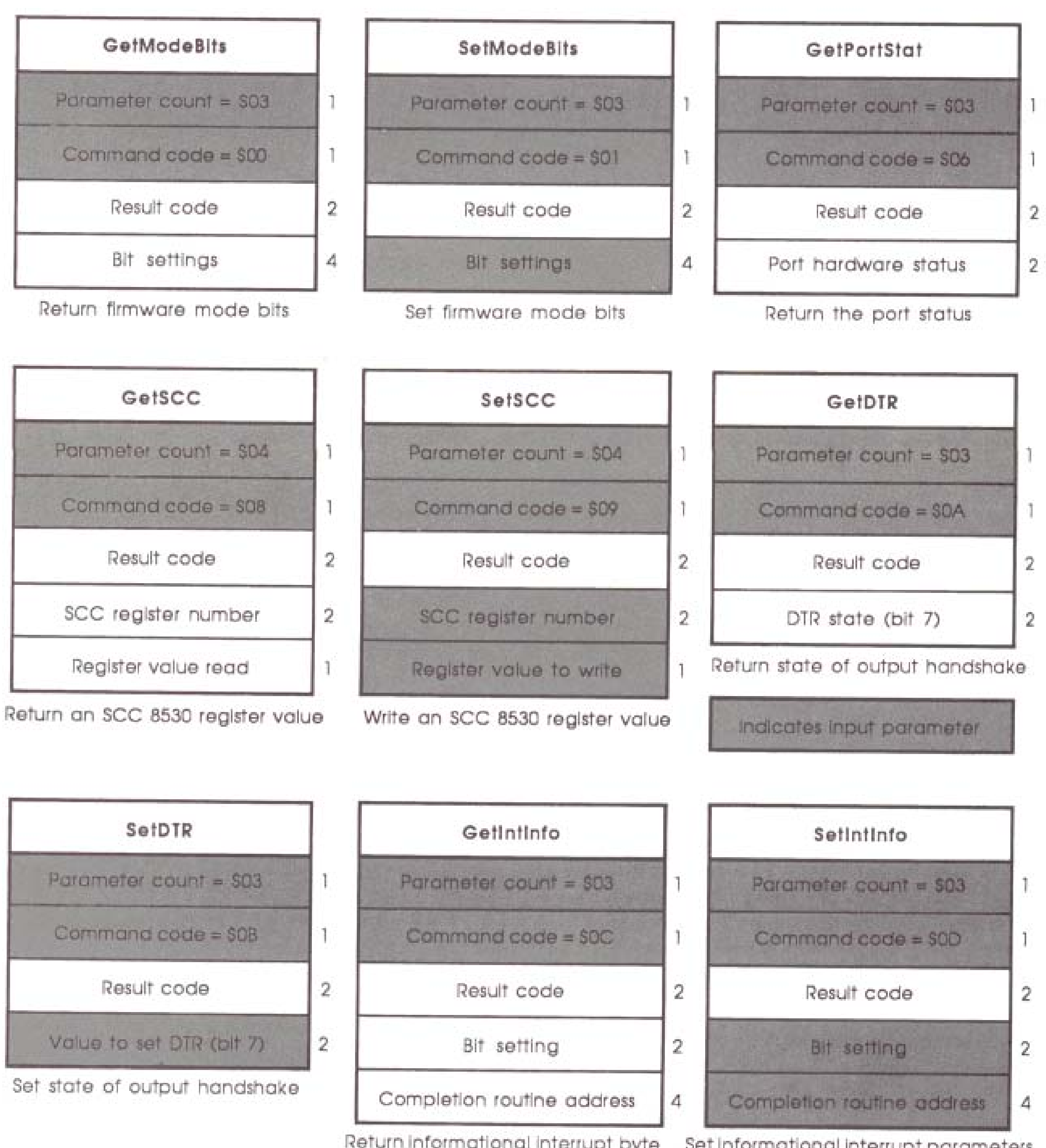


Figure 5-5
Summary of extended serial-port mode and hardware control commands

Chapter 6

Disk II Support

This chapter describes the Apple IIGS Disk II support. Several different types of disk drives can be attached to the Apple IIGS, some of which contain built-in intelligence. This chapter describes the methods by which the Disk II product can be connected to the Apple IIGS. The Apple IIGS disk-support system, with its built-in Integrated Woz Machine (IWM) chip, accommodates Disk II (DuoDisk and UniDisk) 5.25-inch drives, 3.5-inch drives with built-in intelligence (UniDisk 3.5), and Apple 3.5 drives.

The IWM divides the Apple IIGS disk port (on the back of the computer) into I/O ports 5 and 6. The ports are equivalent to internal versions of device drivers installed in expansion slots 5 and 6, respectively. The Control Panel setting for slot 5 or 6 determines whether the I/O port or a card physically present in that slot is active.

Port 6 provides the standard Disk II support. Disk II boot routines are built into ROM. Disk II routines in DOS, ProDOS, and Pascal operate the same as they do in Apple II computers prior to the Apple IIGS. Direct access to Disk II devices (reading and writing tracks and sectors, seeking to specified tracks, and so on) is provided by whichever operating system you boot. Separate firmware support is provided only for booting from Disk II devices.

Port 5 is called *SmartPort*. It consists of an expanded version of the SmartPort firmware used in the 32K Apple II ROM. SmartPort is capable of supporting a combination of character and block devices up to a total of 127 devices. It controls the UniDisk 3.5 and Apple 3.5 drives as well as the ROM disk and the RAM disk. The SmartPort firmware is discussed in detail in Chapter 7, "SmartPort Firmware."

You can attach up to two Disk II drives, two Apple 3.5 drives, and two or more UniDisk 3.5 drives to the Apple IIGS disk port, depending on IWM output specifications. A maximum of six devices can be connected at any one time. The disks must be attached in the order shown in Figure 6-1 (Apple 3.5 drives first, followed by UniDisk 3.5 drives, followed by Disk II drives).

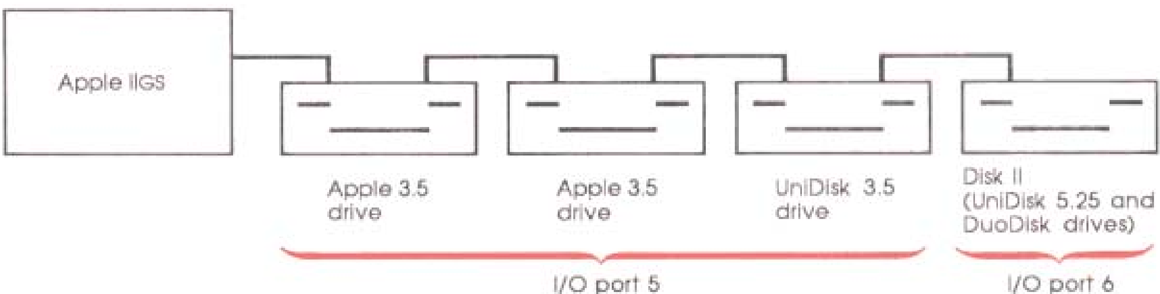


Figure 6-1
Order of disk drives on Apple IIGS disk ports

Interface routines for ports 5 and 6 access the IWM using slot 6 soft switches. The firmware arbitrates between slot use of the same soft switches. If a peripheral card is plugged into slot 6, the firmware in port 5 can still access the disks connected to port 6 by temporarily disabling the external peripheral card, performing disk access, and then reenabling the external peripheral card.

The port 6 disk interface firmware resides in the $C600 address space. It supports up to two drives, addressed as though they are connected to slot 6, as physical drives 1 and 2. Both drives use single-sided, 143K-capacity, 35-track, 16-sector format. Table 6-1 summarizes the Disk II I/O port characteristics.

Table 6-1
Disk II I/O port characteristics

Drive number	Port 6, drive 1 Port 6, drive 2
Commands	`IN#6` or `PR#6` from BASIC or `Call -151` (to get to Monitor from BASIC) and 6 Control-P
Initial characteristics	All resets with valid reset vector, except Control-Reset, pass control to slot 6 drive 1 if this drive is set (through Control Panel) as boot device or if scan is selected and no boot volume is found in higher-priority slot
Hardware location	Internal, $C0E0–$C0EF, reserved for Disk II and SmartPort use
Monitor firmware routines	None
I/O firmware entry points	$C600 (port 6 boot address) $C65E (read first track, first sector and begin execution of code found there)
Use of screen holes	Port 6 main- and auxiliary-memory screen holes reserved

Startup

The Apple IIGS can be started by using either a cold start or a warm start. A cold start clears the machine's memory and tries to load an operating system from disk. A warm start stops the program currently running and leaves the machine in Applesoft BASIC with memory and programs intact.

A cold start can be initiated by any of the following:

- turning the machine on
- pressing ⌘-Control-Reset
- issuing a reboot command from the Monitor, from BASIC, or from a program
- pressing Control-Reset if a valid reset vector does not exist

If you have set the startup device (from the Control Panel) to slot 6 or if you have selected scan and no boot volume is found in a higher-priority slot, the cold-start routine first sets a number of soft switches (see Appendix E, "Soft Switches") and then passes control to the program entry point at $C600. This code turns on the Disk II unit 1 device motor and then recalibrates the head to track 0 and reads sector 0 from that track. The sector contents are loaded into memory starting at address $0800; then program control passes to $0801. The program loaded depends on the operating system or application program on the disk.

To restart the system from BASIC, issue a PR#6 command; from the Monitor command mode, issue 6 Control-P; and from a machine-language program, use JMP $C600.

A warm start begins when you press Control-Reset if a valid reset vector exists. Normally, a warm start leaves you in BASIC with memory unchanged. If a program has changed the reset vector, the system will not perform a warm start. Usually, a program either performs a cold start or beeps and does nothing, leaving you in the currently executing program.

Chapter 7

SmartPort Firmware

The SmartPort firmware is associated with I/O port 5 (internal slot 5). It consists of assembly-language routines that support a series of block or character devices connected to the Apple IIGS external disk port. The SmartPort firmware converts calls to an appropriate format for transmittal over the disk port to control intelligent devices, that is, devices that can interpret command streams, such as the UniDisk 3.5 drive. The SmartPort also provides an interface to several unintelligent devices, that is, devices that require specific hardware control and employ no built-in intelligence, through the use of device-specific drivers that are accessed through the SmartPort extended interface calls. Unintelligent devices supported on the Apple IIGS through the SmartPort include the Apple 3.5 drive, RAM disk, and ROM disk.

To use the SmartPort interface, a program issues calls similar to ProDOS 8 machine-language interface calls. Each call consists of a JSR to the SmartPort entry point, followed by a SmartPort command byte, followed by a pointer to a table containing the parameters necessary for the call. The calls to SmartPort take two possible forms. The standard version of a call allows your program to move data to and from bank $00 of the memory. You use the extended version of the call to move data to and from other banks of memory.

Locating SmartPort

You can determine whether the SmartPort interface is installed in a system by examining the ProDOS block-device signature bytes shown here:

```
$Cn01 = $20
$Cn03 = $00
$Cn05 = $03
```

You must also verify the existence of the SmartPort signature byte:

```
$Cn07 = $00
```

In the preceding addresses, n is the slot number for which the signature bytes are being examined. All peripheral cards or ports with these signature-byte values support both ProDOS block-device calls and SmartPort calls. You can examine the SmartPort ID type byte to obtain more information about any special support that may be built into the SmartPort driver. The SmartPort ID type byte located at $CnFB has been encoded to indicate the types of devices that can be supported by the SmartPort driver. This byte pertains to the interface only. For example, the Apple IIGS SmartPort interface in internal slot 5 may support a RAM disk, but it is not a RAM card, so bit 0 is cleared.

Figure 7-1 illustrates the contents of this ID type byte. Note that a driver that supports extended SmartPort calls must also support standard SmartPort calls. Bit 1, SCSI, indicates support for the Small Computer System Interface (SCSI).

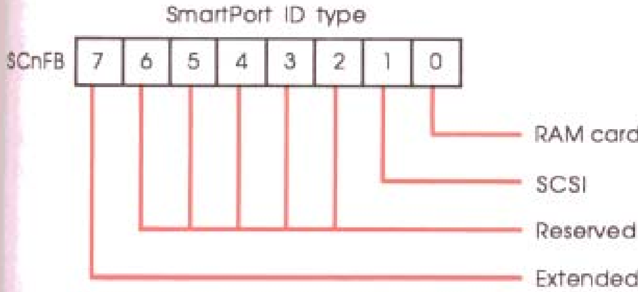


Figure 7-1
SmartPort ID type byte

Locating the dispatch address

Once you have determined that a SmartPort interface exists in a slot or port, you need to determine the entry point, or *dispatch address,* for the SmartPort. This address is determined by the value found at $CnFF, where n is the slot number. By adding the value at $CnFF to the address $Cn00, you can calculate the standard ProDOS block-device driver entry point. More information about this entry point is available in the *ProDOS Technical Reference*. The SmartPort entry point is located 3 bytes after the ProDOS entry point. Therefore, the SmartPort entry point is $Cn00 plus 3 plus the value found at $CnFF.

For example, if the signature bytes for the SmartPort interface are in slot 5 and $C5FF contains a hexadecimal value of $0A, the ProDOS entry point is $C50A, and the SmartPort entry point is $C50A plus 3, or $C50D.

SmartPort call parameters

SmartPort calls include several parameters. Not all parameters appear in every SmartPort call. The parameter types that may be required when making a SmartPort call are as follows:

Parameter	Description
Command name	Name used to identify the SmartPort call
Command number	Byte value that you position contiguous in memory with the JSR to the SmartPort entry point; hexadecimal number that specifies the type of SmartPort call (bit 6 is cleared to 0 for standard calls and set to 1 for extended calls)
Parameter list pointer	Pointer that you position contiguous in memory with the command number that points to the parameter list
Parameter count	The first item in the parameter list; hexadecimal byte value that specifies the number of parameters in the parameter list
Unit number	Hexadecimal byte value that specifies the unit number of the device to or from which the SmartPort call is to direct I/O
Buffer address	Pointer to memory that will be used in the I/O transaction (for standard SmartPort calls, this is a word-wide pointer referencing memory in bank zero; for extended calls, the pointer is a longword referencing memory in any bank)
Block number	Number specifying the block address used in an I/O transaction with a block device (for standard SmartPort calls, this parameter is 24 bits wide; for extended calls, this parameter is 32 bits wide)
Byte count	Specifies the number of bytes to be transferred between memory and the device (this parameter is 16 bits wide)
Address pointer	Specifies an address within the device

SmartPort assignment of unit numbers

The unit number is included in every parameter list. The unit number specifies which device connected to the slot 5 hardware responds to the commands you issue. Calls that allow you to reference the SmartPort itself use a unit number of zero. Only Status, Init, and Control calls may be made to unit zero. The Apple IIGS assigns unit numbers to devices in ascending order starting with unit number $01. Devices are assigned unit numbers starting with the RAM disk, ROM disk, and Apple 3.5 drive, and finally proceeding to intelligent devices such as the UniDisk 3.5.

Allocation of device unit numbers

The Apple IIGS implementation of the SmartPort interacts with the Control Panel selection of boot devices. For any given port, a boot can occur only from the first device logically connected to that port. Booting from Disk II devices is handled by the slot 6 firmware. SmartPort support is provided to allow booting from any of three types of devices:

- RAM disk
- ROM disk
- Disk drive (Apple 3.5 drive or UniDisk 3.5)

Depending on the devices that are connected to the slot 5 hardware, the selected boot device may not be the first logical device in the chain. To boot from the selected device, using the Control Panel settings, the SmartPort firmware logically moves the selected device to the first unit in the device chain. All devices that were previously ahead of the selected boot device must then be moved logically so that they are now located behind the selected boot device.

The initialization call handles assignments of unit numbers in a two-stage process. In the first stage, unit numbers are assigned as described above, in the section "SmartPort Assignment of Unit Numbers." In the second stage, the units are remapped so that the selected boot device is always the first logical device in the chain. If Scan is selected as the boot option in the Control Panel, the SmartPort places the first physical disk drive as the first logical device in the device chain.

Device remapping is necessary for certain device configurations under ProDOS. Current implementations of ProDOS (both ProDOS 8 and ProDOS 16) support only two devices per port or slot. If more than two devices are connected to the device chain, devices beyond the second cannot be accessed. ProDOS 8 and ProDOS 16 get around this restriction by logically mapping devices beyond the second device so that they appear to be connected to slot 2. Using this method, ProDOS 8 and ProDOS 16 can support up to four devices on the chain.

❖ *Note:* Future versions of ProDOS 16 will support more than two devices per port or slot so that no remapping of units to slot 2 will be necessary.

Figures 7-2 through 7-6 show device remapping derived from the selected boot device versus the device configuration. Only a few of the possible remapping variations are shown.

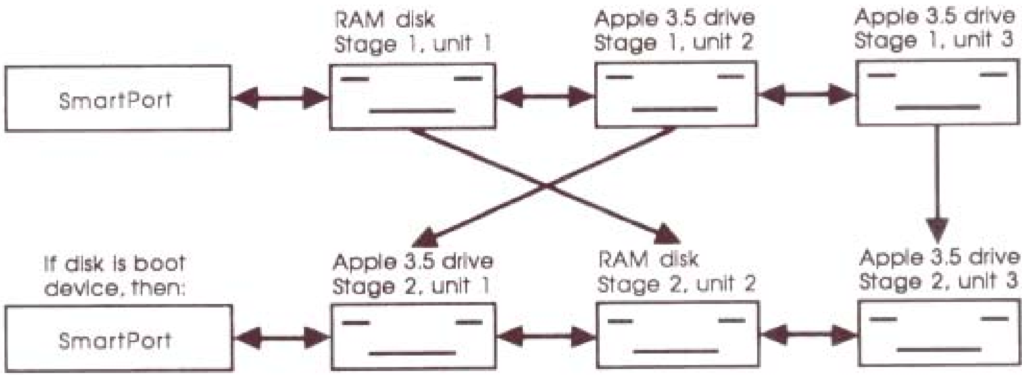


Figure 7-2
Device mapping: configuration 1, derivation 1

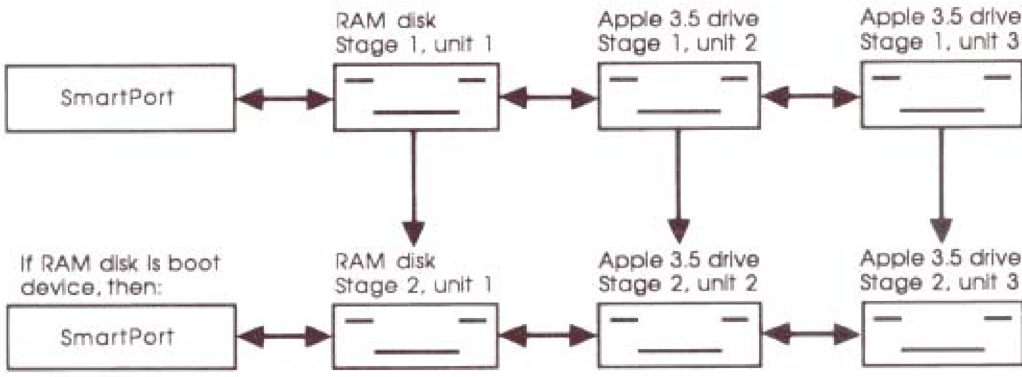


Figure 7-3
Device mapping: configuration 1, derivation 2

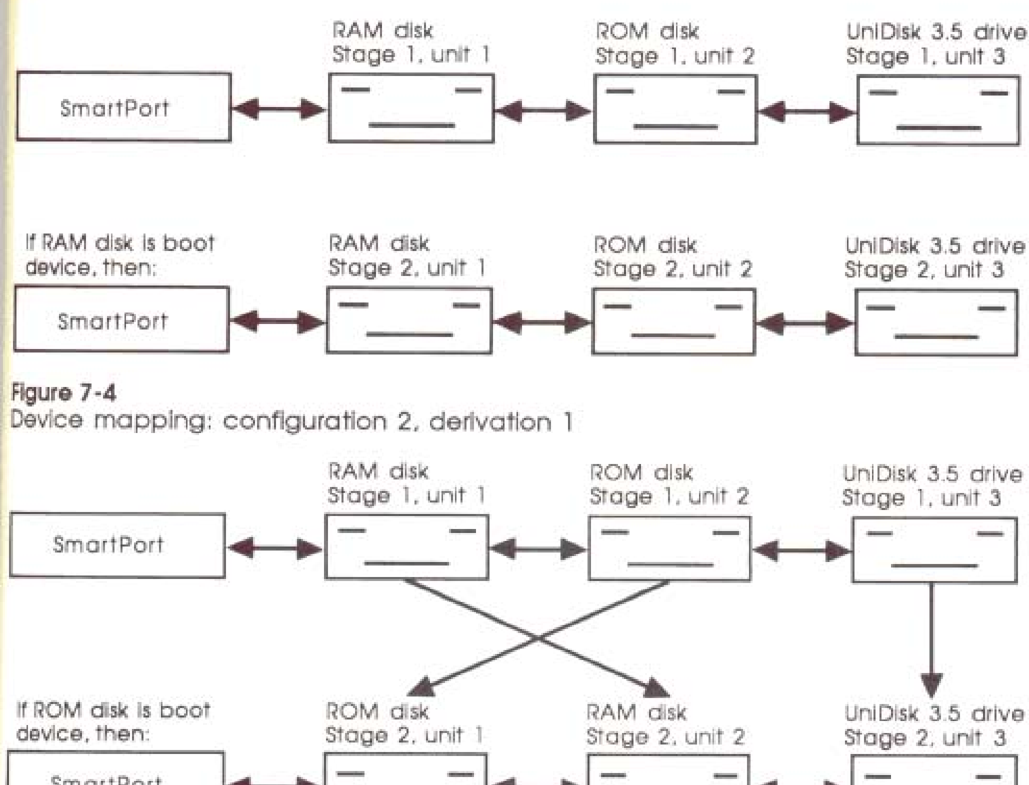


Figure 7-4
Device mapping: configuration 2, derivation 1

Figure 7-5
Device mapping: configuration 2, derivation 2

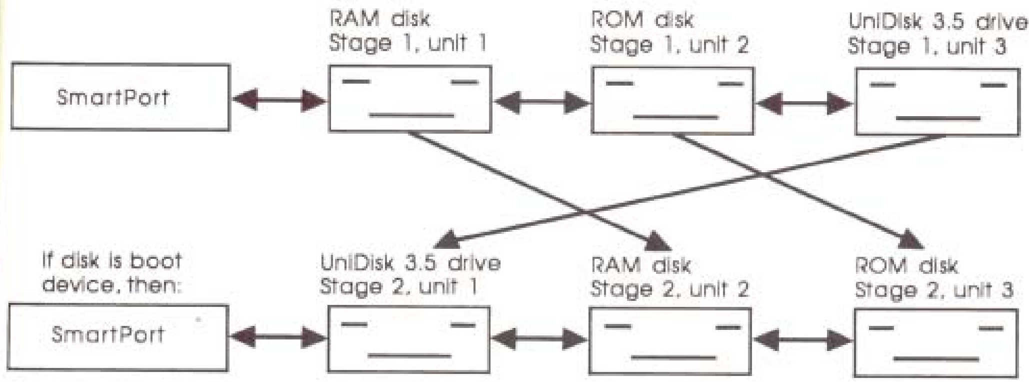


Figure 7-6
Device mapping: configuration 2, derivation 3

Issuing a call to SmartPort

SmartPort calls fall into two categories: standard calls and extended calls. Standard SmartPort calls are designed for interfacing Apple II peripherals. Extended SmartPort calls are designed for peripheral devices that can take advantage of the 65816 processor's ability to transfer data between any memory bank and the peripheral device and may require larger block addressing than is possible with the standard SmartPort calls.

For standard SmartPort calls, the pointer following the SmartPort command byte is a word-wide pointer to a parameter list in bank zero. For extended SmartPort calls, the pointer is a longword pointer to a parameter list in any memory bank.

There are several constraints on the use of the SmartPort:

- The stack use is 30–35 bytes. Programs should allow 35 bytes of stack space for each call.
- The SmartPort cannot generally be used to put anything into absolute zero page locations. Absolute zero page is defined as the direct page when the direct register is set to $0000.
- The SmartPort can be called only from Apple II emulation mode. This means that the emulation flag in the 65C816 processor status byte must be set to 1, and the direct-page register and data bank register must both be set to zero. Native-mode programs wishing to call the SmartPort must switch to emulation mode prior to making a SmartPort call. Such programs may cause corruption of the contents of the stack pointer. Refer to Chapter 2, "Notes for Programmers," for more information about switching processor modes.

This is an example of a standard SmartPort call:

```
SP_CALL      JSR    DISPATCH       ;Call SmartPort command dispatcher
             DFB    CMDNUM         ;This specifies the command type
             DW     CMDLIST        ;Word pointer to the parameter list in bank $00
             BCS    ERROR          ;Carry is set on an error
```

This is an example of a extended SmartPort call:

```
SP_EXT_CALL  JSR    DISPATCH       ;Call SmartPort command dispatcher
             DFB    CMDNUM+$40     ;This specifies the extended command type
             DW     CMDLIST        ;Low-word pointer to the parameter list
             DW     ^ CMDLIST      ;High-word pointer to the parameter list
             BCS    ERROR          ;Carry is set on an error
```

On completion of a call, execution returns to the RTS address plus 3 for a standard call and to the RTS address plus 5 for an extended call (the BCS statement in the examples). If the call was successful, the C flag is cleared and the A register is set to 0; if it was unsuccessful, the C flag is set and the A register contains the error code. If data is transferred from the device to the CPU, the X register contains the low byte count and the Y register contains the high byte count.

The complete register status upon completion is summarized in Table 7-1.

Table 7-1
Register status on return from SmartPort

	65816 status byte												
	N	V	I	B	D	I	Z	C	Acc	X	Y	PC	SP
Successful standard call	X	X	1	X	0	U	X	0	0	n	n	JSR+3	U
Successful extended call	X	X	1	X	0	U	X	0	0	n	n	JSR+5	U
Unsuccessful standard call	X	X	1	X	0	U	X	1	Error	X	X	JSR+3	U
Unsuccessful extended call	X	X	1	X	0	U	X	1	Error	X	X	JSR+5	U

* *Note:* X = undefined, U = unchanged, n = undefined for transfers to the device or number of bytes transferred when the transfer was from the device to the host.

Generic SmartPort calls

Generic SmartPort calls are explained in detail in the following sections.

Status

The Status call returns status information about a particular device or about the SmartPort itself. Only Status calls that return general information are listed here. Device-specific Status calls can also be implemented by a device for diagnostic or other information. Device-specific calls must be implemented with a status code of $04 or greater. On return from a Status call, the X and Y registers contain a count of the number of bytes transferred to the host. X contains the low byte of the count, and Y contains the high byte of the count.

	Standard call	Extended call
CMDNUM	$00	$40
CMDLIST	Parameter count	Parameter count
	Unit number	Unit number
	Status list pointer (low byte)	Status list pointer (low byte, low word)
	Status list pointer (high byte)	Status list pointer (high byte, low word)
	Status code	Status list pointer (low byte, high word)
		Status list pointer (high byte, high word)
		Status code

Required parameters

Parameter count Byte value = $03

Unit number 1-byte value in the range $00, $01 to $7E

Each device has a unique number assigned to it at initialization time. The numbers are assigned according to the device's position in the chain. A Status call with a unit number of $00 specifies a call for the overall SmartPort status.

	Standard call	Extended call
Status list pointer	Word pointer (bank $00)	Longword pointer

This is a pointer to the buffer to which the status list is to be returned. For standard calls, this is a word-wide pointer defaulting to bank $00. For extended calls, this is a longword pointer. Note that the length of the buffer varies, depending on the status request being made.

Status code 1-byte value in the range $00 to $FF

This is the number of the status request being made. All devices respond to the following requests:

Status code	Status returned
$00	Return device status
$01	Return device control block
$02	Return newline status (character devices only)
$03	Return device information block (DIB)

Although devices must respond to the preceding status requests, a device may not be able to support the request. In this case, the device returns an invalid status code error ($21).

Statcode = $00: The device status consists of 4 bytes. The first is the general status byte:

Bit	Function
7	1 = Block device; 0 = Character device
6	1 = Write allowed
5	1 = Read allowed
4	1 = Device online or disk in drive
3	1 = Format allowed
2	1 = Media write protected (block devices only)
1	1 = Device currently interrupting (supported by Apple IIc only)
0	1 = Device currently open (character devices only)

If the device is a block device, the next field indicates the number of blocks in the device. This is a 3-byte field for standard calls or a 4-byte field for extended calls. The least significant byte is first. If the device is a character device, these bytes are set to zero.

Statcode = $01: The device control block (DCB) is device dependent. The DCB is typically used to control various operating characteristics of a device. The DCB is set with the corresponding Control call. The first byte is the number of bytes in the control block. A value of $00 returned in this byte indicates a DCB length of 256, and a value of $01 indicates a DCB length of 1 byte. The length of the DCB is always in the range of 1 to 256 bytes, excluding the count byte.

Statcode = $02: No character devices are currently implemented for use on the SmartPort, so the newline status is presently undefined.

Statcode = $03: This call returns the device information block (DIB). It contains information identifying the device and its type and various other attributes. The returned status list has the following form:

STATLIST	**Standard call**	**Extended call**
	Device status byte	Device status byte
	Block size (low byte)	Block size (low byte, low word)
	Block size (mid byte)	Block size (high byte, low word)
	Block size (high byte)	Block size (low byte, high word)
	ID string length	Block size (high byte, high word)
	ID string (16 bytes)	ID string length
	Device type byte	ID string (16 bytes)
	Device subtype byte	Device type byte
	Version word	Device subtype byte
		Version word

The device status is a 1-byte field that is the same as the general status byte returned in the device Status call (statcode = $00). The block size field is the same as the block size field returned in the device Status call. The ID string consists of 1-byte prefix indicating the number of ASCII characters in the ID string. This is followed by a 16-byte field containing an ASCII string identifying the device. The most significant bit of each ASCII character is set to zero.

If the ASCII string consists of fewer than 16 characters, ASCII spaces are used to fill the unused portion of the string buffer. The device type and device subtype fields are 1-byte fields. Several bits encoded within the DIB subtype byte are defined to indicate whether a device supports the extended SmartPort interface, disk-switched errors, or removable media.

A breakdown of the DIB subtype byte is shown in Figure 7-7.

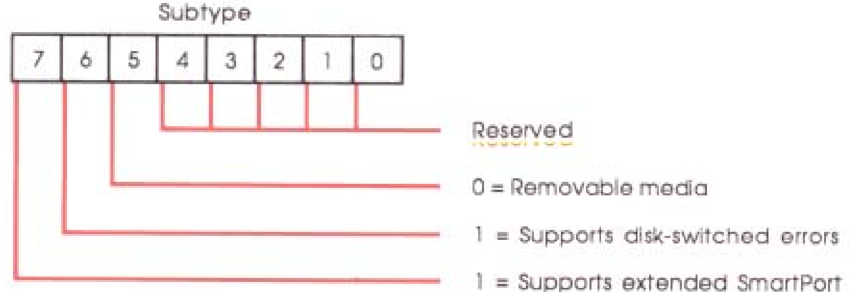


Figure 7-7
SmartPort device subtype byte

Applications requiring specific knowledge about a device should execute a DIB status and examine the type byte. The subtype byte is used to obtain information about special features a device may support. Several device types and subtypes are assigned to existing SmartPort devices. These types and subtypes are as follows:

Type	Subtype	Device
$00	$00	Apple II memory expansion card
$00	$C0	Apple IIGS Memory Expansion Card configured as a RAM disk
$01	$00	UniDisk 3.5
$01	$C0	Apple 3.5 drive
$03	$E0	Apple II SCSI with nonremovable media

Undefined SmartPort devices may implement the following types and subtypes:

Type	Subtype	Device
$02	$20	Hard disk
$02	$00	Removable hard disk
$02	$40	Removable hard disk supporting disk-switched errors
$02	$A0	Hard disk supporting extended calls
$02	$C0	Removable hard disk supporting extended calls and disk-switched errors
$02	$A0	Hard disk supporting extended calls
$03	$C0	SCSI with removable media

The firmware version field is a 2-byte field consisting of a number indicating the firmware version.

SmartPort driver status

A Status call with a unit number of $00 and a status code of $00 is a request to return the status of the SmartPort driver. This function returns the number of devices as well as the current interrupt status. The format of the status list returned is as follows:

STATLIST	Byte 0	Number of devices
	Byte 1	Reserved
	Byte 2	Reserved
	Byte 3	Reserved
	Byte 4	Reserved
	Byte 5	Reserved
	Byte 6	Reserved
	Byte 7	Reserved

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

Possible errors

The following error return values are possible.

$06	BUSERR	Communications error
$21	BADCTL	Invalid status code
$30–$3F	$50–$7F	Device-specific error

ReadBlock

This call reads one 512-byte block from the block device specified by the unit number passed in the parameter list. The block is read into memory starting at the address specified by the data buffer pointer passed in the parameter list.

	Standard call	Extended call
CMDNUM	$01	$41
CMDLIST	Parameter count	Parameter count
	Unit number	Unit number
	Data buffer pointer (low byte)	Data buffer pointer (low byte, low word)
	Data buffer pointer (high byte)	Data buffer pointer (high byte, low word)
	Block number (low byte)	Data buffer pointer (low byte, high word)
	Block number (middle byte)	Data buffer pointer (high byte, high word)
	Block number (high byte)	Block number (low byte, low word)
		Block number (high byte, low word)
		Block number (low byte, high word)
		Block number (high byte, high word)

Required parameters

Parameter count Byte value = $03

Unit number 1-byte value in the range $01 to $7E

	Standard call	Extended call
Data buffer pointer	Word pointer (bank $00)	Longword pointer

The data buffer pointer points to a buffer into which the data is to be read. For standard calls, this is a word pointer into bank $00. For extended calls, the pointer is a longword specifying a buffer in any memory bank. The buffer must be 512 bytes long.

	Standard call	Extended call
Block number	3-byte number	4-byte number

The block number is the logical address of a block of data to be read. There is no general connection between block numbers and the layout of tracks and sectors on the disk. Translation from logical to physical blocks is performed by the device.

Possible errors

The following error return values are possible.

$06	BUSERR	Communications error
$27	IOERROR	I/O error
$28	NODRIVE	No device connected
$2D	BADBLOCK	Invalid block number
$2F	OFFLINE	Device off line or no disk in drive

WriteBlock

The Write call writes one 512-byte block to the block device specified by the unit number passed in the parameter list. The block is written from memory starting at the address specified by the data buffer pointer passed in the parameter list.

	Standard call	Extended call
CMDNUM	$02	$42
CMDLIST	Parameter count	Parameter count
	Unit number	Unit number
	Data buffer pointer (low byte)	Data buffer pointer (low byte, low word)
	Data buffer pointer (high byte)	Data buffer pointer (high byte, low word)
	Block number (low byte)	Data buffer pointer (low byte, high word)
	Block number (middle byte)	Data buffer pointer (high byte, high word)
	Block number (high byte)	Block number (low byte, low word)
		Block number (high byte, low word)
		Block number (low byte, high word)
		Block number (high byte, high word)

Required parameters

Parameter count Byte value = $03

Unit number 1-byte value in the range $01 to $7E

	Standard call	Extended call
Data buffer pointer	Word pointer (bank $00)	Longword pointer

The data buffer pointer points to a buffer that the data is to be written from. For standard calls, this is a word pointer into bank $00. For extended calls, the pointer is a longword specifying a buffer in any memory bank. The buffer must be 512 bytes long.

	Standard call	Extended call
Block number	3-byte number	4-byte number

The block number is the logical address of a block of data to be written. There is no general connection between block numbers and the layout of tracks and sectors on the disk. The translation from logical to physical block is performed by the device.

Possible errors

The following error return values are possible.

$06	BUSERR	Communications error
$27	IOERROR	I/O error
$28	NODRIVE	No device connected
$2B	NOWRITE	Disk write protected
$2D	BADBLOCK	Invalid block number
$2F	OFFLINE	Device off line or no disk in drive

Format

The Format call formats a block device. Note that the formatting performed by this call is not linked to any operating system; it simply prepares all blocks on the medium for reading and writing. Operating-system-specific catalog information, such as bit maps and catalogs, are not prepared by this call.

	Standard call	Extended call
CMDNUM	$03	$43
CMDLIST	Parameter count	Parameter count
	Unit number	Unit number

Format call implementation

Some block devices may require device-specific information at format time. This device-specific information may include a spare list of bad blocks to be written following physical formatting of the media. In this case, it may not be desirable to implement the Format call so that a physical format is actually performed because a spare list of bad blocks may not be available from the vendor or because of the time involved in executing a bad-block scan. It may be more desirable to implement device-specific Control calls to lay down the physical tracks and initialize the spare lists. If this latter procedure is followed, the Format call need only return to the application with the accumulator set to $00 and the carry flag cleared. This procedure should be used only when it is not desirable for the application to physically format the media.

Required parameters

Parameter count	Byte value = $01
Unit number	Byte value in the range $01 to $7E

Possible errors

The following error return values are possible.

$06	BUSERR	Communications error
$27	IOERROR	I/O error
$28	NODRIVE	No device connected
$2B	NOWRITE	Disk write protected
$2F	OFFLINE	Device off line or no disk in drive

Control

The Control call sends control information to the device. The information may be either general or device specific.

	Standard call	Extended call
CMDNUM	$04	$44
CMDLIST	Parameter count	Parameter count
	Unit number	Unit number
	Control list pointer (low byte)	Control list pointer (low byte, low word)
	Control list pointer (high byte)	Control list pointer (high byte, low word)
	Control code	Control list pointer (low byte, high word)
		Control list pointer (high byte, high word)
		Control code

Required parameters

Parameter count Byte value = $03

Unit number Byte value in the range $00 to $7E

	Standard call	Extended call
Control list pointer	Word pointer (bank $00)	Longword pointer

The control list is a pointer to the user's buffer from which the control information is to be read. For the standard Control call, the pointer is a word value into bank $00. For the extended Control call, the pointer is a longword value that may reference any memory bank. The first two bytes of the control list specify the length of the control list, with the low byte first. A control list is mandatory, even if the call being issued does not pass information in the list. In this latter case, length of zero is used for the first two bytes.

Control code Byte value
Byte value in the range $00 to $FF

The control code is the number of the control request being made. This number and the function indicated are device specific, except that all devices must reserve the following codes for specific functions:

Code	Control function
$00	Resets the device
$01	Sets device control block
$02	Sets newline status (character devices only)
$03	Services device interrupt

Code = $00: This call performs a soft reset of the device. It generally returns housekeeping values to some reset value.

Code = $01: This Control call sets the device control block. Devices generally use the bytes in this block to control global aspects of the device's operating environment. Because the length is device dependent, the recommended way to set the DCB is to read in the DCB (with the Status call), alter the bits of interest, and then write the same string with this call. The first byte is the length of the DCB, excluding the byte itself. A value of $00 in the length byte corresponds to a DCB size of 256 bytes, and a count value of $01 corresponds to a DCB size of 1 byte. A count value of $FF corresponds to a DCB size of 255 bytes.

Possible errors

The following error return values are possible.

$06	BUSERR	Communications error
$21	BADCTL	Invalid control code
$22	BADCTLPARM	Invalid parameter list
$30–$3F	UNDEFINED	Device-specific error

Init

The Init call provides the application with a way of resetting the SmartPort.

	Standard call	Extended call
CMDNUM	$05	$45
CMDLIST	Parameter count	Parameter count
	Unit number	Unit number

Required parameters

Parameter count Byte value = $01

Unit number Byte value = $00

The SmartPort will perform initialization, hard resetting all devices and sending each their device numbers. This call may not be made to a specific unit; rather, it must be made to the SmartPort as a whole. This call may not be executed by an application. Issuing this call in conjunction with Control Panel changes may relocate devices contrary to the ProDOS device list. Applications wishing to reset a specific device should use the Control call with a control code of $00.

Possible errors

The following error return values are possible.

$06	BUSERR	Communications error
$28	NODRIVE	No device connected

Open

The Open call prepares a character device for reading or writing.

Note that a block device will not accept this call, but will return an invalid command error ($01).

	Standard call	Extended call
CMDNUM	$05	$45
CMDLIST	Parameter count	Parameter count
	Unit number	Unit number

Required parameters

Parameter count	Byte value = $01
Unit number	Byte value in the range $01 to $7E

Possible errors

The following error return values are possible.

$01	BADCMD	Invalid command
$06	BUSERR	Communications error
$28	NODRIVE	No device connected

Close

The Close call tells an extended character device that a sequence of read or write operations has ended. For a printer, this call could have the effect of flushing the print buffer.

Note that a block device will not accept this call, but will return an invalid command error ($01).

	Standard call	Extended call
CMDNUM	$07	$47
CMDLIST	Parameter count	Parameter count
	Unit number	Unit number

Required parameters

Parameter count	Byte value = $01
Unit number	Byte value in the range $01 to $7E

Possible errors

The following error return values are possible.

$01	BADCMD	Invalid command
$06	BUSERR	Communications error
$28	NODRIVE	No device connected

Read

The Read call reads the number of bytes specified by the byte count into memory. The starting address of memory that the data is read into is specified by the data buffer pointer. The address pointer references an address within the device that the bytes are to be read from. The meaning of the address parameter depends on the device involved. Although this call is generally intended for use by character devices, a block device might use this call to read a block of nonstandard size (a block larger than 512 bytes). In this latter case, the address pointer is interpreted as a block address.

	Standard call	Extended call
CMDNUM	$08	$48
CMDLIST	Parameter count	Parameter count
	Unit number	Unit number
	Data buffer pointer (low byte)	Data buffer pointer (low byte, low word)
	Data buffer pointer (high byte)	Data buffer pointer (high byte, low word)
	Byte count (low byte)	Data buffer pointer (low byte, high word)
	Byte count (high byte)	Data buffer pointer (high byte, high word)
	Address pointer (low byte)	Byte count (low byte)
	Address pointer (mid byte)	Byte count (high byte)
	Address pointer (high byte)	Address pointer (low byte, low word)
		Address pointer (high byte, low word)
		Address pointer (low byte, high word)
		Address pointer (high byte, high word)

Required parameters

Parameter count	Byte value = $04	
Unit number	1-byte value in the range $01 to $7E	
	Standard call	**Extended call**
Data buffer pointer	Word pointer (bank $00)	Longword pointer

For standard calls, this is the 2-byte pointer to a buffer into which the data is to be read. For extended calls, the pointer is a longword specifying a buffer in any memory bank. The buffer must be large enough to accommodate the number of bytes requested.

Byte count 2-byte number

The byte count specifies the number of bytes to be transferred. All of the current implementations of the SmartPort utilizing the SmartPort Bus have a limitation of 767 bytes. Other peripheral cards supporting the SmartPort interface may not have this limitation.

	Standard call	**Extended call**
Address pointer	3-byte address	4-byte address

The address is a device-specific parameter usually specifying a source address within the device. This call might be implemented with an extended block device using the address as a block address for accessing a nonstandard block. For example, such an implementation allows the Apple 3.5 drive and UniDisk 3.5 drive to read 524-byte Macintosh blocks from 3.5-inch media.

Possible errors

The following error return values are possible.

$06	BUSERR	Communications error
$27	IOERROR	I/O error
$28	NODRIVE	No device connected
$2B	NOWRITE	Disk write protected
$2D	BADBLOCK	Invalid block number
$2F	OFFLINE	Device off line or no disk in drive

Write

The Write call writes the number of bytes specified by the byte count to the device specified by the unit number. The starting memory address that the data is read from is specified by the data buffer pointer. The address pointer references an address within the device where the bytes are to be written. The meaning of the address parameter depends on the device involved. Although this call is generally intended for use by character devices, a block device might use this call to write a block of a nonstandard size (a block larger than 512 bytes). In this latter case, the address field is interpreted as a block address.

	Standard call	Extended call
CMDNUM	$09	$49
CMDLIST	Parameter count	Parameter count
	Unit number	Unit number
	Data buffer pointer (low byte)	Data buffer pointer (low byte, low word)
	Data buffer pointer (high byte)	Data buffer pointer (high byte, low word)
	Byte count (low byte)	Data buffer pointer (low byte, high word)
	Byte count (high byte)	Data buffer pointer (high byte, high word)
	Address pointer (low byte)	Byte count (low byte)
	Address pointer (mid byte)	Byte count (high byte)
	Address pointer (high byte)	Address pointer (low byte, low word)
		Address pointer (high byte, low word)
		Address pointer (low byte, high word)
		Address pointer (high byte, high word)

Required parameters

Parameter count Byte value = $04

Unit number 1-byte value in the range $01 to $7E

	Standard call	Extended call
Data buffer pointer	Word pointer (bank $00)	Longword pointer

For standard calls, this is the 2-byte pointer to a buffer into which the data is to be read. For extended calls, the pointer is a longword specifying a buffer in any memory bank. The buffer must be large enough to accommodate the number of bytes requested.

Byte count 2-byte number

The byte count specifies the number of bytes to be transferred. All of the current implementations of the SmartPort utilizing the SmartPort Bus have a limitation of 767 bytes. Other peripheral cards supporting the SmartPort interface may not have this limitation.

	Standard call	Extended call
Address pointer	3-byte value	4-byte value

The address is a device-specific parameter usually specifying a destination address within the device. This call might be implemented with a block device, using the address as a block address for accessing a nonstandard block. For example, such an implementation allows the Apple 3.5 drive and UniDisk 3.5 drive to write 524-byte Macintosh blocks to 3.5-inch media.

Possible errors

The following error return values are possible.

$06	BUSERR	Communications error
$27	IOERROR	I/O error
$28	NODRIVE	No device connected
$2B	NOWRITE	Disk write protected
$2D	BADBLOCK	Invalid block number
$2F	OFFLINE	Device off line or no disk in drive

Tables 7-2 and 7-3 summarize the command numbers and parameter lists for standard and extended SmartPort calls.

Table 7-2
Summary of standard commands and parameter lists

Command	Status	ReadBlock	WriteBlock	Format	Control	Init	Open	Close	Read	Write
CMDNUM	$00	$01	$02	$03	$04	$05	$06	$07	$08	$09
CMDLIST byte										
0	$03	$03	$03	$01	$03	$01	$01	$01	$04	$04
1	Unit number	Unit number	Unit number	Unit number	Unit number	Unit number	Unit number	Unit number	Unit number	Unit number
2	Status list pointer	Data buffer pointer	Data buffer pointer		Control list pointer				Data buffer pointer	Data buffer pointer
3	Status list pointer	Data buffer pointer	Data buffer pointer		Control list pointer				Data buffer pointer	Data buffer pointer
4	Status code	Block number	Block number		Control code				Byte count	Byte count
5		Block number	Block number						Byte count	Byte count
6		Block number	Block number						*	*
7									*	*
8									*	*

* This parameter is device specific.

❖ *Note:* The Read byte count and the Control call list contents in some SmartPort implementations may not be larger than 767 bytes.

Upon return from the Read call, the byte count bytes will contain the number of bytes actually read from the device.

Table 7-3
Summary of extended commands and parameter lists

Command	Status	ReadBlock	WriteBlock	Format	Control	Init	Open	Close	Read	Write
CMDNUM	$40	$41	$42	$43	$44	$45	$46	$47	$48	$49
CMDLIST byte										
0	$03	$03	$03	$01	$03	$01	$01	$01	$04	$04
1	Unit number	Unit number	Unit number	Unit number	Unit number	Unit number	Unit number	Unit number	Unit number	Unit number
2	Status list pointer	Data buffer pointer	Data buffer pointer		Control list pointer				Data buffer pointer	Data buffer pointer
3	Status list pointer	Data buffer pointer	Data buffer pointer		Control list pointer				Data buffer pointer	Data buffer pointer
4	Status list pointer	Data buffer pointer	Data buffer pointer		Control list pointer				Data buffer pointer	Data buffer pointer
5	Status list pointer	Data buffer pointer	Data buffer pointer		Control list pointer				Data buffer pointer	Data buffer pointer
6	Status code	Block number	Block number		Control code				Byte count	Byte count
7		Block number	Block number						Byte count	Byte count
8		Block number	Block number						*	*
9		Block number	Block number						*	*
10									*	*
11									*	*

* This parameter is device specific.

❖ *Note:* The Read byte count and the Control call list contents in some SmartPort implementations may not be larger than 767 bytes.

Upon return from the Read call, the byte count bytes will contain the number of bytes actually read from the device.

Device-specific SmartPort calls

In addition to the common command set of SmartPort calls already listed, a device may implement its own device-specific calls. Usually, these calls are implemented as a subset of the SmartPort Status or Control call rather than as new commands.

SmartPort calls specific to Apple 3.5 disk drive

Seven Apple 3.5 drive device-specific calls are provided as extensions to the Control call. These device-specific control calls may be used only with the Apple 3.5 drive. To determine whether a device is an Apple 3.5 drive, examine the type and subtype bytes returned from a DIB status call. If the type byte is returned with a value of $01 and the subtype byte is returned with a value of $C0, then the device is an Apple 3.5 drive. Because device-specific calls to the Apple 3.5 drive are implemented as Control calls, only the control code and control list for these calls are defined here. Refer to the SmartPort Control call section earlier in this chapter for information about the command byte and parameter list.

The following information about Eject and SetHook should be treated as an extension to the extended SmartPort Control call.

Eject

Eject ejects the media from a 3.5-inch drive.

Control code	Control list	
$04	Count low byte	$00
	Count high byte	$00

SetHook

SetHook redirects routines internal to the Apple 3.5 drive. The routine to be redirected is referenced by the hook reference number. The address that the routine is to be redirected to is specified by the 3-byte address field in the control list.

Control code	Control list	
$05	Count low byte	$04
	Count high byte	$00
	Hook reference number	$xx
	Address low	$xx
	Address high	$xx
	Address bank	$xx

Valid hook reference numbers and their associated routines are as follows:

Hook reference	Routine
$01	Read Address Field
$02	Read Data Field
$03	Write Data Field
$04	Seek
$05	Format Disk
$06	Write Track
$07	Verify Track

Read Address Field

The Read Address Field routine reads bytes from the disk until it finds the address marks and a sector number specified as input parameters for the routine. The Read Data Field routine reads a 524-byte Macintosh block or 512-byte Apple II block from the disk.

Write Data Field

The Write Data Field routine writes a 524-byte block of data to the disk. For Apple II blocks, the first 12 bytes will be written as zero.

Seek

The Seek routine positions the read and write head over the appropriate cylinder on the disk.

Format

The Format routine writes address marks, data marks, zeroed data blocks, checksum, and end-of-block marks.

Write Track

The Write Track routine is called by the formatter to write one track of empty blocks. The number of blocks written depends on the track that the read and write head is positioned over.

Figure 7-8 demonstrates the physical layout of the format that this command writes.

Address marks			Address field							Gap	Data marks				Data field		EOB marks		
D5	AA	96	Track	Sector	Sides	Format	Checksum	Slip	Slip	5–10 bytes sync	D5	AA	AD	Sector	342 data bytes	Checksum	DE	AA	FF

Figure 7-8
Disk-sector format

Verify

The Verify routine is called by the formatter to verify that the data written by the Write Track routine was written correctly.

ResetHook

ResetHook restores the default address for the hook specified in the control list.

Control code	Control list	
$06	Count low byte	$01
	Count high byte	$00
	Hook reference number	

SetMark

SetMark changes individual bytes in the mark tables. The count field specifies the number of bytes in the mark table to be written plus 1. The start byte references an offset into the mark table to which the new bytes are to be written. Bounds checking is performed to make sure the byte count does not overflow the internal mark table.

Control code	Control list	
$07	Count low byte	$xx
	Count high byte	$00
	Start byte	$xx
	Data	

The default values for the Mark table are as follows:

Value	Byte number	Value	Byte number
$FF	0 sector number	$AA	11
$AD	1 data marks	$DE	12
$AA	1	$FF	13
$D5	3	$FF	14 interheader gap
$FF	4	$FF	15
$FC	5 sync bytes	$FF	16
$F3	6	$FF	17
$CF	7	$96	18 address marks
$3F	8	$AA	19
$FF	9	$D5	20
$FF	10 bit-slip marks	$FF	21

ResetMark

ResetMark restores individual bytes in the mark tables to the default values. The count field defines the number of bytes in the mark table to be restored plus 1. The start field defines where in the mark table the bytes are to be restored.

Control code	Control list	
$08	Count low byte	$xx
	Count high byte	$00
	Start byte	$xx

SetSides

SetSides sets the number of sides of the media to be formatted by the Format call. It supports both single-sided and double-sided media. If the most significant bit of the number of sides field is set to 1, then double-sided media are formatted. If the most significant bit is cleared to 0, then single-sided media are formatted.

Control code	Control list	
$09	Count low byte	$04
	Count high byte	$00
	Number of sides	$nn

SetInterleave

SetInterleave sets the sector interleave to be layed down on the disk by the Format call.

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

SmartPort calls specific to UniDisk 3.5

Five UniDisk 3.5 device-specific calls are provided as extensions to the Control and Status calls. These device-specific calls may be used only with the UniDisk 3.5. To determine whether a device is a UniDisk 3.5, examine the type and subtype bytes returned from a DIB status call. If the type byte is returned with a value of $01 and the subtype byte is returned with a value of $00, then the device is a UniDisk 3.5. Only the control code and control list are defined for calls here implemented as extensions to the Control call. For calls implemented as extensions to the Status call, only the status code and status list are defined. Refer to the sections discussing the SmartPort Control and Status calls earlier in this chapter for more information about these calls.

Eject

Eject ejects the media from a 3.5-inch drive.

Control code	Control list	
$04	Count low byte	$00
	Count high byte	$00

Execute

Execute dispatches the intelligent controller in the UniDisk 3.5 device to execute a 65C02 subroutine. The register setup is passed to the routine to be executed from the control list.

Control code	Control list	
$05	Count low byte	$06
	Count high byte	$00
	Accumulator value	$xx
	X register value	$xx
	Y register value	$xx
	Processor status value	$xx
	Low program counter	$xx
	High program counter	$xx

SetAddress

SetAddress sets the address in the UniDisk 3.5 controller memory space into which the DownLoad call loads a 65C02 routine. The download address must be set to free space in the UniDisk 3.5 memory map.

Control code	Control list	
$06	Count low byte	$02
	Count high byte	$00
	Low byte address	$xx
	High byte address	$xx

Download

Download downloads an executable 65C02 routine into the memory resident in the UniDisk 3.5 controller. The address that the routine is loaded into is set by the SetAddress call. The count field must be set to the length of the 65C02 routine to be downloaded.

Control code	Control list	
$07	Count low byte	$xx
	Count high byte	$xx
	Executable 65C02 routine	

UniDiskStat

UniDiskStat allows an application to get more information about an error that occurs during a read or write operation. It also allows an application to access the 65C02 register state after dispatching the UniDisk 3.5 controller to execute a 65C02 routine via the Execute call.

Memory-mapped I/O addresses internal to the UniDisk 3.5 controller are shown in Figure 7-9 and Tables 7-4 and 7-5.

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

UniDisk 3.5 internal functions

Copy protecting a UniDisk 3.5 is more complicated than protecting a Disk II because the 3.5-inch disk has its own controller. The drive itself (beyond the small 65C02 system that controls it) is somewhat intelligent; performing such operations as stepping the drive to a half track is not possible with the double-sided Sony disk.

The design of the UniDisk 3.5 firmware, however, affords the copy-protection engineer (CPE) tools with which to alter the data on the disk sufficiently to make copying very difficult. In all cases, code or other information is downloaded to the controller's on-board RAM. The firmware provides a defined method for setting RAM, but not for reading it; this increases the difficulty of the copy-protection buster's job. Information downloading is accomplished using the Set_Down_Adr and the DownLoad commands, detailed in the SmartPort documentation.

Further, running nibble-copy programs with the UniDisk 3.5 is difficult to do. Nibble-copy programs typically dump an entire track into memory and then try to make sense of what they have read and duplicate the data stream. The UniDisk 3.5 controller contains only 2K of RAM, and this limitation makes track dumping and copying extremely difficult. A track would have to be dumped in 1 or 2K pieces, and then the pieces would have to be correctly reassembled, processed in host memory, and somehow written in 1 or 2K pieces to the target disk. (The difficulty of creating a reasonable bit copier means that elaborate copy-protection measures may not be necessary and that relatively simple techniques, such as simply changing marks, will suffice.)

Mark table

All address and data marks used by the RdAddr, ReadData, WriteData, and Format routines are located in page zero. The following details the table values and their functions (note that these tables are all reversed from the order in which they appear on the disk):

Function	Address	Default value
Data marks	$008E	$AD, $AA, $D5
Data-sync marks	$0091	$FF, $FC, $F3, $CF, $3F, $FF
Bit-slip marks	$0097	$FF, $AA, $DE
Address marks	$009F	$96, $AA, $D5

The CPE can alter the values in this table and format a disk with the new marks, and read and write operations will recognize sectors with these new marks.

The CPE must, however, be careful when changing the marks. The address, data, and bit-slip marks were chosen so that no bytes in the user's encoded data could be mistaken for them, and the CPE should consider this when changing the marks. Probably the safest marks to alter are the bit-slip marks because the firmware never uses these to try to find a field; they are simply double checks to ensure that synchronization was maintained during a read operation.

The data-sync marks could conceivably be altered and some identifying mark used instead. The CPE should be aware, however, that this field is partially rewritten every time the block is written and that whatever marks are there must guarantee the synchronization of the IWM so that the first data-field mark (normally $D5) can be read.

Hook table

Each major disk-access routine has a JMP instruction to jump through a hook in zero page. Hooks in these routines are collected in a section of zero page known as the *hook table*. Each hook is a 3-byte 65C02 JMP instruction that vectors to the corresponding routine. This allows the CPE to install routines to take the place of ones such as RdAddr and ReadData. Because the hooks are reset when power up occurs or a reset control call is issued, the CPE may preserve the "default" address in a hook, point the hook at his or her own routine, and then have this new routine end by jumping to the old routine. This in effect allows the CPE to insert in his or her own code at strategic points in the disk read and write processes.

The CPE must ensure that any code installed in place of a routine emulates the behavior of the code it replaces. The functional and flag return specifications for the routine must be obeyed; otherwise, higher-level routines will become confused. The "hookable" routines are as follows:

Address	Vector	Routine function
$0072	RdAddr	Find and decode an address field
$0075	ReadData	Find and load a data field into RAM
$0078	WriteData	Write data-sync field marks, data, bit-slip marks
$007B	Seek	Turn motor on and seek the specified track
$007E	Format	Write address and data fields (all zeros)
$0081	WriteTrk	Seek head and write track full of sectors
$0084	Verify	Verify the integrity of an entire track
$0087	Vector	Dispatch a command received from the host

Specifications for each of these routines follow. Note that you will be able to use these functions more effectively if you understand the 3.5-inch disk data format.

When bits of bytes are specified, they are numbered 76543210 and enclosed in brackets [ ]. Also, note that the controller supports *two* drives (drive 0 and drive 1), even though all UniDisks 3.5 use a single-drive configuration (drive 0 only).

UniDisk 3.5 internal routines

RdAddr

Find and decode an address field.

Output Carry set on timeout, checksum, or bit-slip error; clear otherwise.
SectInfo (5 bytes) at $0027 (if carry clear).
On error: $0057[5] is set, meaning address error.

Register requirements None. A, X, Y are not preserved.

This routine waits for the /READY line to go low and then waits for an address field to spin by. A timeout of almost two sector times is allowed. If no address mark is found during this period, or if the data in the address field has a bad checksum, or if the bit-slip bytes are wrong, the routine returns with the carry flag set. If the carry flag is set, then the status byte has the address error bit set. If a good address field was read, its contents are denibblized and the results left in $27–$2B in reverse order from the way they appear on the disk.

ReadData

Find and load a data field into RAM.

Output Carry set if timeout, checksum, or bit-slip error; clear otherwise.
Data read into buffers at $100, $640, and $740.
On error: $0057[3] set for bit slip, [4] set for checksum error.

Register requirements None. A, X, Y are not preserved.

This routine searches for marks identifying a data field. This routine is called immediately after a successful call to RdAddr; therefore, the timeout is extremely short (25 bytes). After a data-field mark is found, the next byte is denibblized and checked to see if it has the correct sector number, and an error is returned if it does not. If the header is all right, the data is read, decoded on the fly, and placed in the three data buffers in reverse order. The bit-slip marks are checked, and an error is generated if they are not as expected. If an error occurs, the status byte $0057 is set to indicate the type of error encountered.

WriteData

Write data-sync field, marks, data, bit-slip marks.

Input	Data in buffers at $100, $640, and $740; checksum.
Register requirements	None. A, X, Y are not preserved.

This routine is called just after RdAddr has found the correct address field. It writes out the data-sync field, the data marks, the nibblized sector number, the data, and the bit-slip marks. At this point, checksumming and pump priming will already have been performed by the WritePrep routine.

Seek

Turn motor on and seek the specified track.

Input	Cyl ($14): new cylinder ($00–$4F) to seek. Drive ($13): drive currently selected. CurCyl ($0D, $0E): cylinder where each head initially rests.
Output	Carry set if seek error; clear otherwise. CurNSect ($1A): number of sectors this cylinder. On error: $0057[1] set for seek error.
Register requirements	None. A, X, Y are not preserved.

If CurCyl[7] for this drive is set, the routine recalibrates the head. The motor is turned on, the stepping direction is set, and the correct number of step pulses is issued.

Format

Write address and data fields (all zeros).

Input	Drive ($13): drive currently selected. FormSides ($63): format a double-sided disk ($80).
Output	Carry set if error; clear otherwise. On error: $005E has $A7 error code.
Register requirements	None. A, X, Y are not preserved.

The formatter turns on the motor and checks whether a write-enabled disk is in the drive. If one is, a sector image is generated and WriteTrk is called. Then Verify is called; if verification fails, up to 10 retries are attempted. If FormSides is set to double sided ($80), both heads are formatted before the head is stepped to the next track.

WriteTrk

Seek head and write track full of sectors.

Input Drive ($13): drive currently selected.
Cyl ($14): cylinder to format.
Side ($16): head number.
FormSides ($63): format a double-sided disk ($80).
Interleave ($62): set physical interleave.

Register requirements None. A, X, Y are not preserved.

This routine seeks the head (if necessary), writes a large group of sync marks (to guarantee the entire track), and then writes the appropriate number of sectors with the correct interleave.

Verify

Verify the integrity of an entire track.

Input CurNSect ($1A): number of sectors this cylinder.

Output Carry set if error; clear otherwise.
On error: $0057 bits are set specifying error.

Register requirements None. A, X, Y are not preserved.

This routine uses RdAddr and ReadData to verify that all sectors on the track are all right, that sectors are unique and that the data fields can be read without error.

Vector

Dispatch a command received from the host.

Input CmdTab ($4C..$54): command from SmartPort.

Output StatusTab ($56..$5B): set to $00.
StatByte ($5E): $80 for no error; error code otherwise.

Register requirements None. A, X, Y are not preserved.

This routine looks in the command table, checks the validity of the command code and parameter count, turns on the drive specified, and jumps to the routine that services the type of command specified. It also sets up the default parameters for the communication routines. If an error is detected in the parameter count or command code, the status byte is set appropriately. The command table looks like this:

```
CMDTab      DFB   Command_Code      ;0 = status, 1 = read, etc.
CMDPCount   DFB   Parameter_Count   ;Logical count for this command
CMDRemain   DS    0,7               ;Call specific
```

The contents of the last 7 bytes depend on the call type. They are the bytes after the unit number in the SmartPort command list.

Memory allocation

The firmware does not use page 5 of RAM or the top 64 bytes of the zero page. The CPE is free to install patches and other code in $0500–$05FF and $00C0–$00FF. Figure 7-9 shows the entire UniDisk 3.5 memory map as well as firmware RAM space use.

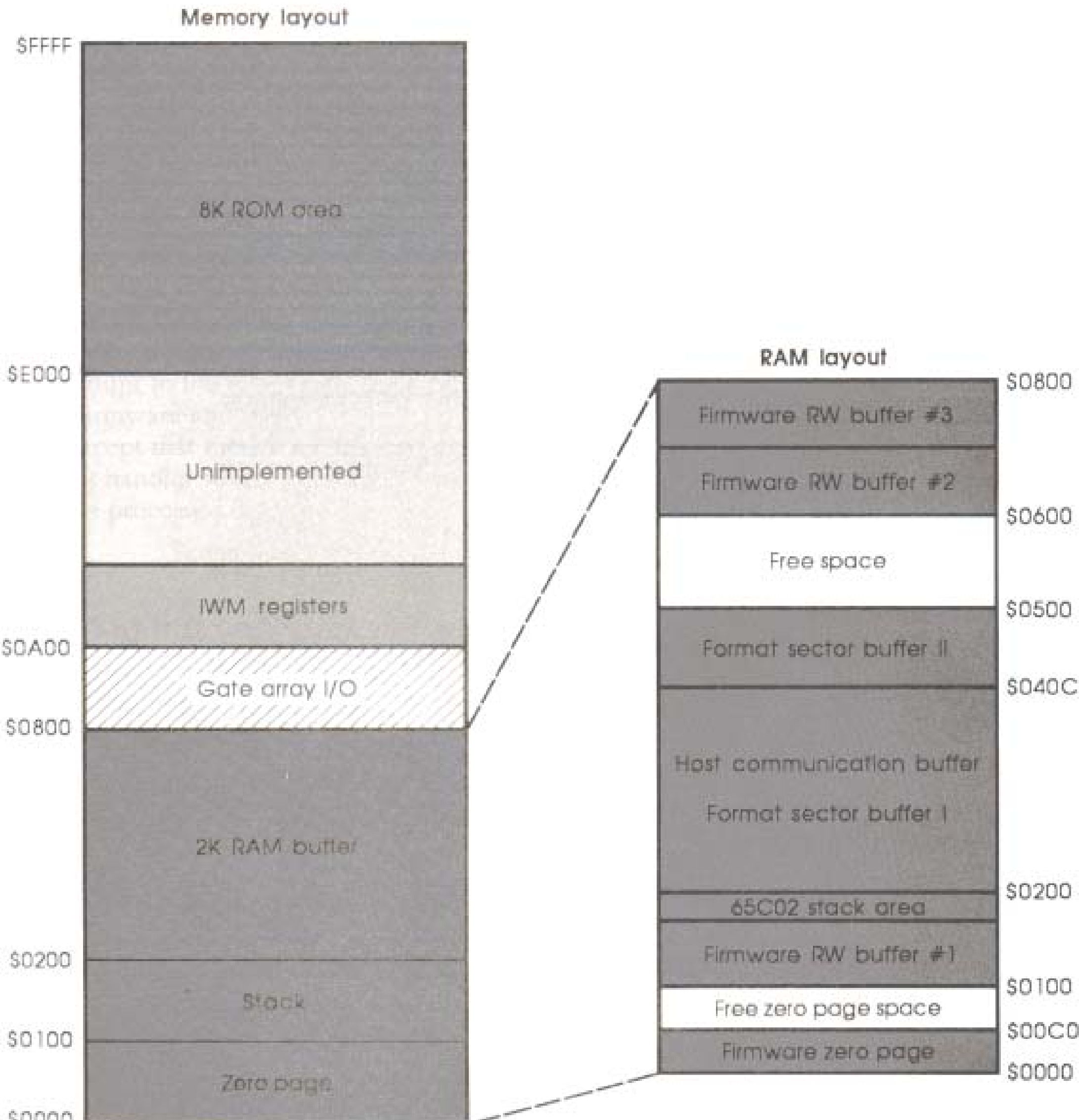


Figure 7-9
UniDisk 3.5 memory map

Table 7-4
UniDisk 3.5 gate array I/O locations

Function	data4	data3	data2	data1	data0
Read $800	LASTONE/	BUSEN/	WRREQ	/GATENBL	HDSEL
Wrt $800	TRIGGER	ENBUS	PH3EN	IWMDIR	HDSEL
Read $801	SENSE	BLATCH1	BLATCH2	LIRONEN	CAO
Wrt $801	/RSTIWM	/BLATCH CLR1	/BLATCH CLR2	DRIVE1	DRIVE2

Table 7-5
UniDisk 3.5 IWM locations

Location	Specific label	IWMDIR = 0 (drv)	IWMDIR = 1 (host)
$0A00	PHASE0 reset	CA0 reset	/BSY handshake
$0A01	PHASE0 set	CA0 set	/BSY handshake
$0A02	PHASE1 reset	CA1 reset	
$0A03	PHASE1 set	CA1 set	
$0A04	PHASE2 reset	CA2 reset	
$0A05	PHASE2 set	CA2 set	
$0A06	PHASE3 reset	LSTRB reset	
$0A07	PHASE3 set	LSTRB set	
$0A08	MOTOROFF		
$0A09	MOTORON		
$0A0A	ENABLE1		
$0A0B	ENABLE2		
$0A0C	L6 reset		
$0A0D	L6 set		
$0A0E	L7 reset		
$0A0F	L7 set		

ROM disk driver

The ROM disk is a plug-in card that houses ROM that may be organized into blocks to emulate a disk device or provide space for ROM-based programs. Although the SmartPort has no built-in ROM disk, SmartPort does support an external ROM disk driver.

Installing a ROM disk driver

The driver for a ROM disk must reside at address $F0/0000. The ROM disk may occupy only the address space from $F0/0000 through $F7/FFFF. The base address of the driver must contain the ASCII string ROMDISK in uppercase letters with the most significant bit on. Entry to the ROM disk driver is through address $F0/0007. The SmartPort firmware will search for a ROM disk driver during the boot process while assigning unit numbers to each of the SmartPort devices. If the SmartPort finds the ASCII string ROMDISK at address $F0/0007, it executes an Initialization call to the ROM disk driver via the ROM disk entry point. If the ROM disk returns with no error, the ROM disk driver is installed in the SmartPort device chain. If the ROM disk Initialization call returns an error, the ROM disk driver is not installed in the SmartPort device chain. Note that the ROM disk driver is called via a JSL instruction in 8-bit native mode.

Passing parameters to a ROM disk

Call parameters are passed to the ROM disk from the SmartPort through fixed memory locations in absolute zero page. All input to device-specific drivers is passed in an extended format, even for standard SmartPort calls, so that the call parameters can always be found in fixed locations. Note that standard calls are not changed into extended calls; only parameter organization is affected.

Some parameters do not occupy contiguous memory when they are presented in an extended format because the order of parameters has been prepared so the parameters can be transmitted over the SmartPort bus to intelligent devices. Absolute zero page locations $40 to 62 are saved by the SmartPort prior to their dispatch to the ROM disk and are restored by the SmartPort after their return from the driver. Thus, these locations are available for use by the ROM disk driver.

Call parameters are passed to the ROM disk driver as follows:

Location	Parameters	Call type
$42	Buffer address (bits 0 to 7)	All
$43	Buffer address (bits 8 to 15)	All
$44	Buffer address (bits 16 to 23)	All
$45	Command	All
$46	Parameter count	All
$47	Buffer address (bits 24 to 31)	All
$48	Extended block (bits 0 to 7)	ReadBlock and WriteBlock
	Status code or control code	Status and Control
	Byte count (bits 0 to 7)	Read and Write
$49	Extended block (bits 8 to 15)	ReadBlock and WriteBlock
	Byte count (bits 8 to15)	Read and Write
$4A	Extended block (bits 16 to 23)	ReadBlock and WriteBlock
	Address pointer (bits 0 to 7)	Read and Write
$4B	Extended block (bits 24 to 31)	ReadBlock and WriteBlock
	Address pointer (bits 8 to 15)	Read and Write
$4C	Address pointer (bits 16 to 23)	Read and Write
$4D	Address pointer (bits 24 to 31)	Read and Write

Parameters returned to the application from the ROM disk driver are passed in absolute zero page locations as follows:

Location	Output parameter passed
$000050	Error code
$000051	Low byte of count of bytes transferred to host
$000052	High byte of count of bytes transferred to host

All I/O information passed between the application making the SmartPort call and the ROM disk driver is passed through the buffer specified in the parameter list.

ROM organization

ROM for a ROM disk must contain the ROM disk signature string as well as a ROM disk driver. A map of the ROM address space when portions of ROM are organized as blocks is shown in Figure 7-10.

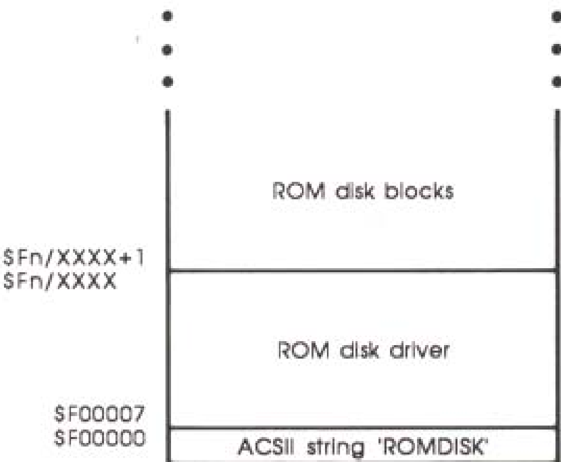


Figure 7-10
The ROM disk

A block diagram of a ROM disk that occupies 128K of ROM (including the driver itself) is shown in Figure 7–11. Note that no ROM space has been reserved for toolset expansion in this example.

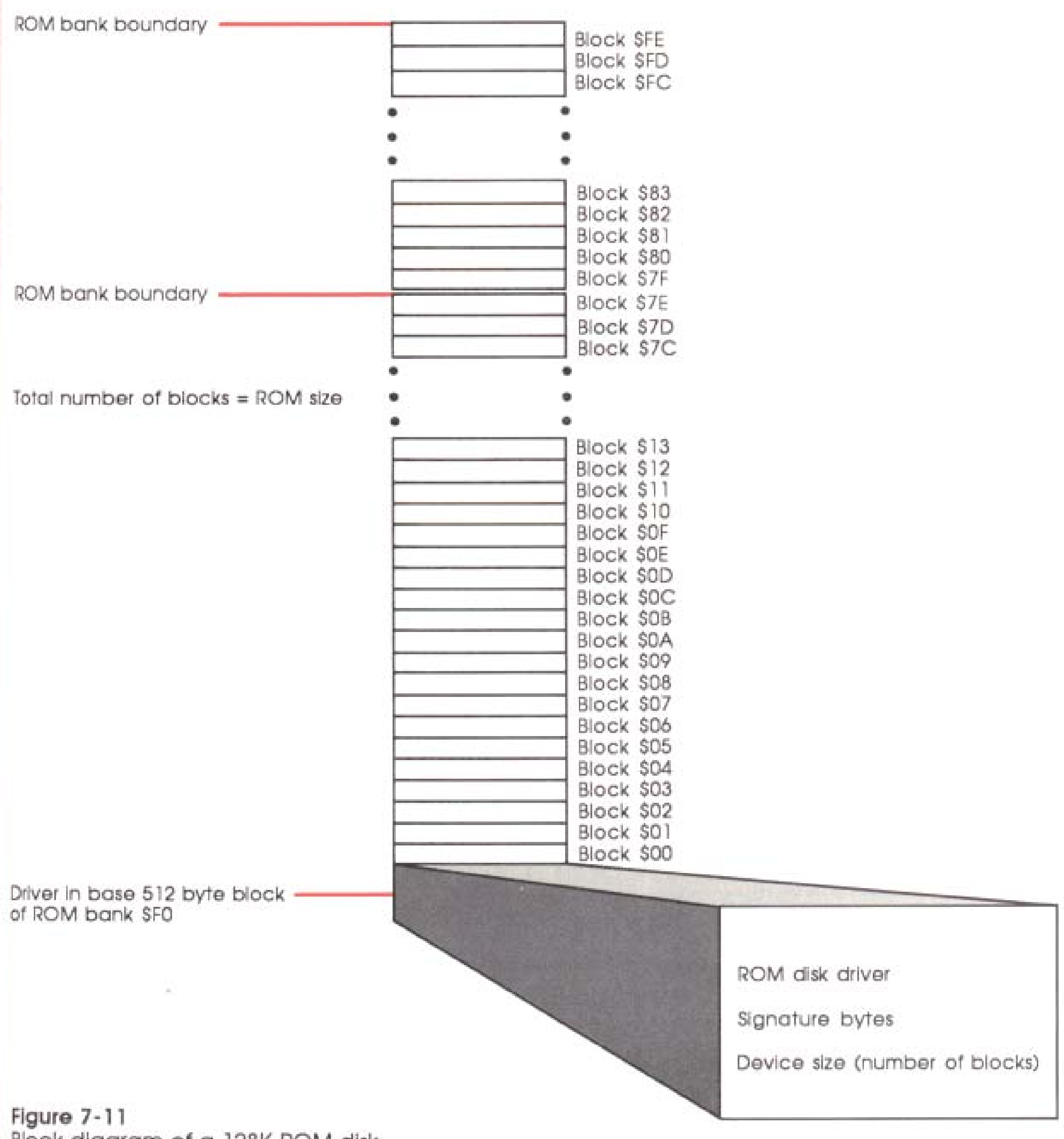


Figure 7-11
Block diagram of a 128K ROM disk

Summary of SmartPort error codes

SmartPort error codes are summarized in Table 7-6.

Table 7-6
SmartPort error codes

Acc value	Error type	Description
$00	No error	No error occurred.
$01	BADCMD	A nonexistent command was issued.
$04	BADPCNT	A bad call parameter count was given. This error occurs only when the call parameter list is not properly constructed.
$06	BUSERR	A communications error occurred in the IWM.
$11	BADUNIT	An invalid unit number was given.
$1F	NOINT	Interrupt devices are not supported.
$21	BADCTL	The control or status code is not supported by the device.
$22	BADCTLPARM	The control list contains invalid information.
$27	IOERROR	The device encountered an I/O error.
$28	NODRIVE	The device is not connected. This error can occur if the device is not connected but its controller is.
$2B	NOWRITE	The device is write protected.
$2D	BADBLOCK	The block number is not present on the device.
$2E	DISKSW	Media has been swapped (extended calls only).
$2F	OFFLINE	The device is off line or no disk is in drive.
$30–$3F	DEVSPEC	These are device-specific error codes.
$40–$4F	RESERVED	Reserved for future use.
$50–$5F	NONFATAL	A device-specific soft error occurred. The operation was completed successfully, but an abnormal condition was detected.
$60–$6F	NONFATAL	These errors are the same as the errors in the $20–$2F range. Bit 6 indicates a soft error.

The SmartPort bus

The SmartPort bus is a daisy chain configuration of intelligent devices, sometimes called *bus residents,* connected to the disk port of the host CPU. A Disk II device may be physically connected to the end of the SmartPort device chain on the Apple IIGS, and its operation will be transparent to the host firmware. The Disk II device is dormant when a SmartPort bus resident is addressed. The number of bus residents that can be supported is limited by supply-power and IWM-drive considerations. Although the software supports up to 127 bus residents, power requirements usually limit the maximum number of residents to 4.

Drive selection is performed through the firmware. The command string contains a byte specifying the device to be accessed. These device ID bytes are assigned by the SmartPort at bus reset.

Two functions are strictly hardware invoked: bus reset and bus enable. Both of these conditions are invoked through combinations of phase lines on the disk port that never occur under normal Disk II operation (Both functions involve invoking opposing phases, which is pointless on a Disk II.) This allows a Disk II device and other bus residents to stay out of each other's way. The bus reset and enable functions are summarized below.

Function	PH3	PH2	PH1	PH0
Enable	1	X	1	X
Reset	0	1	0	1

The state of the PH0 line during the enable function can be either a 1 or a 0 because PH0 is used as a REQ handshake line cycled on a packet basis when the bus is enabled. ACK is sensed from the device through the IWM write-protect sense status.

How SmartPort assigns unit numbers

The assignment of unit numbers is initiated by executing a call to the slot 5 boot entry point. This assignment always begins with a bus reset. The reset flips a latch on all bus residents, which causes the daisy-chained phase 3 line to become low. This makes all daisy-chained devices incapable of receiving the bus-enable signal, which requires phase 3 to be high.

The host then sends the ID definition command. Whenever a device receives this command (with Enable), it assigns the unit number embedded in the command string as its own unit number. Thereafter it will not respond to any command string with a unit number other than that given it in the ID definition command.

Upon completing the ID definition command, the bus resident reenables the phase 3 line, allowing the next resident to receive its ID definition command. This process continues so long as there are bus residents. The last bus resident in the device chain returns an exception, indicating that it is the last bus resident.

Although Disk II devices are connected to the disk port, they are not bus residents and do not respond to the ID definition command. A resident determines that it is the last intelligent device in the chain by sensing a signal, normally unused in Disk II operations, which is grounded by all intelligent devices. If no bus resident or Disk II device is daisy chained to the port, the phase 3 line is read as high.

SmartPort–Disk II interaction

The disk port built into the Apple IIGS supports daisy-chained 5.25-inch disks (UniDisk 5.25, Disk II, or DuoDisk) by sharing the same disk port hardware between two different ROM slot areas. The slot 5 ROM area contains the SmartPort interface and ProDOS block device driver, and the slot 6 ROM area contains the Disk II interface. The Disk II device is enabled by the disk port signal /ENABLE2. The SmartPort must activate the /ENABLE2 line to communicate with intelligent bus residents. If this line were not intercepted before being passed to daisy-chained devices, any attempt to talk to devices on the bus would result in spurious operation of the Disk II at the end of the chain.

For the Disk II to remain aloof from SmartPort activity, each resident must gate the /ENABLE2 line so that whenever any SmartPort bus resident is enabled (PHASE1 and PHASE3), any Disk II at the end of the chain is disabled. In other words, the /ENABLE2 line is passed to daisy-chained devices only when either PHASE1 or PHASE3 is low:

BUS ENABLE (PH1 and PH3)	/ENABLE2 (daisy chained)
PHASE1=0 or PHASE3=0	/ENABLE2
PHASE1=1 and PHASE3=1	Deasserted (high)

Other considerations

All intelligent residents try to process every command packet sent over the bus; a resident responding only if it recognizes its own ID, type, and subtype encoded in the packet. The device type and command are used by the device to arbitrate between extended and standard packets. Thus, one resident can tell when some other resident is being accessed or if the packet type (extended or standard) is compatible with the device. A device controller can therefore reduce its power consumption when it is not being constantly accessed.

Extended and standard command packets

The number of bytes passed over the SmartPort bus in a standard command packet is the same as the number contained in an extended command packet. Standard SmartPort command parameter lists can consist of up to 9 bytes. Extended SmartPort command parameter lists can consist of up to 11 bytes. The command packet was designed for a maximum of 9 bytes of information. The first 2 bytes always contain the SmartPort command number and parameter count. The remaining 7 bytes consist of 7 bytes of the parameter list starting with the third byte for standard commands or the fifth byte for extended commands; 7 bytes from the parameter list always are copied into the command packet, even though the parameter list for the current command may consist of fewer than 7 bytes.

SmartPort bus flow of operations

The general flow of control in the SmartPort is illustrated in Figure 7-12.

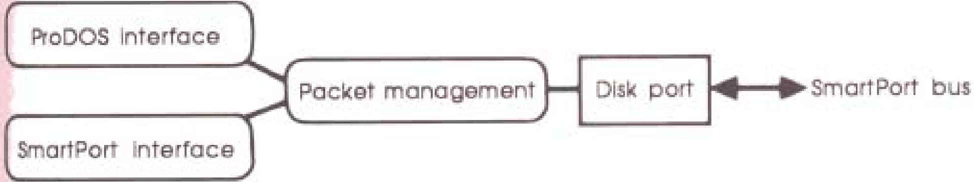


Figure 7-12
SmartPort control flow

Whenever a call is made to the SmartPort device driver that uses the SmartPort bus, the command table sent to the device driver is converted into a *command packet* before being sent to the device. The results of the call are also sent back from the device in a *packet*. All data sent over the bus is placed in these packets.

❖ *Note:* Each byte of the packet is a 7-bit quantity (bit 7 is always set), a limitation imposed by the IWM. All data sent is converted from 8-bit quantities to 7-bit quantities before transmission.

The information of the packet can be broken down into the following categories:

- general overhead
- source and destination IDs
- contents type and auxiliary (aux) type
- contents status
- contents

The identifiers are 7-bit quantities assigned sequentially according to the device's position in the chain. The host is always ID=0. Because every byte in the packet has the most significant bit set, the host is $80, the first device in the chain is $81, and so on.

The contents type consists of a type and aux type byte. Three contents types are currently defined: Type = $80 is a command packet, type = $81 is a status packet, and type = $82 is a data packet. Bit-6 is the command byte, and the aux type byte defines the packet as either extended or standard. Aux type = $80 indicates a standard packet, and $C0 indicates an extended packet. Command = $8X indicates a standard packet, and $CX indicates an extended packet.

The contents byte is used for status and data packets. It contains the error code for read and write operations. The SmartPort returns the contents byte as an error code for the call.

The contents itself consists of bytes of 7 bits (high bit set) of encoded data. Preceding the bytes themselves are two length bytes. If the number of content bytes is BYTECOUNT, then the first byte is defined as BYTECOUNT DIV 7, and the second byte is defined as BYTECOUNT MOD 7. In other words, the first byte specifies the number of groups of 7 bytes of content, and the second is the remainder. Note that the second byte will never have a value greater than 6. Both these bytes have their most significant bit set.

The general overhead bytes are packet begin and end marks, sync bytes (6, to ensure correct synchronization of the IWMs), and a checksum. The checksum is computed by exclusive ORing all the content data bytes (8 bits) and the IDs, type bytes, status bytes, and length bytes. The checksum is 8 bits sent as 16.

Figure 7-13 demonstrates the sequence of signal transitions that define the protocol for executing a read from a device. The signal transition points are described below.

1. Host asserts REQ when ACK is negated; command packet is coming from host.
2. Host enables IWM and sends packet to device.
3. Device deasserts ACK, signaling host that packet was received.
4. Host responds by deasserting REQ.
5. Device asserts ACK when it is ready to send response packet to host.
6. Host asserts REQ when it is ready to receive response packet from device.
7. Device enables IWM and sends response packet to host.
8. Device deasserts ACK at end of packet.
9. Host deasserts REQ when packet is received.
10. Device asserts ACK to indicate it is ready to receive a command.

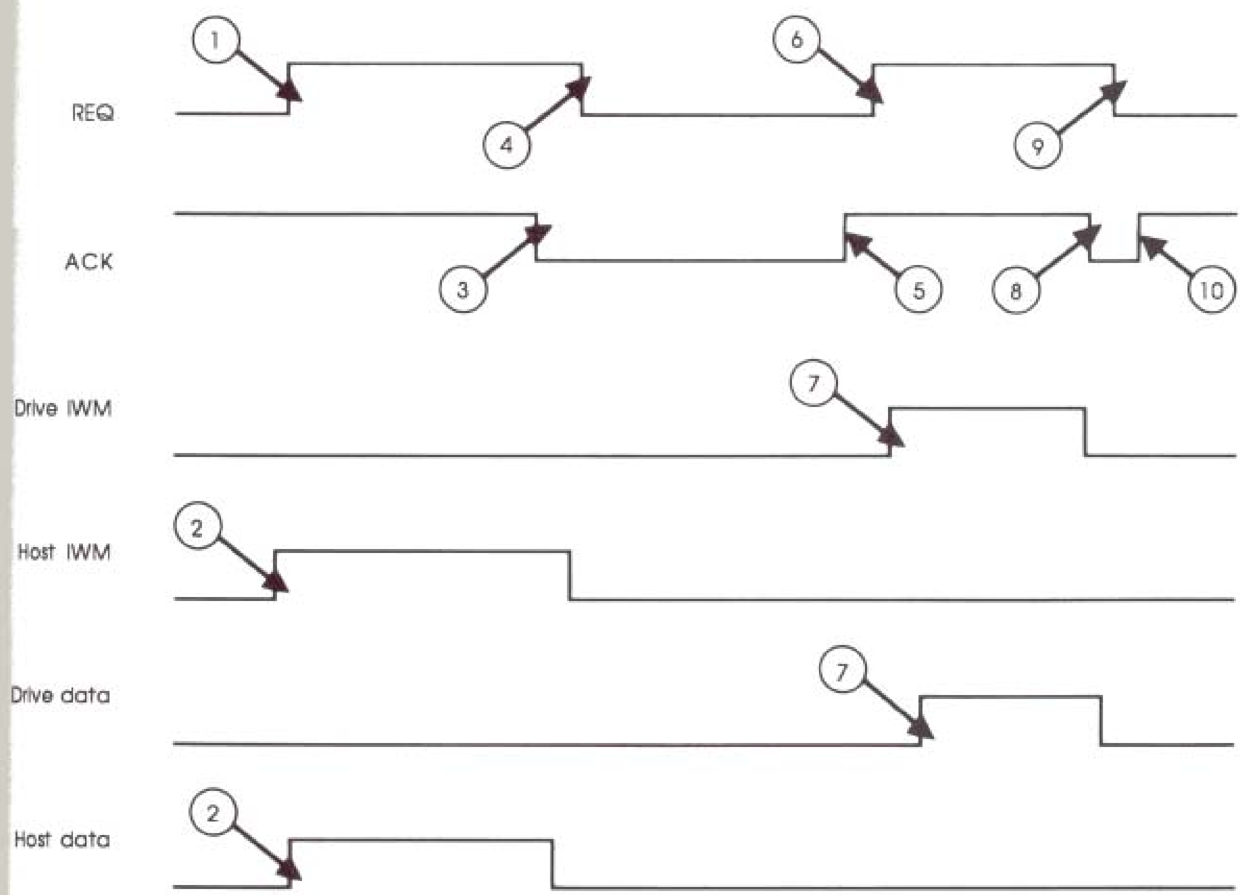


Figure 7-13
SmartPort bus communications: read protocol

Figure 7-14 demonstrates the sequence of signal transitions that define the protocol for executing a write to a device. The signal transition points are described below.

1. Host asserts REQ when ACK is negated and command packet is coming from host.
2. Command packet is sent.
3. Device asserts ACK, signaling it received the packet.
4. Host negates ACK, finishing the command handshake.
5. When REQ is negated and device is ready to receive data, device negates ACK.
6. When ACK is negated and host is ready to send, host asserts REQ.
7. Host sends write data.
8. Device asserts ACK, signaling it received the REQ.
9. Host negates REQ, allowing device to write data to its media.

10. Device negates ACK and writes data to its media.
11. Host responds to negated ACK by asserting REQ, signaling it is ready for status.
12. Device responds to REQ by sending status to host.
13. Device asserts ACK, signaling status has been sent.
14. Host acknowledges receipt of status by negating REQ.
15. Device negates ACK when it is ready for the next command.

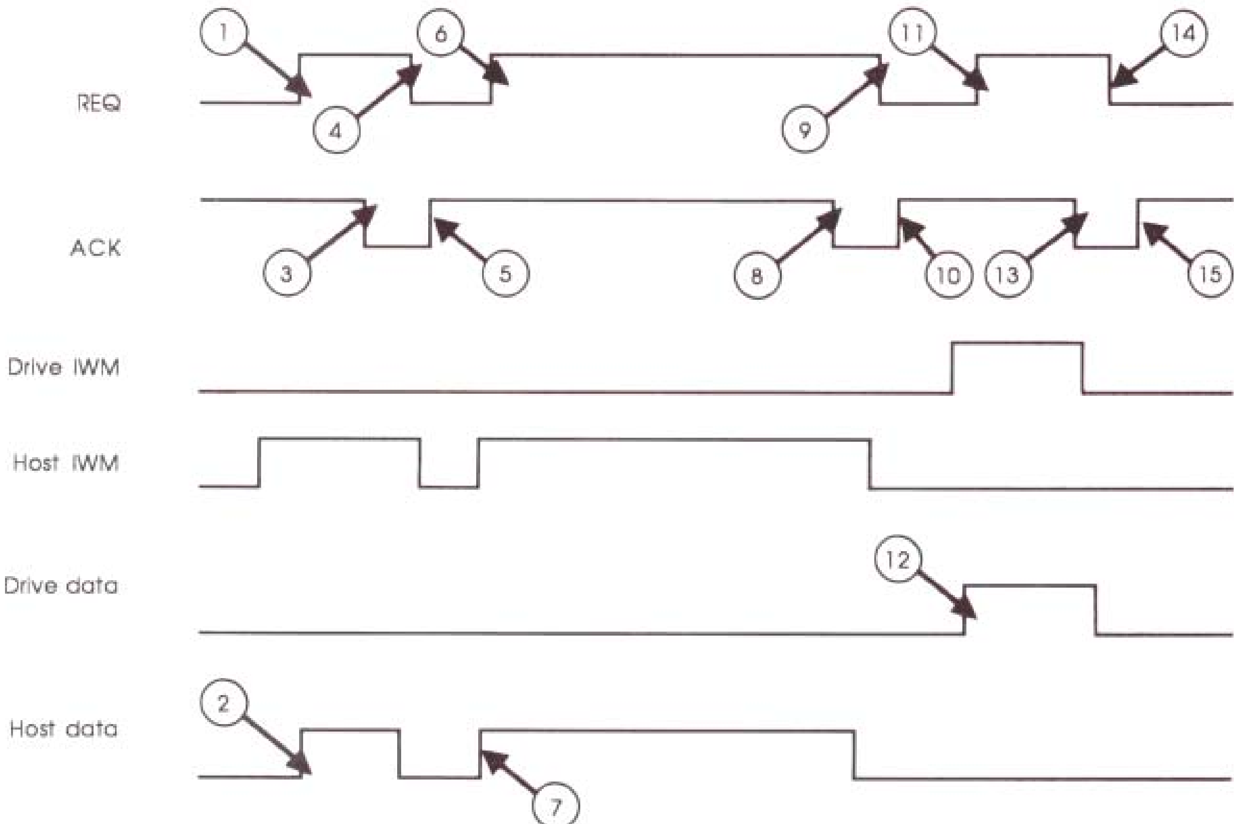


Figure 7-14
SmartPort bus communications: write protocol

Figure 7-15 illustrates that a command packet contains as few as zero and as many as 767 data bytes. Each packet of 7 data bytes is encoded in a specific manner, described below, to assure that each data byte that is part of the packet has its most significant bit set. To allow all possible bit combinations to be transmitted in this manner, it is necessary to transmit 8 data bytes of encoded information for every 7 bytes of data. If there is not an even multiple of 7 bytes in the total data block to be sent, then the remaining 0 to 6 data bytes are encoded and sent, preceding the packets of 7 encoded bytes, as 2 to 7 data bytes as described below.

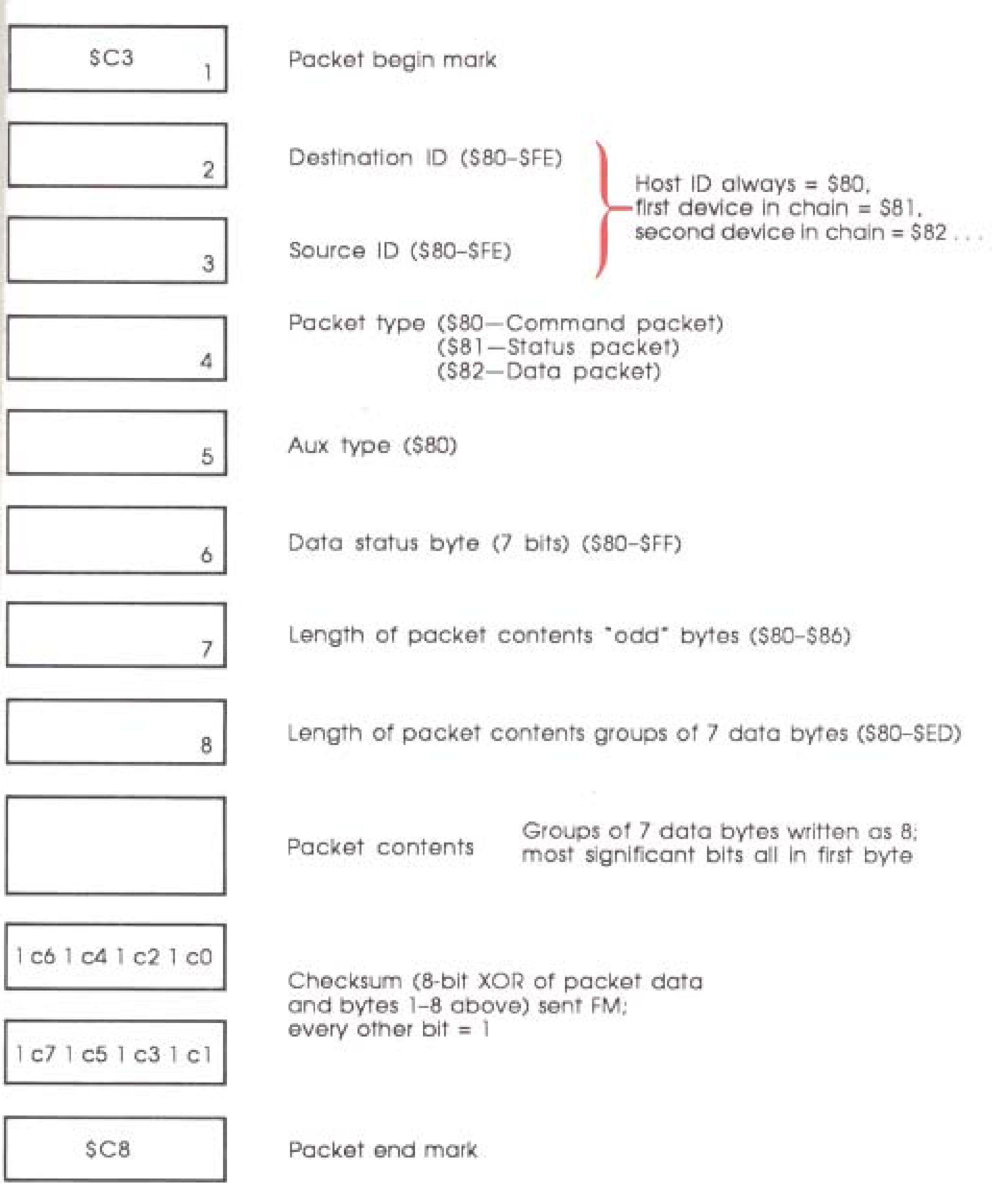


Figure 7-15
SmartPort bus packet format

For each group of 7 data bytes in the block to be sent, take the bits of which those bytes are composed and rearrange them as shown in Figure 7-16. This changes the 7 bytes of input data into 8 bytes of encoded data, in which each output data byte has its most significant bit set.

Figure 7-16
SmartPort bus packet contents

As Table 7-7 shows, the first byte contains the most significant bit of each of the 7 data bytes, the second byte contains the seven least significant bits of the first data byte, the third byte contains the seven least significant bits of the third data byte, and so on for a total of 8 bytes of encoded data. This data is transmitted with the byte containing the most significant bits first, followed by each of the other 7 encoded data bytes in turn. Thus, you can see that if there are fewer than 7 data bytes in an odd group, fewer than 8 bytes of encoded data will be required to transmit this odd group.

Table 7-7
Data byte encoding table

Top bits byte	$d1_7$	$d2_7$	$d3_7$	$d4_7$	$d5_7$	$d6_7$	$d7_7$
Byte 1	$d1_6$	$d1_5$	$d1_4$	$d1_3$	$d1_2$	$d1_1$	$d1_0$
Byte 2	$d2_6$	$d2_5$	$d2_4$	$d2_3$	$d2_2$	$d2_1$	$d2_0$
Byte 3	$d3_6$	$d3_5$	$d3_4$	$d3_3$	$d3_2$	$d3_1$	$d3_0$
Byte 4	$d4_6$	$d4_5$	$d4_4$	$d4_3$	$d4_2$	$d4_1$	$d4_0$
Byte 5	$d5_6$	$d5_5$	$d5_4$	$d5_3$	$d5_2$	$d5_1$	$d5_0$
Byte 6	$d6_6$	$d6_5$	$d6_4$	$d6_3$	$d6_2$	$d6_1$	$d6_0$
Byte 7	$d7_6$	$d7_5$	$d7_4$	$d7_3$	$d7_2$	$d7_1$	$d7_0$

The number of bytes in the odd group is the remainder of the number of data bytes in the packet divided by 7. When encoding the odd bytes, assume that the rest of the data bytes making up a group of 7 bytes all contain zeros. Also note that if there are no odd bytes (that is, if the packet size divides by 7 evenly with no remainder), the odd-bytes group is simply omitted. Similarly, if the number of bytes in the packet is less than 7, there will be no encoded packets of 7 bytes, but only an odd-bytes group will be sent.

For example, if you are sending a 512-byte packet, the number of groups of 7 bytes is 73, with a remainder of 1. Therefore, the first data byte will be sent as an odd group, followed by 73 groups of 7 bytes each. The groups of 7 bytes will be encoded as indicated above and the odd bytes (byte number 1 of the packet, data bits 7..0) will be sent as shown in Figure 7-17.

$d1_{bits\ 7..0}$ $d2_{bits\ 7..0}$ $d3_{bits\ 7..0}$ $d4_{bits\ 7..0}$ $d5_{bits\ 7..0}$ $d6_{bits\ 7..0}$ $d7_{bits\ 7..0}$

Figure 7-17
Bit layout of a 7-byte data packet

Top bits byte	1	$d1_7$	0	0	0	0	0	0
Byte 1	1	$d1_6$	$d1_5$	$d1_4$	$d1_3$	$d1_2$	$d1_1$	$d1_0$

Figure 7-18
Transmitting a 1-byte data packet

Note that the top bits for data bytes 2 through 7 in this example are set to zero, and the data bytes that would have contained the least significant data bits of bytes 2 through 7 are not transmitted. This is simply a special case of an instance of a group of 7 bytes.

Tables 7-8 and 7-9 provide a visual summary of the contents of the standard and extended command packets. Where there is an asterisk in the table, the value of the corresponding byte position is undefined and should be ignored by the device.

Table 7-8
Standard command packet contents

Byte	Status	ReadBlock	WriteBlock	Format	Control	Init	Open	Close	Read	Write
1	$00	$01	$02	$03	$04	$05	$06	$07	$08	$09
2	Param count	Param count	Param count	Param count	Param count	Param count	Param count	Param count	Param count	Param count
3	Byte 3 of param list	Byte 3 of param list	Byte 3 of param list	•	Byte 3 of param list	•	Byte 3 of param list	Byte 3 of param list	Byte 3 of param list	Byte 3 of param list
4	Byte 4 of param list	Byte 4 of param list	Byte 4 of param list	•	Byte 4 of param list	•	Byte 4 of param list	Byte 4 of param list	Byte 4 of param list	Byte 4 of param list
5	•	Byte 5 of param list	Byte 5 of param list	•	•	•	•	•	Byte 5 of param list	Byte 5 of param list
6	•	Byte 6 of param list	Byte 6 of param list	•	•	•	•	•	Byte 6 of param list	Byte 6 of param list
7	•	Byte 7 of param list	Byte 7 of param list	•	•	•	•	•	Byte 7 of param list	Byte 7 of param list
8	•	•	•	•	•	•	•		Byte 8 of param list	Byte 8 of param list
9	•	•	•	•	•	•	•		Byte 9 of param list	Byte 9 of param list

• A byte with an indeterminate value; the device should ignore the byte.

Table 7-9
Extended command packet contents

Byte	Status	ReadBlock	WriteBlock	Format	Control	Init	Open	Close	Read	Write
1	$40	$41	$42	$43	$44	$45	$46	$47	$48	$49
2	Param count	Param count	Param count	Param count	Param count	Param count	Param count	Param count	Param count	Param count
3	Byte 5 of param list	Byte 5 of param list	Byte 5 of param list	*	Byte 5 of param list	*	Byte 5 of param list	Byte 5 of param list	Byte 5 of param list	Byte 5 of param list
4	Byte 6 of param list	Byte 6 of param list	Byte 6 of param list	*	Byte 6 of param list	*	Byte 6 of param list	Byte 6 of param list	Byte 6 of param list	Byte 6 of param list
5	*	Byte 7 of param list	Byte 7 of param list	*	*	*	*	*	Byte 7 of param list	Byte 7 of param list
6	*	Byte 8 of param list	Byte 8 of param list	*	*	*	*	*	Byte 8 of param list	Byte 8 of param list
7	*	Byte 9 of param list	Byte 9 of param list	*	*	*	*	*	Byte 9 of param list	Byte 9 of param list
8	*	*	*	*	*	*	*		Byte 10 of param list	Byte 10 of param list
9	*	*	*	*	*	*	*		Byte 11 of param list	Byte 11 of param list

* A byte with an indeterminate value; the device should ignore the byte.