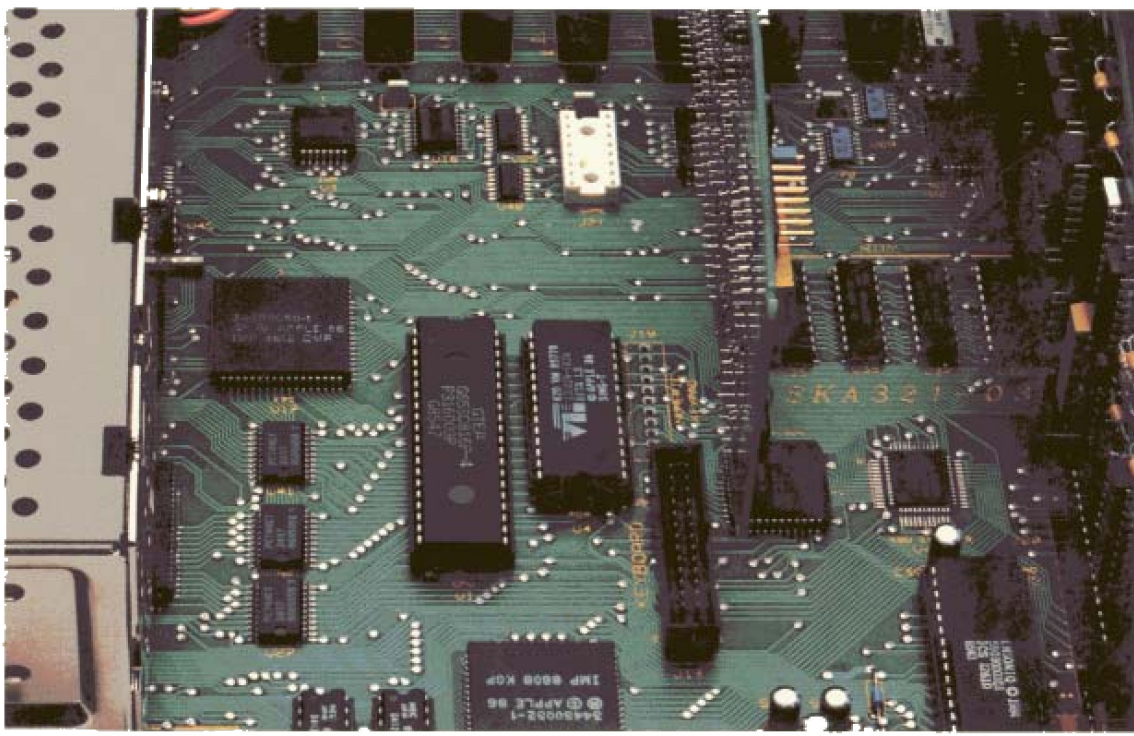
Apple® II

Apple IIGS™ Firmware Reference



APPLE COMPUTER, INC.

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

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.)

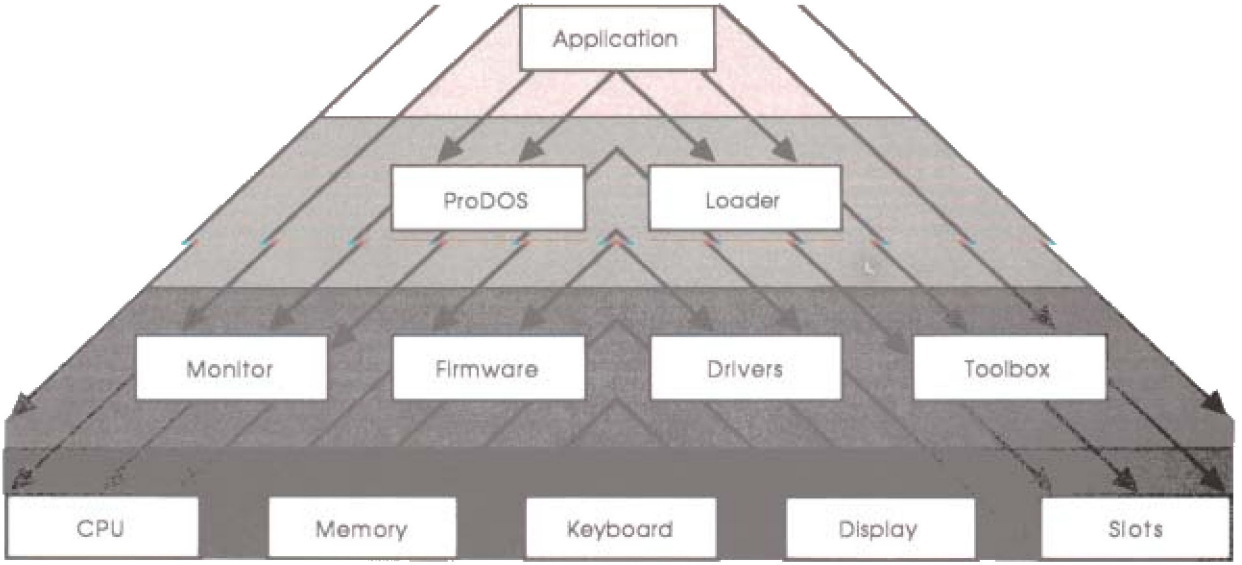


Figure 1-1
Levels of program operation

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).

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).

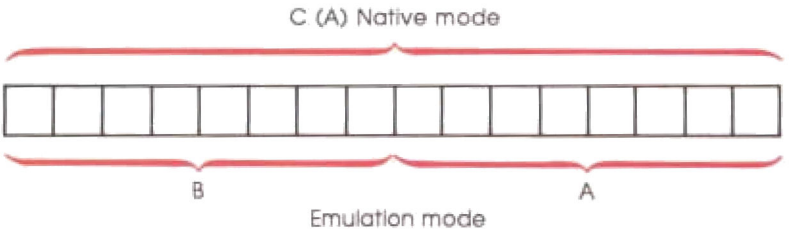


Figure 2-2
Accumulator for emulation and native modes

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
*.1008 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 0FF 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.
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=l 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.

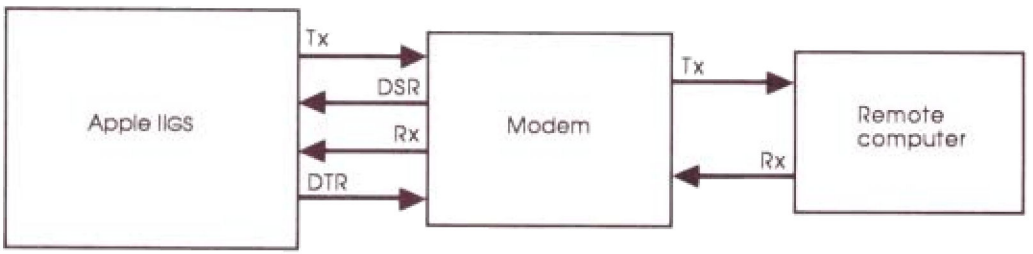


Figure 5-1
Handshaking when DTR/DSR option is turned on

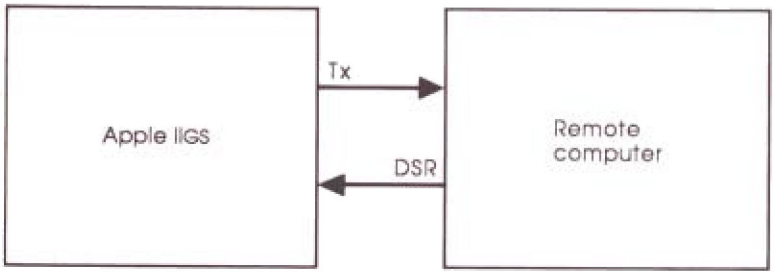


Figure 5-2
Handshaking when DTR/DSR option is turned off

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.

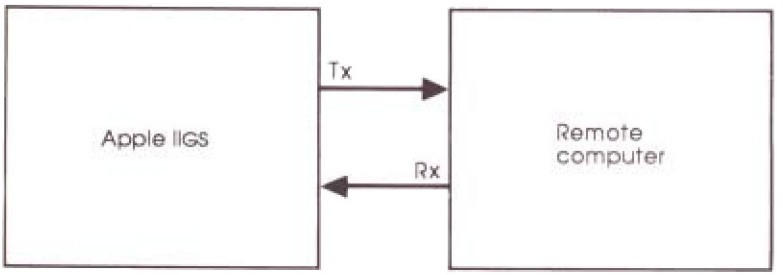


Figure 5-3
Handshaking via XON/XOFF

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):

Cn00 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

System speed = fast
DBR = $00
Native mode (that is, m = 0, x = 0, e = 0)

On exit

System speed = fast
DBR = $00
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.

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)
[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

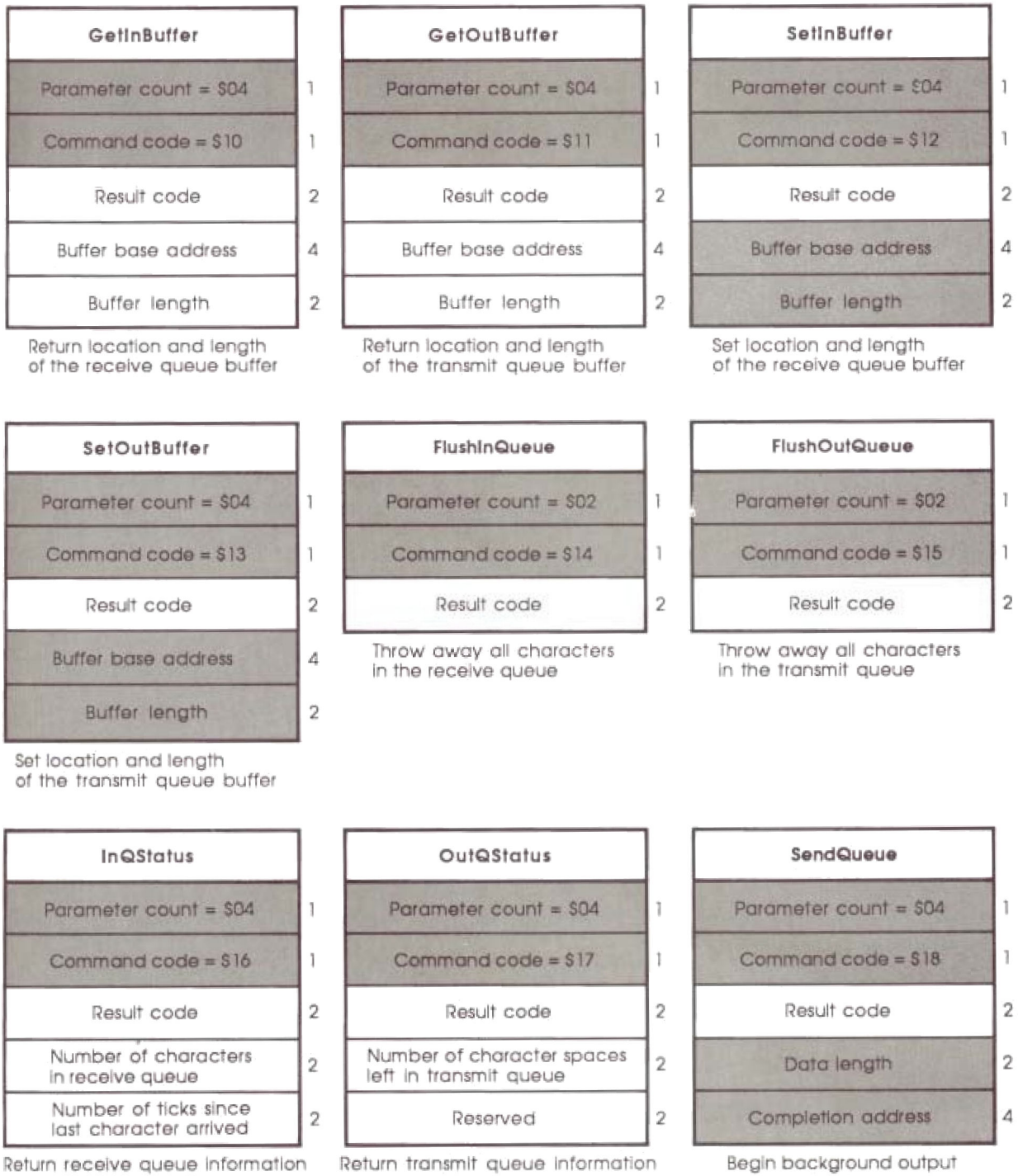


Figure 5-4
Summary of extended serial-port buffer commands

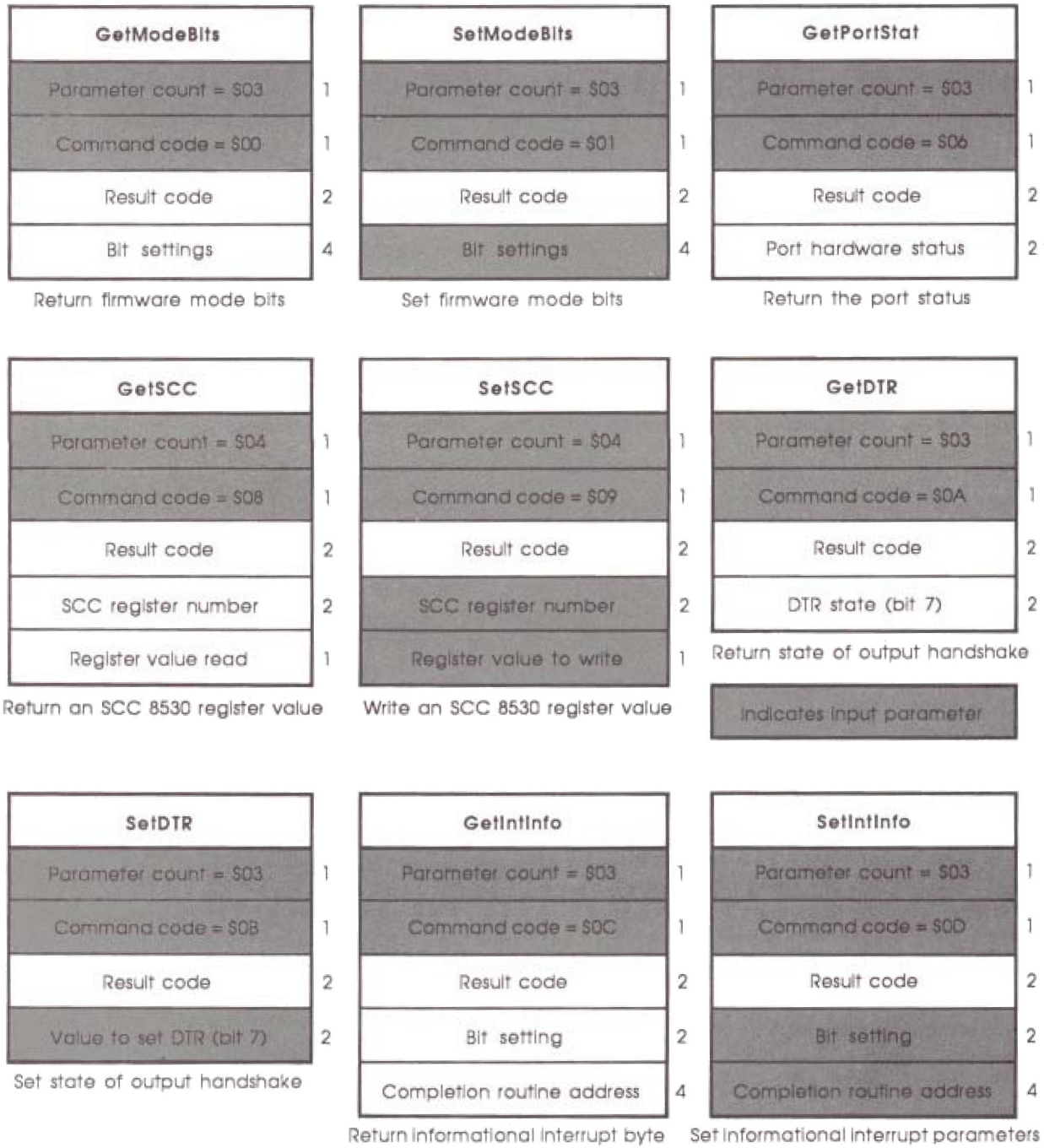


Figure 5-5
Summary of extended serial-port mode and hardware control commands

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).

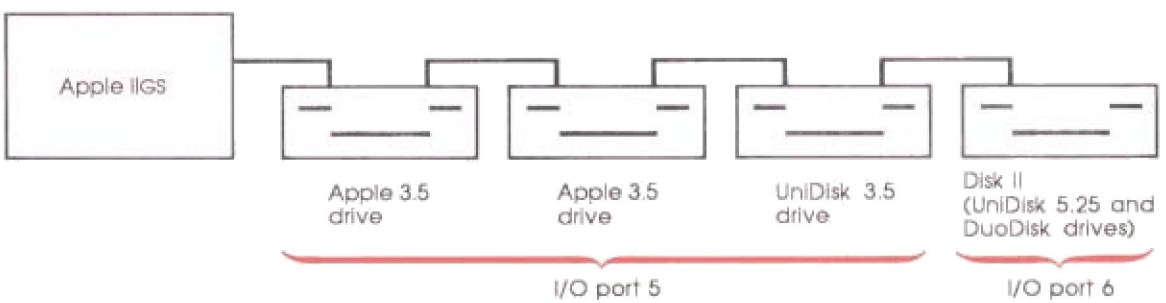


Figure 6-1
Order of disk drives on Apple IIGS disk ports

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).

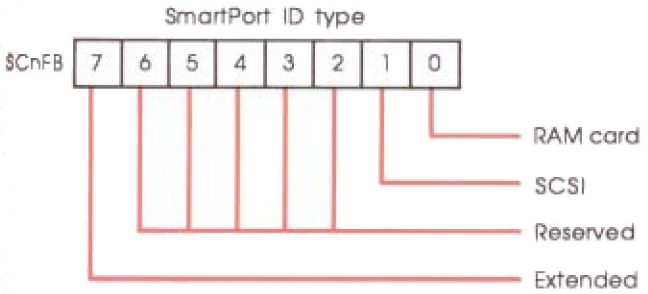


Figure 7-1
SmartPort ID type byte

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:

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.

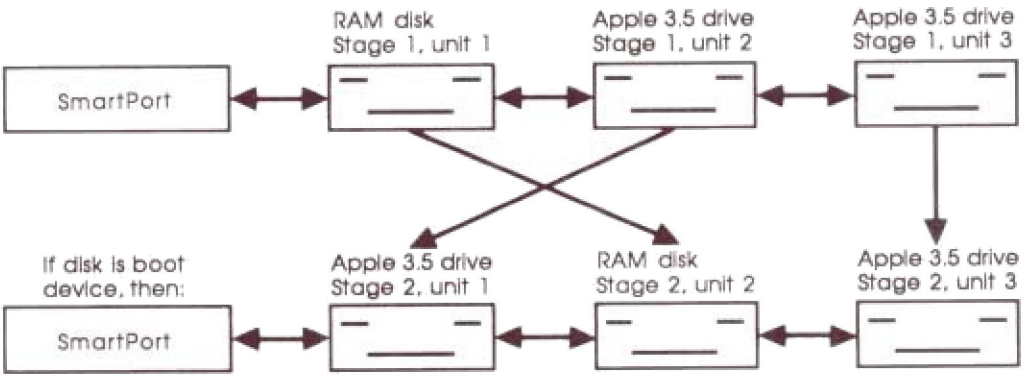


Figure 7-2
Device mapping: configuration 1, derivation 1

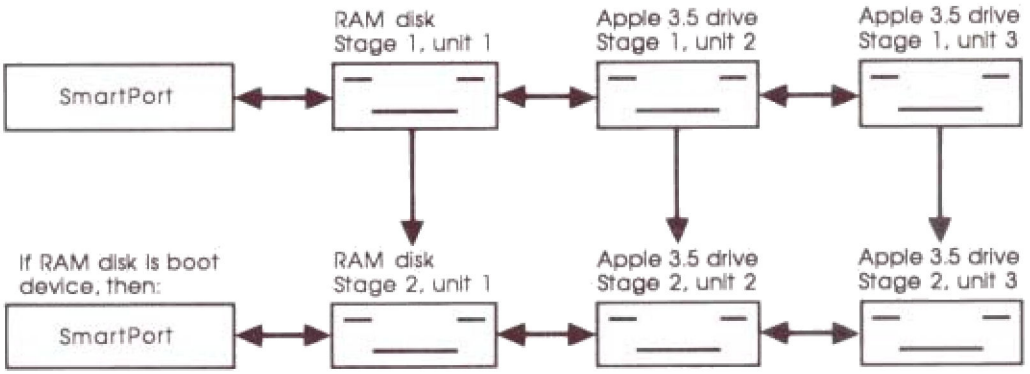


Figure 7-3
Device mapping: configuration 1, derivation 2

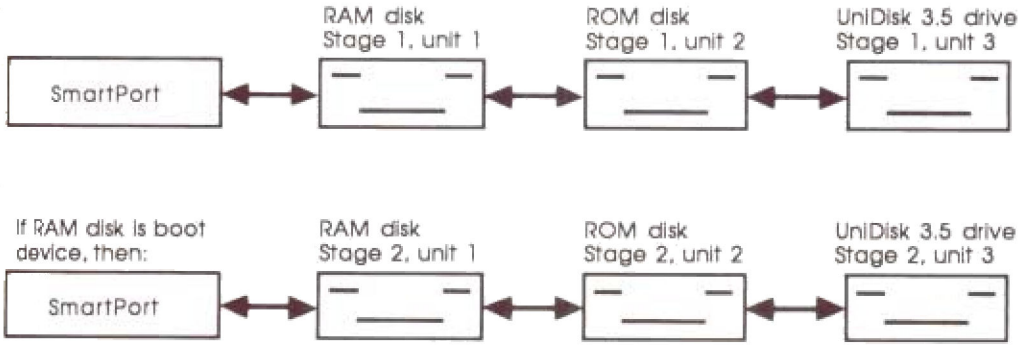


Figure 7-4
Device mapping: configuration 2, derivation 1

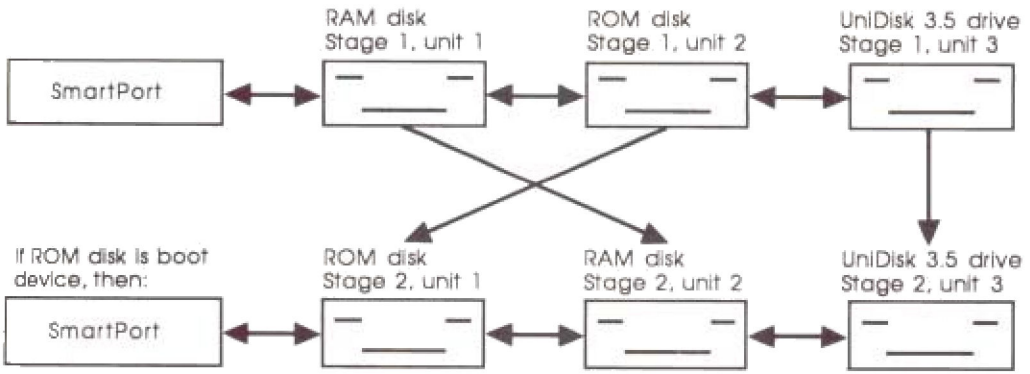


Figure 7-5
Device mapping: configuration 2, derivation 2

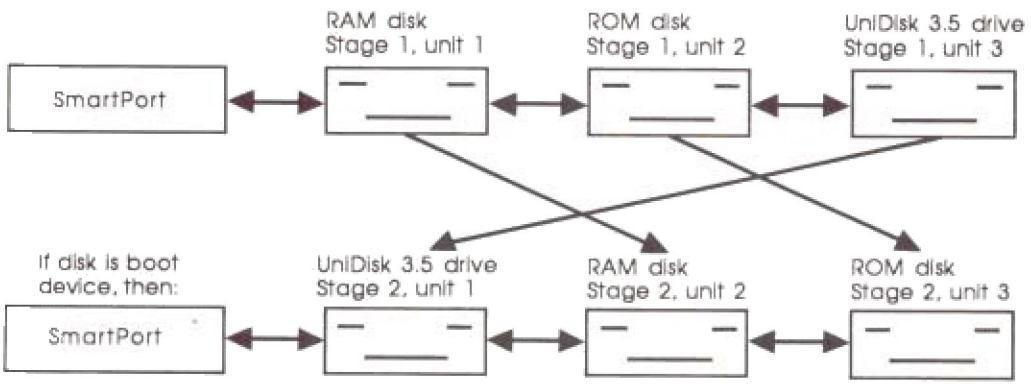


Figure 7-6
Device mapping: configuration 2, derivation 3

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.

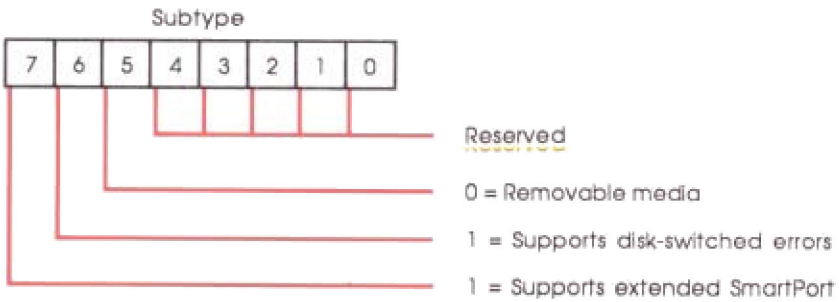


Figure 7-7
SmartPort device subtype byte

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.

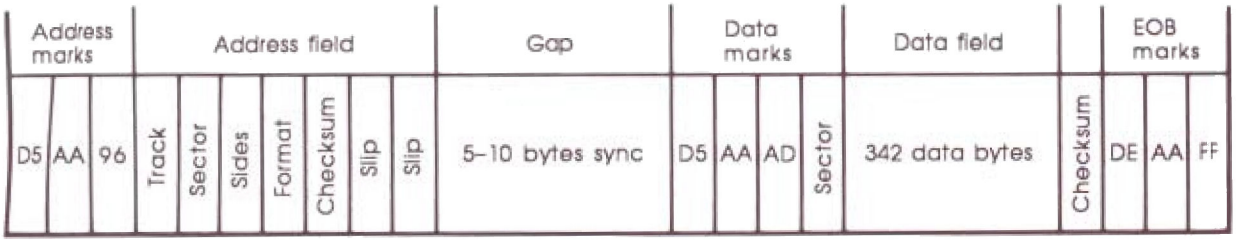


Figure 7-8
Disk-sector format

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).

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.

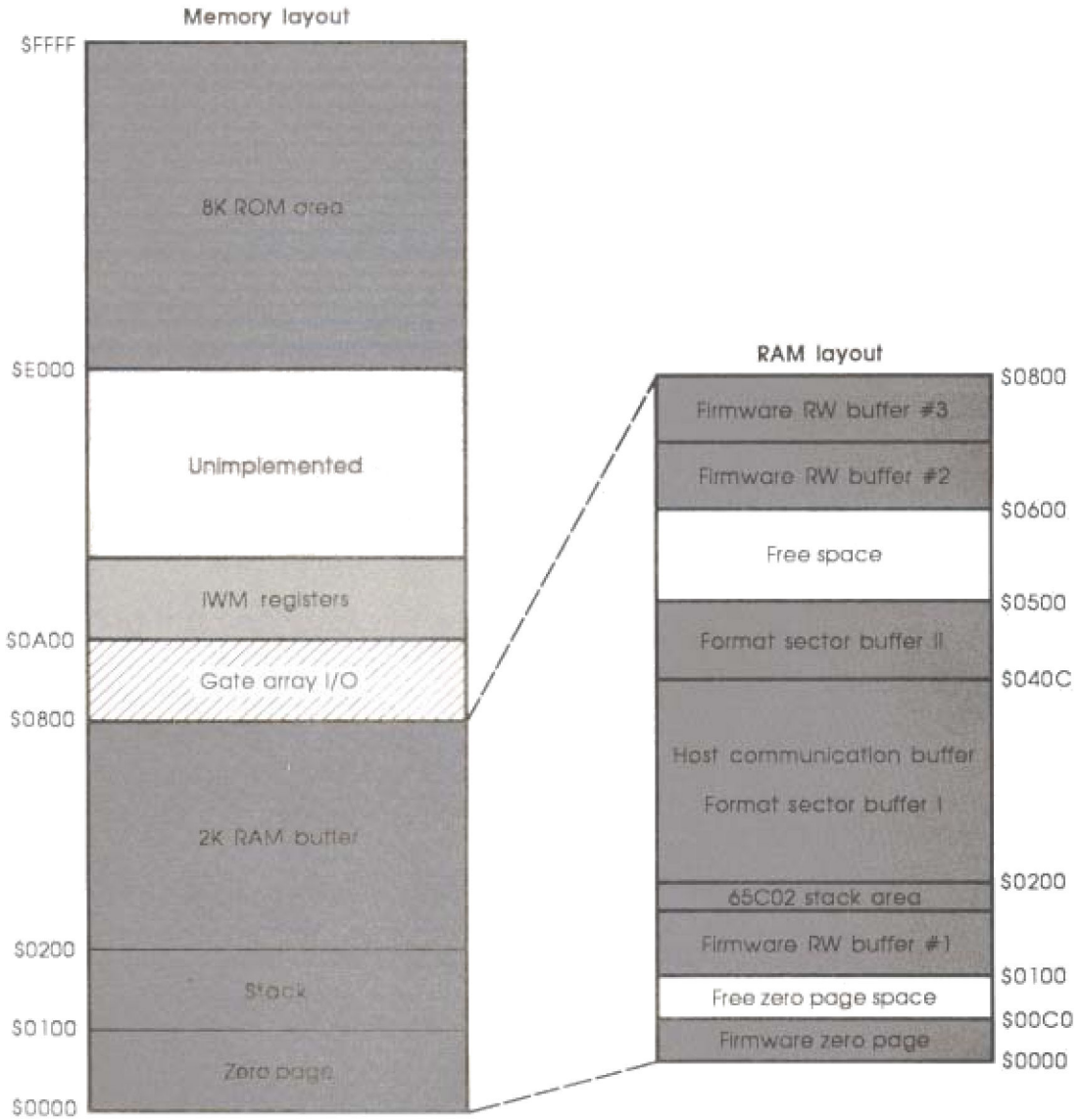


Figure 7-9
UniDisk 3.5 memory map

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	CA0
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.

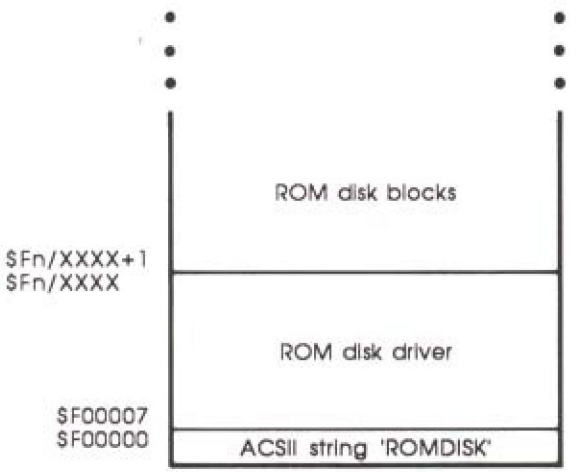


Figure 7-10
The ROM disk

\ block diagram of a ROM disk that occupies 128K of ROM (including the driver itself) ; shown in Figure 7–11. Note that no ROM space has been reserved for toolset xpansion in this example.

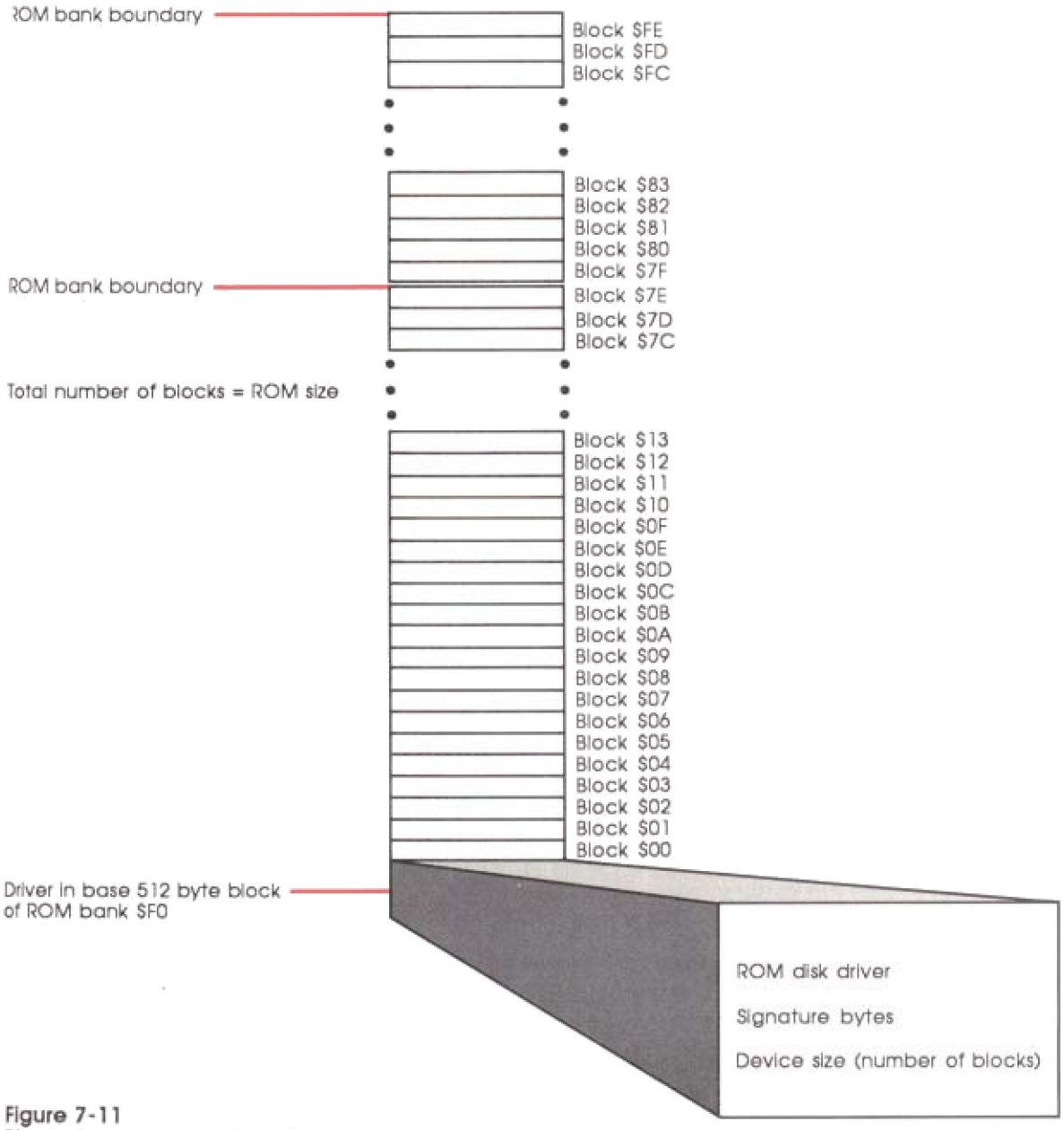


Figure 7-11
Block diagram of a 128K ROM disk

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.

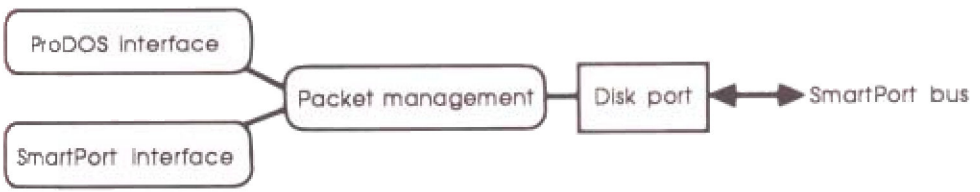


Figure 7-12
SmartPort control flow

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.

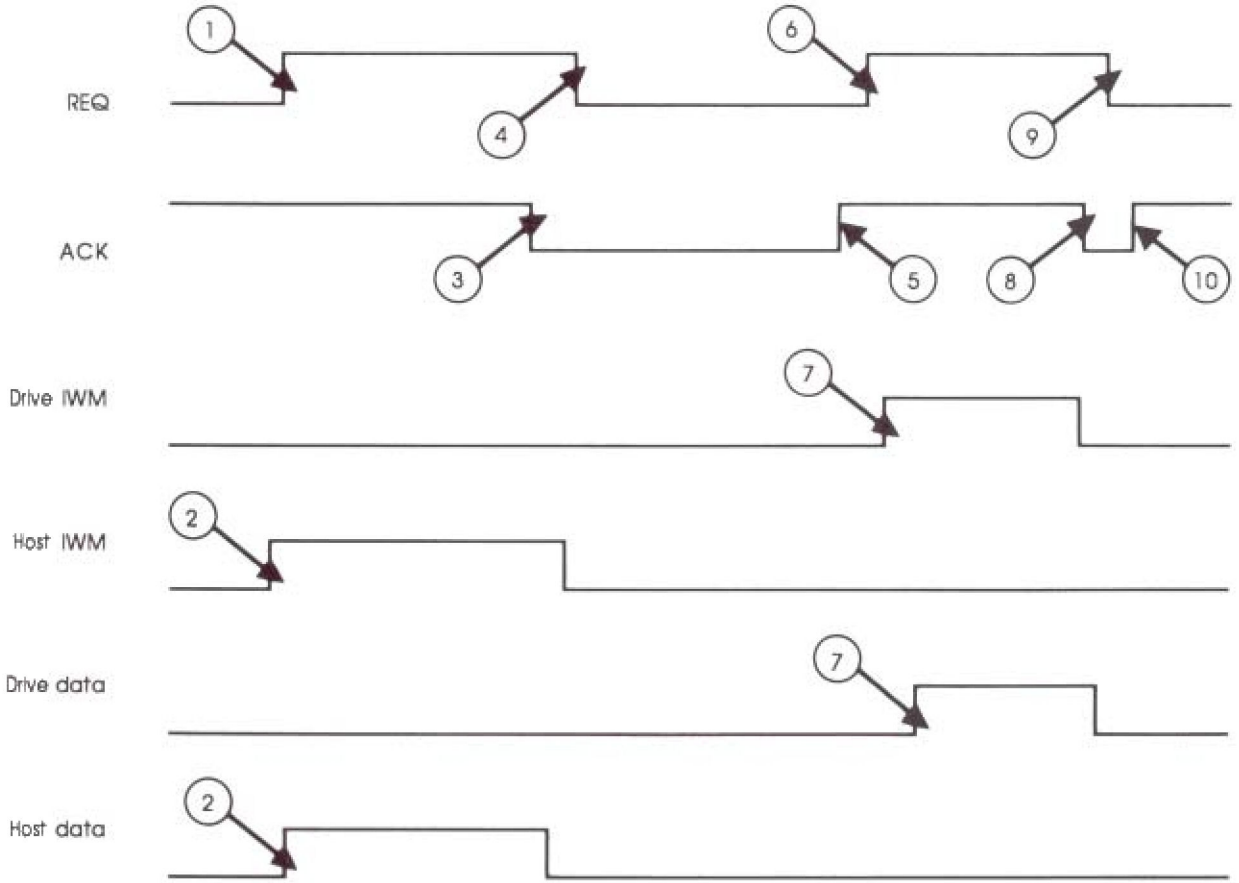


Figure 7-13
SmartPort bus communications: read protocol

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.

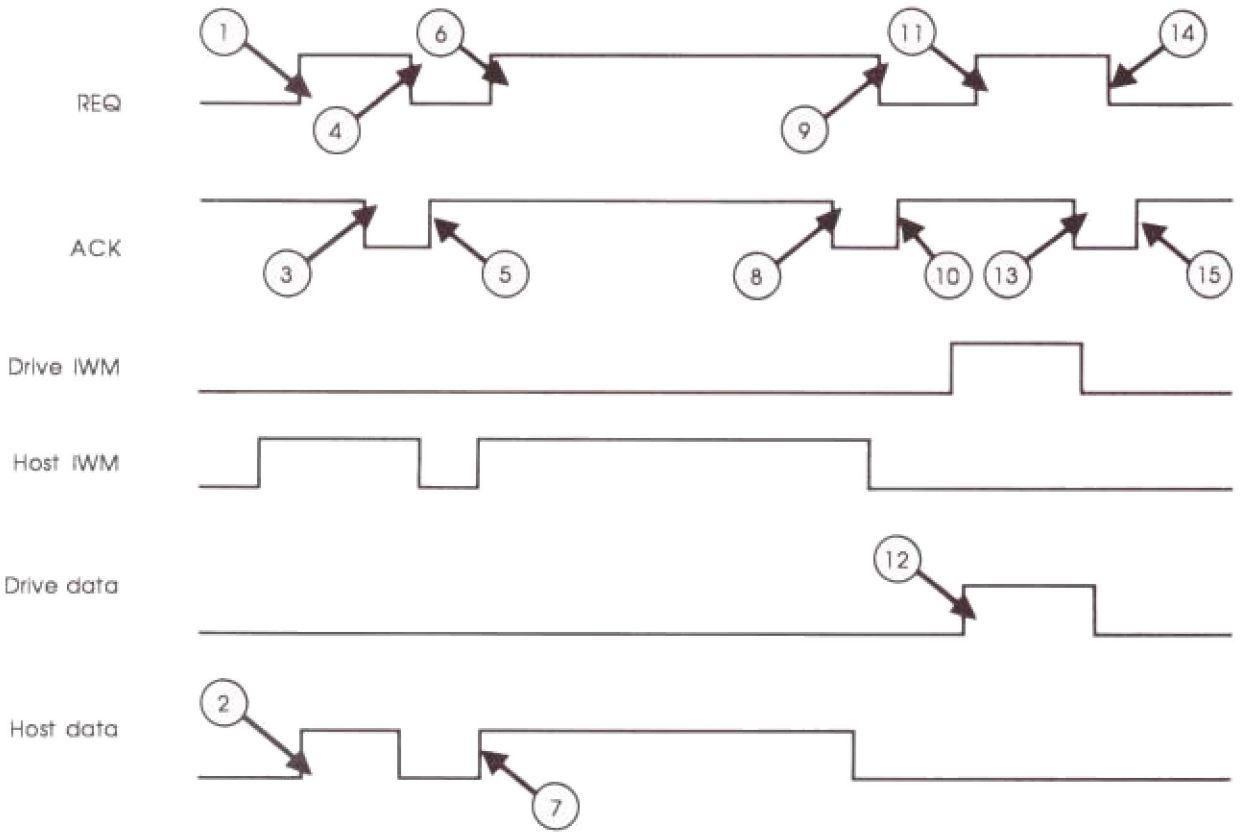


Figure 7-14
SmartPort bus communications: write protocol

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.

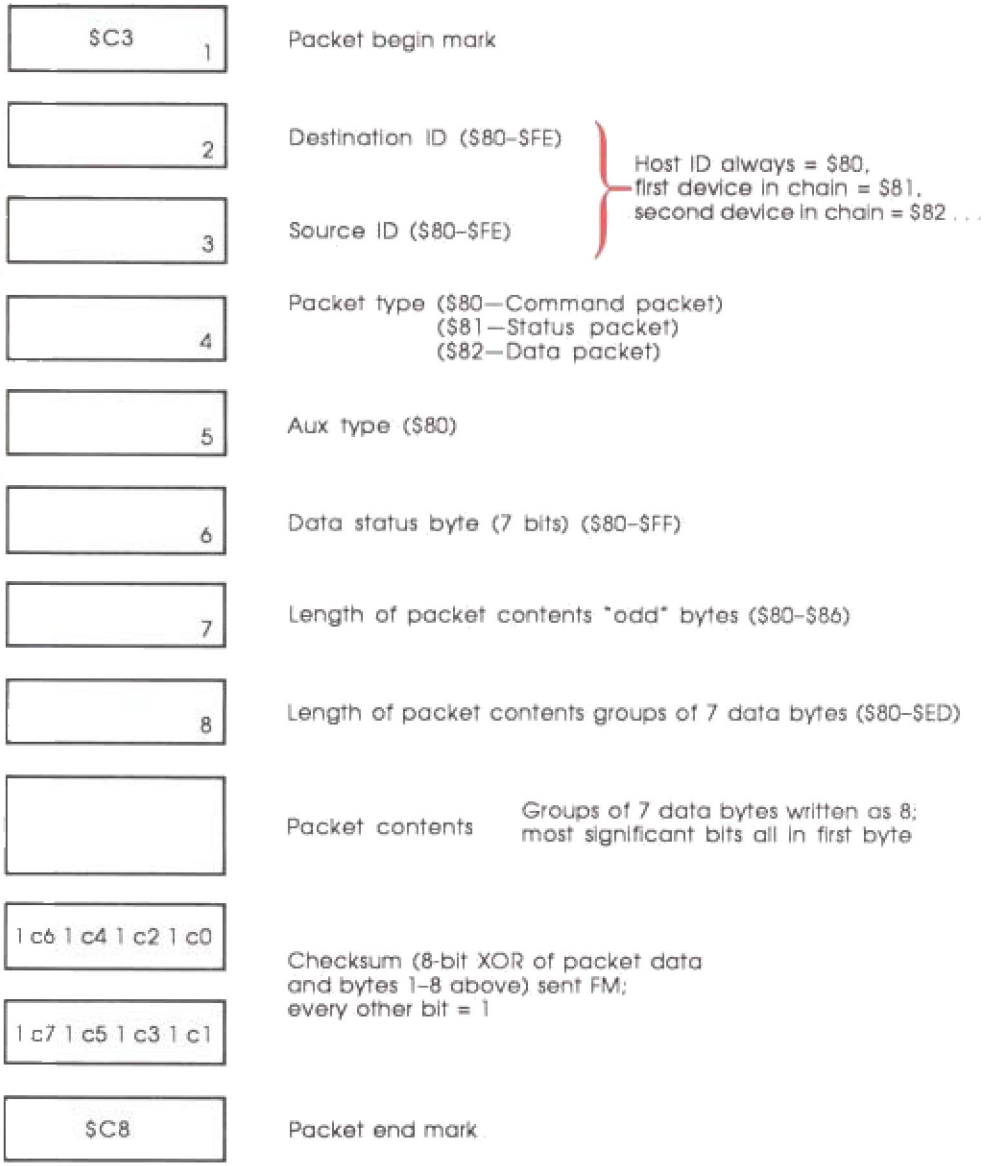


Figure 7-15
SmartPort bus packet format

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.

Packet sizes range from 0 to 767 data bytes.

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.

Chapter 8

Interrupt-Handler Firmware

This chapter describes how the Apple IIGS handles interrupts from the available interrupt sources. You can find additional information about interrupts in Appendix D, "Vectors." This chapter describes interrupts in general and the Apple IIGS built-in interrupt-handler firmware in particular and how to manage environment variables during interrupt handling. It also summarizes all interrupt sources, discussing how often each source interrupts the system and the relative priority assigned by the system to each source, and provides some details about Break instructions, the AppleMouse™, and serial-port interrupt handling.

As a user's program runs, it may get interrupted by various sources to process important external inputs. The system assigns priorities to each of these interrupt sources and handles them in a defined sequence. When the user's program is interrupted, the state of the system at the time of the interrupt is saved. On completion of interrupt processing, the program can continue as though nothing had happened.

There are many reasons for the system to interrupt the execution of a program. For example, if the user moves the mouse, the system should read the mouse location to keep the pointer location current. If the system handles the interrupt promptly, the mouse pointer's movement on the screen will be smooth instead of jerky and uneven. Or your program may be performing another operation while characters are being received in a serial input buffer, and you do not want to lose any characters from the input stream. These conditions, and many others, can cause your program to be interrupted to handle an error or some other special condition that requires immediate attention.

The Apple IIGS interrupt-handler firmware supports interrupts in any memory configuration. To do this, the system saves 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 restores the original state when your program's interrupt handler is finished.

If you write your own interrupt-processing routines, you can attach them to the system by modifying the interrupt vector locations specified in Appendix D, "Vectors." However, you must obey all of the conventions specified in this chapter regarding interrupt processing and make sure to restore the environment to the state in which you found it on entry to your interrupt-processing routine. This will allow the system to restore the environment to its original state.

What is an interrupt?

An interrupt is most often caused by an external signal that tells the computer to stop what it is currently doing and devote its attention to a more important task. Besides this external hardware-related signal, software interrupts are possible as well.

Hardware interrupt priorities are established through a daisy-chain arrangement using two pins, INT IN and INT OUT, on each peripheral-card slot. Each peripheral card breaks the chain when it issues an interrupt request. On peripheral cards that don't use interrupts, the designer of the peripheral card should connect these pins to one another, thereby passing the interrupt signal directly through the card slot.

When the Interrupt Request (IRQ) line on the Apple IIGS microprocessor is activated or when a software interrupt occurs, the microprocessor transfers control to the interrupt-processing routines by jumping through vectors stored in ROM. The built-in interrupt handler processes the interrupt if the application has not provided its own interrupt handler.

The built-in interrupt handler

The Apple IIGS built-in interrupt handler performs a sequence of steps to handle system interrupts. Figure 8-1 shows the structure of the built-in interrupt handler.

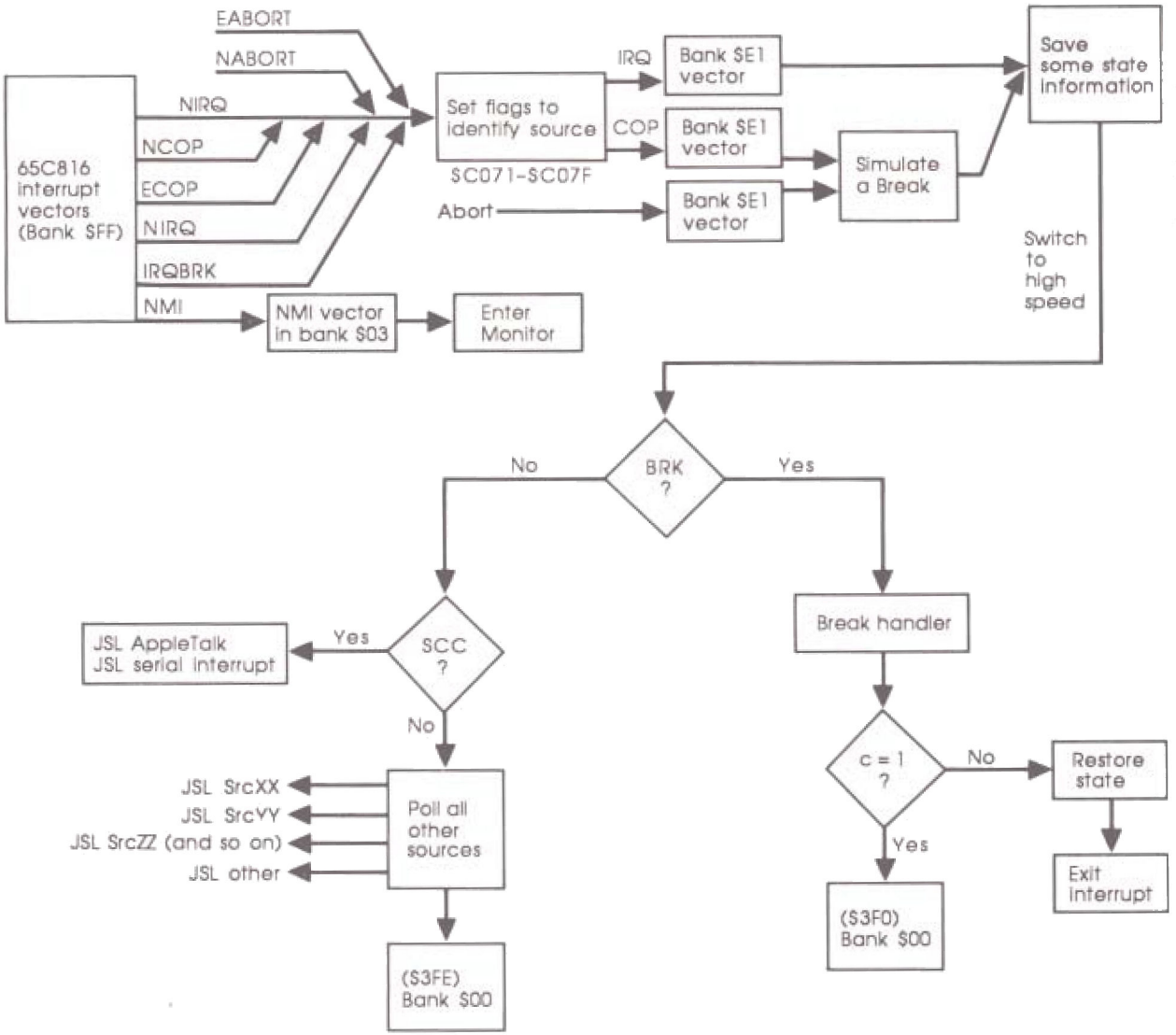


Figure 8-1
Built-in interrupt handler

If I/O shadowing is on, then the system ROM in bank $FF is shadowed (and readable) in bank $00. The system jumps indirectly through the interrupt vector located either at EIRQ ($FFFE, $FFFF) if it was running in emulation mode when the interrupt occurred or at NIRQ ($FFE4, $FFE5) if it was running in native mode.

Important

If I/O shadowing is off, RAM will be addressed in the memory space of bank $00 in the area of $FFE0–$FFFF, the location at which the interrupt vectors are stored. When an interrupt occurs, the 65C816 uses the interrupt vector located in the RAM vector table if I/O shadowing is off and uses the vector located in the ROM vector table if I/O shadowing is on. If you have not correctly set up the RAM vectors and you turn off I/O shadowing, the system will fail.

Both EIRQ and NIRQ jump to ROM located within the soft-switch area at $C071–$C07F. This special ROM code sets status flags that identify the type of interrupt that has just occurred.

At this point, the system tests to see whether the interrupt was a result of a software Break instruction. If it was, the system vectors to the break handler (normally the system Monitor) through the user interrupt-handler vector in bank $E1. An application will patch this vector only if it wants to be responsible for handling or to be informed about all interrupts that occur. If the application simply wants information, it must save the vector value that the application finds in this location and then jump through this vector as the user-interrupt code is completed. Saving and using the vector allows the system to proceed as though the application had never gotten in the way in the first place. If the application wants to handle all interrupt processing on its own, it must be responsible for restoring any environment variables that it changes and must execute an RTI instruction directly from its own code, returning to the application that was interrupted.

If the interrupt source was not a Break instruction, the interrupt handler saves the absolute minimum amount of information about the machine state. The interrupt source might have been AppleTalk (tested first) or the serial port (tested next). If you are running at high baud rates and if interrupt processing takes too long, you might begin to miss characters. To save the minimum machine state, save only the environment variables that have to be used in the routine that saves an incoming serial character in a buffer and points the buffer pointer to its next location. To see whether the interrupt was from a serial port, the SCC is tested. If it is a serial interrupt, the firmware performs a JSL instruction through a patch address in bank $E1 to the port handler (see Appendix D, "Vectors," for more information).

If the port handler returns with the carry bit set, the system does not have an internal serial-port handler installed. The interrupt handler now proceeds to save the rest of the machine state and establish a specific interrupt memory configuration as described in the section "Saving the Current Environment" later in this chapter. (You must poll each of the possible interrupt sources to determine which requires service.)

At this point, the interrupt system begins a polling loop, testing each of the possible interrupt sources in turn. If no internal interrupt handler is installed, then (and only then) the firmware jumps through the user interrupt vector routine to handle the interrupt. The address of the user interrupt routine is found in bank $00, addresses $3FE (low byte) and $3FF (high byte).

The $3FE interrupt handler (user interrupt vector routine) must do the following:

- verify that the interrupt came from the expected source
- handle the interrupt appropriately
- clear the appropriate interrupt soft switch
- restore everything to the state it was in when the Interrupt Request routine was entered, if your routine has made any changes to the state of the machine
- return to the built-in interrupt handler by executing an RTI instruction

After the user interrupt vector routine completes its action, the built-in interrupt handler restores the memory configuration and then executes another RTI to return to where it was when the interrupt occurred.

Here are some factors to remember when you are dealing with programs that run in an interrupt environment:

- There is no guaranteed maximum response time for interrupts because the system may be performing a disk operation that lasts for several seconds when the interrupt occurs.
- Interrupt overhead will be greater if your interrupt handler is installed through an operating system's interrupt dispatcher. The length of delay depends on the operating system and on whether the operating system dispatches the interrupt to other routines before calling yours.

Summary of system interrupts

Table 8-1 lists the source and type of each interrupt and describes each one.

Table 8-1
Summary of system interrupts

Interrupt source	Type	Description
Power up	RESET	Generated by powering up Apple IIGS.
Reset key	RESET	Generated by the ADB microcontroller when Control-Reset is pressed.
External card	RESET	Available.
External card	NMI	Used only for debugging.
Abort signal	ABORT	Activated by memory card slot.
COP instruction	COP/native	In native mode, the user executed a COP instruction.
COP	COP/emulation	In emulation mode, the user executed a COP instruction.
Break instruction	BRK/native	In native mode, the user executed a Break (BRK) instruction.
Break	BRK/emulation	In emulation mode, the user executed a Break (BRK) instruction.
AppleTalk	IRQ	Interrupts upon address recognition or an error.
Serial input #1 (SCC channel A)	IRQ	Interrupts when transmitter is empty, transmission is received, or an error occurs.
Serial input #2 (SCC channel B)	IRQ	Same as serial input #1.
Scan line	IRQ	Interrupts at end of requested scan lines.

(continued)

Table 8-1
Summary of system interrupts (continued)

Interrupt source	Type	Description
Ensoniq chip	IRQ	Interrupts when an oscillator completes a waveform table (32 possible interrupts from here).
VBL signal	IRQ	Interrupts when vertical blanking (VBL) is requested.
Mouse	IRQ	Interrupts as requested at mouse button press or movement or at a VBL signal.
Quarter-second timer	IRQ	Interrupts system every 0.26667 second for AppleTalk use.
Keyboard	IRQ	Interrupts upon keypress.
Response	IRQ	Generated when data is ready for the system from the Apple DeskTop Bus (ADB) microcontroller; initiated as a result of a system-generated command.
SRQ	IRQ	Generated when an ADB device requires servicing.
Desk Manager	IRQ	Generated by the ADB microcontroller when Control-⌘-Esc is pressed.
Flush command	IRQ	⌘-Control Delete was pressed.
Micro abort	IRQ	Generated if the ADB microcontroller detects a fatal error within itself.
Clock chip	IRQ	A 1-second timer interrupt is generated by the 1-hertz signal from the clock chip through the VGC chip.
External card	IRQ	The card wants the attention of the 65C816.
EXTINT	IRQ	Available from the VGC, but not to hook an external interrupting device; hardware is not available.

Interrupt vectors

Table 8-1 described the sources of interrupt and named the interrupt vector that contains the address of the routine that processes each interrupt. Table 8-2 defines the locations at which each of the named interrupt vectors resides.

Table 8-2
Interrupt vectors

Address	Name	Description
$FFFE–$FFFF	IRQVECT	Emulation-mode IRQ/BRK vector
$FFFC–$FFFD	RESET	Emulation- or native-mode RESET vector
$FFFA–$FFFB	NMI	Emulation-mode NMI vector
$FFF8–$FFF9	EABORT	Emulation-mode ABORT vector
$FFF4–$FFF5	ECOP	Emulation-mode COP vector
$FFEE–$FFEF	NIRQ	Native-mode IRQ vector
$FFEA–$FFEB	NNMI	Native-mode NMI vector
$FFE8–$FFE9	NABORT	Native-mode ABORT vector
$FFE6–$FFE7	NBREAK	Native-mode BRK vector
$FFE4–$FFE5	NCOP	Native-mode COP vector

If I/O shadowing is on, the vectors contained in ROM are always used by the 65C816, regardless of the language-card settings. This allows you to run native-mode code with interrupts enabled in old applications.

If the application program or operating system disables I/O shadowing in bank $00 or $01, then either the application program or the operating system must copy the ROM vectors from $FFEE to $FFFF and the code from $C071 to $C07F into RAM at the same locations before enabling interrupts. If the code is not copied from ROM to RAM, the Monitor's interrupt code cannot be used.

Interrupt priorities

The 65C816 processes each type of interrupt on a priority basis. For instance, if several of the many IRQ interrupts should occur at the same time, the 65C816 will process all AppleTalk IRQs before any keyboard interrupts. Priorities for each type of interrupt are indicated by their relative position in the following paragraphs. In other words, the highest-priority interrupts appear closest to the beginning of these descriptions. Lower-priority interrupts appear later in the descriptions.

RESET

RESET forces emulation mode. The interrupt is processed by the firmware and then vectors to the user link. A cold start attempts to boot a disk. A cold start can be performed in two ways:

- by turning the power off and on
- by pressing ⌘-Control-Reset

RESET cold-start functions are as follows:

- sets up video
- sets video as output device
- sets keyboard as input device
- reads clock chip and places system configuration in firmware RAM
- sets up system to match configuration in firmware RAM
- sets up the power-up byte so the next RESET performs a warm start
- scans slots for Disk II devices and sets motor-on detect bit (motor-on detect causes the FPI chip to slow the system down to 1 MHz when the motor-on soft switch is enabled, and it restores the system speed when the motor is turned off)
- goes to, or scans, for boot device (if boot device is found, jumps to it; if no boot device is detected, switches in Applesoft BASIC and jumps to it)

A warm start vectors to user links. If user did not alter links, then a BASIC cold start is executed. A warm start can be performed in two ways:

- by pressing Control-Reset
- by using peripheral cards (pulling RESET line low)

The system executes the following reset warm-start functions:

- sets up video
- sets video as output device
- sets keyboard as input device
- reads image of system configuration in firmware RAM
- sets up system to match configuration
- generates tone (beep replaced with tone)
- jumps to user reset vector

NMI

NMI vectors to user link. No NMI interrupts are used by the Monitor. Peripheral cards pull NMI line low.

ABORT

ABORT vectors to the user link. If no user link exists, it vectors to the break handler that displays the address and opcode of the code being executed at the time the abort pin on the 65C816 was pulled low (see BRK). The ABORT interrupt can be activated only by hardware installed in the memory-expansion slot.

COP

COP vectors to the COP (coprocessor opcode) manager vector in RAM, which points to the firmware. If the COP manager is not installed, the firmware displays the COP message via a software COP instruction.

In emulation mode, COP prints the following:

```
bb/addr:  00  cc      COP  cc
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=1
```

In native mode, COP prints the following:

```
bb/addr:  00  cc      COP  cc
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=0
```

❖ *Note:* The preceding formats are for a 40-column screen. On an 80-column screen, the second two lines become one line. The `cc` appearing in both modes is the operand of the COP instruction and indicates to the user where the COP occurred ($00 through $FF are valid COP operands).

BRK

In emulation mode, the interrupt vectors to the interrupt (IRQ) handler and then to the break handler. In native mode, the interrupt vectors directly to a break handler. This occurs via a software BRK instruction only. The break handler saves as much data as the interrupt handler. This allows you to invoke the Monitor Resume command (R) to continue program execution.

In emulation mode, the Break instruction prints the following:

```
bb/addr:  00  bc      BRK  cc
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=1
```

In native mode, the Break instruction prints the following:

```
bb/addr:  00  bc      BRK  cc
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=0
```

❖ *Note:* The preceding formats are for a 40-column screen. On an 80-column screen, the second two lines become one line. The `cc` appearing in both modes is the operand of the BRK instruction and indicates to the user where the BRK occurred ($00 through $FF are valid BRK operands).

IRQ

IRQ interrupts are as follows:

AppleTalk: This interrupt has the highest priority because its code is very time intensive; data can be lost if the SCC is not read within 104.167 microseconds (baud = 230,400) after an AppleTalk SCC interrupt occurs.

Serial ports: In interrupt mode, data will be lost if the SCC is not read within 1.094 milliseconds (baud = 19,200) after the interrupt occurs.

Scan line: The scan-line interrupt can occur every 63.694 microseconds. When the video counters count down to zero, the interrupt occurs. The video counters reach zero when the scanning beam reaches the right side of the scan line.

Ensoniq chip: The Ensoniq chip interrupts when the waveform buffer is completed. Because the chip contains 32 oscillators, there are 32 possible interrupts from the chip.

VBL: The VBL interrupts every 16.6667 milliseconds. The interrupt occurs when the scanning beam is retracing from the bottom-right corner to the upper-left corner of the screen. (*Note:* Using the heartbeat interrupt handler is the approved method of executing VBL interrupt tasks.)

Mouse: The mouse interrupts only if the interrupt option is specified. The interrupt options are mouse movement, mouse button press, and VBL signal.

Quarter-second timer: This timer interrupts every 0.26667 second. The timer is used by AppleTalk to trigger event processing.

Keyboard: The keyboard interrupts if a key is pressed.

Response: If a command is sent to the ADB microcontroller, the interrupt occurs when the "done" flag is set. The microcontroller interrupts the system when the response data is ready for the system to read. If this interrupt occurs, control is passed to the response manager.

SRQ: If an ADB device requires servicing, an SRQ (service request) is issued. This event can interrupt the system. When this interrupt occurs, control is passed to the SRQ manager.

Desk Manager: The ADB microcontroller causes this interrupt if Control-⏏-Esc is pressed. Control is then passed to the Desk Manager.

Flush: If ⏏-Control-Backspace (Delete) is pressed, the ADB microcontroller clears its internal type-ahead buffer, issues a Flush command to external keyboards, and causes an interrupt. If this interrupt occurs, control is passed to the Scrap Manager.

Micro abort: If the ADB microcontroller detects a fatal error and the fatal-error interrupt occurs, the system is interrupted. If this interrupt occurs, control is passed to the ADB Tool Set.

Clock chip: The clock chip interrupts once each second.

External cards: External cards cause interrupts as defined by the card manufacturer.

Environment handling for interrupt processing

For each interrupt discussed in the previous section, the processor can be in either emulation or native mode. Each mode has its own interrupt vector; therefore, there are two different entry points to the interrupt handler. To process interrupts correctly, the system interrupt handler must save the current environment, set the interrupt environment, and process the interrupt through the appropriate interrupt handler. (You can find more information about saving and restoring the environment in Chapter 2, "Notes for Programmers." That chapter contains sample assembly-language code that saves a part of your environment and sets the system into the correct mode for interrupt processing.)

Saving the current environment

On entry to each interrupt, the system interrupt handler saves the current environment and sets the program bank, data bank, and direct-page register contents to zero.

The state of the machine upon entry into each interrupt handler is indicated by the contents of the following registers:

- program bank
- data bank
- direct register
- processor status
- A register
- X register
- Y register

The RAM or ROM state, including emulation or native mode, is indicated by the following:

- language-card state (bank 1 or 2, ROM or RAM)
- main or alternate memory (and main and alternate zero page)
- 80STORE switch
- 80STORE switch
- 40- or 80-column video
- main stack or zero page in use
- speed register
- Shadow register

Going to the interrupt environment

If the interrupt can be processed by the firmware or a tool set, the processor vectors to the appropriate handler in native mode, 8-bit m/x, in high speed. If the interrupt cannot be processed by the firmware, the processor performs the following steps:

1. Switches to emulation mode
2. Switches speed to 1 MHz
3. Switches in text page 1 to make main screen holes available
4. Switches in main memory for reading and writing
5. Maps $D000–$FFFF ROM into bank $00
6. Switches in main stack and zero page
7. Saves the auxiliary stack pointer and restores the main stack pointer

After the environment is saved and the new environment is set, the interrupt handler checks for the source of the interrupt. If the interrrupt is a firmware interrupt only (a BRK or COP instruction), the firmware jumps (using a JSL) to the appropriate firmware routine. If it is an interrupt that is passed directly to the user, then the firmware passes the interrupt to the user via the appropriate links. An interrupt can be both processed by the firmware and passed to the user. If both occur, the preceding rules listed still hold, except that the particular firmware interrupt handler will return to the main interrupt handler with carry set (C = 1) instead of clear (C = 0), which indicates that the firmware processed the interrupt and the user does not need to know about it.

Restoring the original environment

After the interrupt has been processed, the system interrupt handler restores the environment and registers to their preinterrupt state and performs an RTI, returning to the executing program.

❖ *Note:* The peripheral card (or equivalent internal card) in use is responsible for saving its slot number in the form $Cn (n = slot number) at MSLOT ($0007F8). MSLOT is used in the interrupt handler to restore the currently executing slot number's $C800 space after an interrupt has been processed.

Emulation-mode interrupts are supported in bank $00 only. Native-mode interrupts are supported everywhere in memory. Therefore, code running anywhere except in bank $00 must be native-mode code.

Handling Break instructions

In emulation mode, the Apple IIGS detects a software Break (BRK) instruction as an IRQ and jumps through the emulation-mode IRQ vector. In that code, the firmware determines that a Break instruction was issued and so jumps through the emulation-mode BRK vector. In native mode, the 65C816 can tell the difference between BRK and IRQ, so it jumps directly through the native-mode BRK vector.

Apple IIGS mouse interrupts

The Apple DeskTop Bus (ADB) microcontroller periodically polls the ADB mouse to check for activity. If the mouse has moved or the mouse button has been pushed, the mouse firmware will respond to the microcontroller by returning 2 bytes of data. The microcontroller returns this data to the system by writing both mouse data bytes to the GLU chip (mouse byte Y followed by byte X—this enables the interrupt). Data bytes are read *only* if the Event Manager (if active) or the application program issues the mouse firmware call or the tool call ReadMouse. The GLU chip is the general logic unit that provides logic elements enabling the 65C816 to communicate with the ADB microcontroller.

The Apple IIGS mouse firmware causes interrupts for the 65C816 microprocessor only if the interrupt mode has been selected via firmware. The Apple IIGS mouse interrupts in synchronization with the Apple IIGS vertical blanking signal (VBL). The mouse can interrupt the 65C816 a maximum of 60 times per second. This cuts down on the burden the mouse puts on the 65C816.

At power-up or reset, the GLU chip turns the mouse interrupt off and enters the mouse into a noninterrupt state.

Serial-port 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, the application 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 interrupt vector). The user's interrupt service handler is then completely responsible for all serial-port interrupt service. You can find further details about the serial-port firmware and its commands in Chapter 5, "Serial-Port Firmware."

If the application does not want to disable buffering, but does wish to be *notified* that a certain type of serial-port interrupt has occurred, the application 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. 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 this 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. When the firmware receives a break character, the firmware SCC interrupt handler then records and clears the interrupt and finally passes 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 the 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 at their entry to your routine:

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

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

Chapter 9

Apple DeskTop Bus Microcontroller

This chapter describes the Apple DeskTop Bus (ADB) microcontroller. This hardware device collects information from the ADB peripheral devices. In association with the ADB Tool Set, the data that is collected is available to the user. Typical data includes key-down and key-up sequences, mouse moves, and button clicks. The firmware that performs these operations is not documented here. See the ADB Tool Set documentation for information about the ADB firmware. This chapter is for reference only, providing a developer's view of the complete ADB system.

The ADB device is an I/O port with its own microcontroller. The microcontroller accepts commands from the 65C816, manages the internal keyboard, and acts as a host processor for ADB peripheral devices such as the mouse, the detachable keyboard, and other devices that follow the ADB protocol.

The ADB system has four components and three distinct software interfaces. Figure 9-1 shows the ADB system from a hardware perspective.

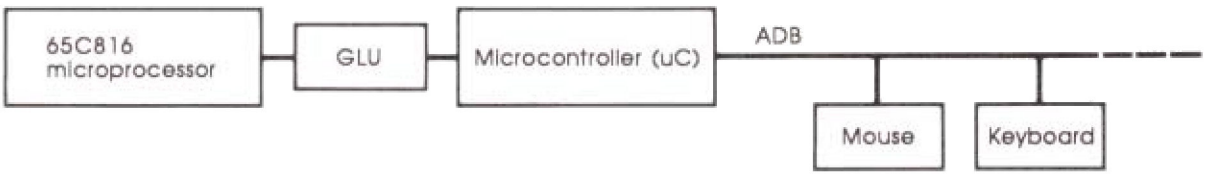


Figure 9-1
Apple DeskTop Bus components

The four hardware components are the 65C816, the **GLU** (general logic unit) chip, the ADB microcontroller, and the components attached to the Apple DeskTop Bus device. The application accesses the ADB components through the ADB Tool Set. The ADB Tool Set talks to the hardware by sending commands through the GLU chip to the microcontroller. Some of these commands require data transfer over the ADB, and others terminate in the microcontroller.

The GLU chip is actually a set of hardware registers (sometimes called *mailbox* registers) that the 65C816 uses to transmit commands and data to the microcontroller from the 65C816 and that the microcontroller uses to pass data to the 65C816. Both the 65C816 and the microcontroller are independent processors, each running at its own pace. They exchange data through the GLU chip.

The microcontroller translates the commands it receives into data streams that it sends along the Apple DeskTop Bus device itself. All peripheral devices attached to the bus listen to the data stream being transmitted. If the command is intended for a specific peripheral device, it responds and possibly transmits data and status information back to the microcontroller. The microcontroller, in turn, translates the data and sends the translated data to the 65C816.

There is actually one more software interface: the program running independently in the microcontroller itself. But that is immaterial here. It is sufficient to note that this is an intelligent peripheral device that manages communication.

The *Apple IIGS Hardware Reference* provides details about the hardware interface between the ADB microcontroller and its attached peripheral devices and how the microcontroller manages the internal and external keyboard and the mouse, the reset sequence and the ⏻ key, key buffering (type-ahead), and so on.

The *Apple IIGS Toolbox Reference* provides details about the high-level commands that allow access to the items attached to the ADB.

Although most applications do not require the information in this chapter, there are a few exceptions:

- applications that allow the user to temporarily change Control Panel options
- alternative input devices such as a graphics tablet (however, an application may not need to worry about this because a device driver can be transparently hooked into the Event Manager)
- multiplayer or multidevice applications

If an application needs to temporarily change some Control Panel options, use the ADB Tool Set. Note, however, that changing certain options can cause the system to fail.

An application should not call the ADB Tool Set to change Control Panel options permanently. If a permanent change in certain system characteristics, such as the auto-repeat rate or buffer-mode options, is necessary, the application should make the changes by changing the Battery RAM (using the Miscellaneous Tool Set). Then the application should call the routine TOBRAMSETUP to update the system with the new Battery RAM values.

If you are writing a user program that uses the mouse and the keyboard, you will probably not need the information in the rest of this chapter. For that level of information, see the *Apple IIGS Toolbox Reference*. If you are a hardware developer developing a new peripheral device for the Apple DeskTop Bus, you will need the information given here as well as the information about the bus protocol itself and interface specifications for ADB devices. This latter information is in the *Apple IIGS Hardware Reference*.

The discussion in this chapter focuses on the ADB microcontroller and its commands. The rest of this chapter is for reference only; it shows the application designer the kinds of commands the ADB Tool Set issues to the microcontroller. You should ***not*** attempt to send any of these command streams to the microcontroller yourself.

Important

Microcontroller communication is exclusively the job of the Apple IIGS Tool Set.

ADB microcontroller commands

The microcontroller uses two types of commands: default and mode commands and ADB commands. The default and mode commands are used by the Control Panel to change system settings. The ADB commands are used to communicate with ADB devices other than the detachable keyboard and the mouse (these are handled automatically).

Caution

An application program must issue microcontroller commands only through the ADB Tool Set. If you attempt to use these commands directly, bypassing the tool set, you could cause a system failure. (For more information about the tool set, see the *Apple IIGS Toolbox Reference*.)

This section provides a detailed description of each ADB microcontroller command. The command values are given in binary format where the most significant bit is the leftmost bit. A percent sign (%) preceding a string of zeroes and ones indicates a binary value. The notation xy substituted for a binary digit pair in a command byte stands for 2 bits that select one of four possible registers. The notation abcd substituted for four binary digits in a command byte, stands for 4 bits that select one of 16 possible device addresses. The ADB can support up to 16 different device addresses, each of which may have four hardware registers.

Abort, $01

This command synchronizes the microcontroller with the 65C816 microprocessor when a command error occurs. Abort is a 1-byte command with a value of %00000001.

Reset Keyboard Microcontroller, $02

This command returns the keyboard microcontroller to its power-up state. It is a 1-byte command with a value of %00000010.

Flush Keyboard Buffer, $03

This command clears the keyboard buffer. Any keystrokes that were pending are forgotten. It is a 1-byte command with a value of %00000011.

Set Modes, $04

This command sets modes. It is a 2-byte command; the first byte value is %00000100. For each bit set in the byte that follows the Set Modes command, the corresponding mode bit is set.

Clear Modes, $05

This command clears modes. It is a 2-byte command; the first byte value is %00000101. For each bit set in the byte that follows the Clear Modes command, the corresponding mode bit is cleared.

Table 9-1 lists command bit functions.

Table 9-1
Bit functions

Bit	Function
7	Resets from the ADB detachable keyboard alone when the Reset key alone is pressed (Control not needed); works only with the detachable keyboard.
6	Sets the exclusive-OR/Lock-Shift mode. (With the Caps Lock key down, you type lowercase characters when you press the Shift key.)
5	Reserved.
4	Buffer keyboard mode.
3	Enables 4X repeat instead of dual (2X) repeat. (When the Control key is pressed, the repeat speed for arrows is four times the normal speed.)
2	Includes the Space bar and Delete key on dual repeat. (When the Control key is pressed, the repeat speed for Space bar, Delete key, and arrows is doubled.)
1	Disables ADB mouse autopoll (disables the mouse).
0	Disables ADB keyboard autopoll (disables the keyboard).

Set Configuration Bytes, $06

This command sets configuration bytes. This is a 4-byte command (%00000110) that uses the 3 bytes following the command as follows:

Byte 1

High nibble ADB mouse address
Low nibble ADB keyboard address

Byte 2

High nibble Sets character set (needed for certain languages) most significant bit if keypad "." swapped with ","

Low nibble Sets keyboard layout language (see Table 9-2)

Byte 3

High nibble Sets delay to repeat rate (3 bits)

0 1/4 sec
1 1/2 sec
2 3/4 sec
3 1 sec
4 No repeat

Low nibble Sets auto-repeat rate (3 bits)

0 40 keys/sec
1 30 keys/sec
2 24 keys/sec
3 20 keys/sec
4 15 keys/sec
5 11 keys/sec
6 8 keys/sec
7 4 keys/sec

Table 9-2 lists the keyboard language codes used for byte 2 of the Set Configuration Bytes command.

Table 9-2
Keyboard language codes

Language	Abbreviation	Code	Language	Abbreviation	Code
English (U.S.)	US	0	Italian	IT	5
English (U.K.)	UK	1	German	GR	6
French	FR	2	Swedish	SW	7
Danish	DN	3	Dvorak	DV	8
Spanish	SP	4	Canadian	CN	9

Sync, $07

This command performs three of the preceding commands in sequence. It sets the mode byte (see "Set Modes, $04" and "Clear Modes, $05") followed by the Set Configuration Bytes (see "Set Configuration Bytes, $06"). This command is issued by the system after a reset operation. After receiving the command, the microcontroller resets itself to its internal power-up state and then resets all ADB devices. Sync is a 1-byte command with a value of %00000111.

Write Microcontroller Memory, $08

This command writes a value into the ADB microcontroller RAM. It is a 3-byte command. The first byte has a value of %00001000. The second byte is the address to write into. The third byte is the value to be written.

Read Microcontroller Memory, $09

This command reads a byte from the ADB microcontroller memory. The command reads ROM or RAM locations, depending on the value of the high byte of the address sent for reading. This is a 3-byte command. The value of the first byte is %00001001. The second byte is the low byte of the microcontroller address. The third byte is the high byte of the microcontroller address. If the third byte is 0, RAM is read; otherwise, ROM is read. This command returns 1 byte.

Read Modes Byte, $0A

This command reads the modes byte (see "Set Modes, $04" or "Clear Modes, $05"). It is a 1-byte command with a value of %00001010. It returns 1 byte.

Read Configuration Bytes, $0B

This command reads configuration bytes. It is a 1-byte command with a value of %00001011. This command returns to the 65C816 (through the data latch in the GLU) a total of 3 bytes (presented one at a time for reading by the 65C816) representing the most recently set configuration (from the most recent Set Configuration Bytes command). The 3 bytes are returned in the following sequence:

Byte 1

High nibble	ADB mouse address
Low nibble	ADB keyboard address

Byte 2

High nibble	Sets character set (needed for certain languages)
Low nibble	Sets keyboard layout language (see Table 9-2)

Byte 3

High nibble Sets delay to repeat rate (3 bits)

0	1/4 sec
1	1/2 sec
2	3/4 sec
3	1 sec
4	No repeat

Low nibble Sets auto-repeat rate (3 bits)

1	30 keys/sec
2	24 keys/sec
3	20 keys/sec
4	15 keys/sec
5	11 keys/sec
6	8 keys/sec
7	4 keys/sec

Read and Clear Error Byte, $0C

This command returns the ADB error byte to the data latch in the GLU. It clears the ADB error byte to zero. It is a 1-byte command with a value of %00001100. This command is useful for hardware developers debugging new ADB devices.

Get Version Number, $0D

This command returns the device version number into the data latch in the GLU. It is a 1-byte command with a value of %00001101.

Read Available Character Sets, $0E

This instruction reads available character sets. It is a 1-byte command with a value of %00001110. The first byte value returned specifies how many character-set identifiers follow this first byte. Subsequent bytes returned through the data latch identify the character sets. This command is used by the Control Panel to determine which character sets are available in the system. It is assumed that each microcontroller is paired with a specific megachip. However, when the Apple IIGS is manufactured, the factory may install one type of megachip and a different type of microcontroller. This command allows the system to correctly match the capabilities of the megachip with the microcontroller that is actually installed in the system.

The order in which the character sets are returned is important. The first number returned corresponds to character set 0 in the megachip; the next number corresponds to character set 1.

Read Available Keyboard Layouts, $0F

This command (%00001111) returns the number of keyboard layouts available. This command is used by the Control Panel to determine which keyboard layouts are available in the system. Like the Read Available Character Sets command, the order in which the numbers are returned is important. The first number returned represents layout 0 in the microcontroller.

Reset the System, $10

This command resets the system and pulls the reset line low for 4 milliseconds. It is a 1-byte command with a value of %00010000.

Send ADB Keycode, $11

This command is used to emulate an ADB keyboard by accepting ADB keycodes from a device and then sending them to the microcontroller to be processed as keystrokes. This command does not process either reset-up or reset-down codes; these reset keycodes must be processed separately. This command can be used to detect key-up events or to emulate a keyboard with another device, such as might be used for the handicapped. This is a 2-byte command. The first byte has a value of %00010001; the second byte contains the keystroke to be processed. See the *Apple IIGS Hardware Reference* for details about the values that correspond to specific key-down, key-up sequences.

Reset ADB, $40

This command pulls the ADB low for 4 milliseconds. Care must be taken with this command because resetting an ADB keyboard clears any pending commands, including all key-up events. This means that if this command is issued as a result of a key being pressed, when the key is released, the key-up code will be lost and the key will autorepeat until another key is pressed. All keys should be up before this command is executed. This is a 1-byte command with a value of %01000000.

Receive Bytes, $48

This command is used to receive data from an ADB device. This is a 2-byte command. The first byte value is %01001000. The second byte value is a combination of the ADB command (see the *Apple IIGS Hardware Reference*) in the high nibble and the device address in the low nibble. The microcontroller sends this ADB command byte on the ADB and then waits for the device to return data. The microcontroller then returns the data bytes to the system in the opposite order that they were received from the ADB. (The issuer of this command must know about ADB commands and the values they return.)

Transmit num Bytes, $49–$4F

This command is used to transmit data to an ADB device. This is a 3- to 9-byte command. The first byte value is the Transmit command itself and has a value of %01001num, where num is a set of 3 binary bits that represent a number. The value of (num + 1) specifies how many data bytes will be transmitted as part of this command. The second byte value is an actual ADB command. The third and subsequent bytes (num + 1) are bytes that are transmitted directly to the devices on the ADB bus immediately following the ADB command.

Enable Device SRQ, $50–$5F

This command enables an SRQ (service request) on the ADB device at address abcd. It is a 1-byte command with a value of %0101abcd.

Flush Device Buffer, $60–$6F

This command flushes the ADB device buffer at address abcd. It is a 1-byte command with a value of %0110abcd.

Disable Device SRQ, $70–$7F

This command disables the SRQ on an ADB device at address abcd. It is a 1-byte command with a value of %0111abcd.

Caution

If data is pending when this command is executed, the pending data could be lost. For example, if SRQ is disabled on the ADB keyboard, then all key-up codes could be lost. See "Reset ADB, $40."

Transmit Two Bytes, $80–$BF

This command transfers 2 bytes of data (data and status information) from a specific device using the ADB Listen command (see the *Apple IIGS Hardware Reference*). It is a 1-byte command with a value of %10xyabcd, where xy is the register number and abcd is the device address.

Poll Device, $C0–$FF

This command is used to get data from a specific device. It uses the ADB Talk command. After the Talk command is executed, the microcontroller waits for the device to send back data or for timeout. The microcontroller waits until all data has been received and then returns a status byte (see Table 9-3) to the system indicating the number of bytes received and then returns the data. It returns the bytes in an order opposite that in which they were received by the ADB. This is a 1-byte command with a value of %11xyabcd, where xy is the register number and abcd is the ADB device address.

- *Note:* All commands (except the Sync command) that require more than a 1-byte transfer automatically return timeout in 10 milliseconds if there is no response. The Sync command may require 20 milliseconds to process the ADB address byte.

Microcontroller status byte

The ADB microcontroller sends a status byte to the system when it detects one of the conditions listed in Table 9-3. When the system receives the microcontroller status byte, a system interrrupt occurs. The system then determines which of the conditions caused the interrupt and jumps to the appropriate vector. The responses to these interrupts are as follows:

- **Response byte:** Jumps to the response vector and processes incoming data from the microcontroller.
- **Abort/flush:** Jumps to the abort vector and attempts to resynchronize the system with the Apple DeskTop Bus; if this fails, a system error occurs.
- **Desktop Manager key sequence:** Jumps to the Desktop Manager vector.
- **Flush buffer key sequence:** Jumps to the flush buffer vector.
- **SRQ:** Jumps to the SRQ handler that is used to gather data from the ADB devices. (This interrupt occurs if the device has some data that it wants to transmit. The device generates a service request to catch the attention of the microcontroller.)

Table 9-3
Status byte returned by microcontroller

Bit	Condition
7	Response byte if set; otherwise, status byte
6	Abort/flush
5	Desktop Manager key sequence pressed
4	Flush buffer key sequence pressed
3	SRQ valid
2–0	If all bits are clear, then no ADB data is valid; if data is available, then the bits indicate the number of valid bytes received minus 1—between 2 and 8 bytes total (001 means 2 bytes ready, 011 means 4 bytes, and so on).

Chapter 10

Mouse Firmware

This chapter describes the Apple IIGS mouse firmware. You can read the mouse position and the status of the mouse buttons using this firmware.

Important

The material in this manual regarding soft switches and hardware registers for the Apple IIGS mouse firmware is provided for information only. Applications must use the firmware calls only if they wish to be compatible with the mouse device used in all Apple II systems.

The Apple IIGS mouse is an intelligent device that uses the Apple DeskTop Bus (ADB) to communicate with the Apple IIGS ADB microcontroller. This is a departure from the AppleMouse™ card and the Apple IIc mouse interface, each of which depends extensively on firmware to support the mouse. The Apple IIGS mouse firmware has a true passive mode like the AppleMouse, but it differs from the Apple IIc mouse, which requires interrupts to function.

Certain devices, to operate properly, must be the sole source of interrupts within a system because they have critical times during which they require immediate service by the microprocessor. An interrupting communications card is a good example of a device that has a critical service interval. If it is not serviced quickly, characters might be lost. The true passive mode permits such devices to operate correctly. The passive mode also prevents the 65C816 from being overburdened with interrupts from the mouse firmware, as can occur in the Apple IIc if the mouse is moved rapidly while an application program is running.

The Apple IIGS mouse firmware can cause an interrupt only if all of the following conditions are true:

- The interrupt mode is selected.
- The mouse device is on.
- An interrupt condition has occurred.
- A vertical blanking signal (VBL) has occurred.

Unlike the Apple IIc mouse, which interrupts whenever the mouse device is moved, the Apple IIGS mouse device interrupts in synchronization with the VBL. This automatically limits the total number of mouse firmware interrupts to 60 per second, cutting down on the overhead the mouse device puts on the 65C816. If an interrupt condition (determined by the mode byte setting) occurs, it will be passed to the 65C816 only when the next VBL occurs.

Warning

Because the mouse firmware information is updated only once each vertical blanking interval, your program must be certain that at least one vertical blanking interval has elapsed between mouse reads if it expects to obtain new information from the mouse device.

Mouse position data

When the mouse is moved, data is returned as a delta move as compared to its previous position, where the change in X or Y direction can be as much as to ± 63 counts. The maximum value of 63 in either direction represents approximately 0.8 inch of travel.

❖ *Note:* A delta move represents a number of counts change in position as compared to the preceding position that the mouse occupied. The Apple IIGS mouse firmware converts this relative-position data (called a *delta*) to an absolute position.

The mouse device also provides the following information to the mouse firmware:

- current button 0 and button 1 data (1 if down, 0 if up)
- delta position since last read

❖ *Note:* At power up or reset, the GLU chip enters a noninterrupt state and also turns the mouse interrupts off.

The ADB microcontroller automatically processes mouse data. The microcontroller periodically polls the mouse to check for activity. If the mouse device is moved or its button is pushed, 2 bytes are sent to the microcontroller. The microcontroller sends both mouse data bytes to the GLU chip (byte Y followed by byte X; this enables the interrupt). The 65C816 then checks the status register to verify that a mouse interrupt has occurred, the 2 data bytes have been read, and mouse byte Y was read first. The GLU chip clears the interrupt when the second byte has been read. To prevent overruns, the microcontroller writes mouse data only when the registers are empty (after byte X has been read by the system). Table 10-1 shows the 16 bits returned by the Apple IIGS mouse firmware.

Table 10-1
Apple IIGS mouse data bits

Bit	Function
15	Button 0 status
14–8	Y movement (negative = up, positive = down)
7	Button 1 status
6–0	X movement (negative = left, positive = right)

Register addresses—firmware only

Table 10-2 shows the contents of the register addresses that the ADB microcontroller uses to transmit Apple IIGS mouse data and status information to the 65C816.

Table 10-2
Apple IIGS mouse register addresses

Address	Function
$C027	GLU status register, defined as follows:
	Bit 0 = d Must not be altered by mouse
	Bit 1 = 0 X position available (read only)
	Bit 1 = 1 Y position available (read only)
	Bit 2 = k Must not be altered by mouse
	Bit 3 = k Must not be altered by mouse
	Bit 4 = d Must not be altered by mouse
	Bit 5 = d Must not be altered by mouse
	Bit 6 = 1 Mouse interrupt enable (read or write)
	Bit 7 = 1 Mouse register full (read only)
	k Used by keyboard handlers
	d Used by ADB handlers
$C024	Mouse data register:
	First read yields X position data and button 1 data. Second read yields Y position data and button 0 data.

To enable mouse interrupts, set bit 6 of location $C027 to 1. Recall, however, that only this bit and no other should be changed. This entails reading the current contents, changing only that single bit and then writing the modified value back into the register.

If mouse interrupts are enabled, the firmware determines whether the interrupt came from the mouse by reading bits 6 and 7 of $C027; if both bits = 1, then a mouse interrupt is pending.

Reading mouse position data—firmware only

The following sequence of steps must be taken, in this exact order, for accurate mouse readings to be obtained. Failure to perform the steps in this order will necessitate some corrective action because the data will be contaminated. Contaminated data is a condition that occurs when the X and Y values that you are trying to read are from different VBL reads of the mouse.

□ Read bit 7 of $C027.

If bit 7 = 0, then X and Y data is not yet available.
If bit 7 = 1, then data is available, but could be contaminated.

- Read bit 1 of $C027 only if bit 7 = 1.

  If bit 1 = 0, then X and Y data are not contaminated and can be read. The first read of $C024 returns X data and button 1 data; the second read of $C024 returns Y data and button 0 data.

  Use caution when using indexed instructions. The false read and write results of some indexed instructions can cause X data to be lost and Y data to appear where X data was expected.

  If bit 1 = 1 and $C024 has not been read, then the data in $C024 is contaminated and must be considered useless. If this condition occurs, perform the following steps:

  - Read $C024 one time only.
  - Ignore the byte read in.

Exit the mouse read routine without updating the X, Y, or button data. This will not harm the program; however, it guarantees that the next time the program reads mouse positions, the positions will be accurate.

The data bytes read in contain the following information:

- **X data byte**

  If bit 7 = 0, then mouse button 1 is up.
  If bit 7 = 1, then mouse button 1 is down.

- **Bit 0–6 delta mouse move**

  If bit 6 = 0, then a positive move is made up to $3F (63).
  If bit 6 = 1, then a negative move in two's complement is made up to $40 (64).

- **Y data byte**

  If bit 7 = 0, then mouse button 0 is up.
  If bit 7 = 1, then mouse button 0 is down.

- **Bit 0–6 delta mouse move**

  If bit 6 = 0, then a positive move is made up to $3F (63).
  If bit 6 = 1, then a negative move in two's complement is made up to $40 (64).

Position clamps

When the mouse moves the cursor across the screen, the cursor is allowed to move only within specified boundaries on the screen. These boundaries are the maximum cursor positions on the screen in the X and Y directions. These maximum positions are indicated to the firmware by clamps.

Clamps are data values that specify a maximum or minimum value for some other variable. In this instance, the mouse clamps specify the minimum and maximum positions of the cursor on the screen.

The mouse clamps reside in RAM locations reserved for the firmware. You should only access these locations using the Apple IIGS tools.

Using the mouse firmware

You can use the mouse firmware by way of assembly language or BASIC. There are several procedures and rules to follow to be effective in either language. The following paragraphs outline these procedures and rules and give examples of the use of the mouse firmware from each of these languages.

Firmware entry example using assembly language

To use a mouse routine from assembly language, read the location corresponding to the routine you want to call (see Table 10-4 at the end of this chapter). The value read is the offset of the entry point to the routine to be called.

❖ *Note:* Interrupts must be disabled on every call to the mouse firmware.

The following assembly code example correctly sets up the entry point for the mouse firmware. Note that n is the slot number of the mouse. To use the code, you must decide which mouse firmware command you wish to use and then duplicate the code for each of the routines you use. For example, to use SERVEMOUSE from assembly code, you would replace the line `SETMENTRY LDA SETMOUSE` with a line that reads `SERVEMENTRY LDA SERVEMOUSE`, where SERVEMOUSE is $Cn13. Table 10-4 defines all of the offset locations for the built-in mouse firmware routines.

```
SETMOUSE    EQU  $Cn12          ;Offset to SETMOUSE offset ($C412 for Apple IIGS)
SETMENTRY   LDA  SETMOUSE       ;Get offset into code
            STA  TOMOUSE+2      ;Modify operand
            LDX  Cn             ;Where Cn = C4 in Apple IIGS
            LDY  n0             ;Where n = 40 in Apple IIGS
            PHP                 ;Save interrupt status
            SEI                 ;Guarantees no interrupts
            LDA  #$01           ;Turn mouse passive mode on
            JSR  TOMOUSE        ;JSR to a modified JMP instruction
            BCS  ERROR          ;C = 1 if illegal-mode-entered error
            PLP                 ;Restore interrupt status
            RTS                 ;Exit
ERROR       PLP                 ;Restore interrupt status
            JMP  ERRORMESSGE    ;Exit to error routine
TOMOUSE     JMP  $Cn00          ;Modified operand for correct entry point; $C400 for
                                 Apple IIGS
```

Firmware entry example using BASIC

To turn the mouse on using BASIC, execute the following code:

```
PRINT  CHR$(4);"PR#4"     :REM  Mouse ready for output
PRINT  CHR  (1)           :REM  1 turns the mouse on from BASIC
PRINT  CHR$(4);"PR#0"     :REM  Restore screen output
```

❖ *Note:* Use `PRINT  CHR$(4);"PR#3"` to return to 80-column mode.

To accept outputs from BASIC, the firmware changes the output links at $36 and $37 to point to $C407 and performs an INITMOUSE routine (resets the mouse clamps to their default values and positions the mouse to location 0,0).

To turn the mouse off, execute the following BASIC program:

```
PRINT  CHR$(4) ; "PR#4"   :REM  Mouse ready for output
PRINT  CHR  (0)           :REM  0 turns the mouse off from BASIC
PRINT  CHR$(4) ; "PR#0"   :REM  Restore screen output
```

❖ *Note:* Use `PRINT  CHR$(4);"PR#3"` to return to 80 columns.

To read mouse position and button statuses from BASIC, execute the following code:

```
PRINT  CHR$(4); "IN#4"    :REM  Mouse ready for input
INPUT  X, Y, B            :REM  Input mouse position
PRINT  CHR$(4); "IN#0"    :REM  Return keyboard as the input device when mouse
                                positions have been read
```

When the mouse is turned on from BASIC (for data entry), the firmware changes the input links at $38 and $39 to point to $C405. When you execute an INPUT statement while the input link is set for mouse input, the firmware performs a READMOUSE operation before converting the screen-hole data to decimal ASCII and places the converted input data in the input buffer at $200.

In BASIC, the mouse runs in passive mode or a noninterrupt mode. Clamps are set automatically to 0000–1023 ($0000–$03FF) in both the X and the Y direction, and position data in the screen holes are set to 0.

During execution of a BASIC INPUT statement, the firmware reads the position changes (deltas) from the ADB mouse and adds them to the absolute position in the screen holes, clamping the positions if necessary, and converts the absolute positions in the screen holes to ASCII format. The firmware then places that data, with the button 0 status, in the input buffer, issues a carriage return, and returns to BASIC.

❖ *Note:* The term *screen holes* has absolutely nothing to do with the appearance of anything on the actual display. Screen holes are simply unused bytes in the memory area normally reserved for screen-display operations. Because screen holes are unused by the display circuitry, they can be used by the firmware for other purposes.

Reading button 1 status

Button 1 status cannot be returned to a BASIC program. This would add another input variable to the input buffer, and an error message that states `?EXTRA IGNORED` would be displayed.

If you want to read button 1 status, you can use the BASIC Peek command to read the screen hole that contains that data. The data returned to the input buffer is in the following form:

```
s x1 x2 x3 x4 x5, s y1 y2 y3 y4 y5, sb B0 b5 cr
```

where

`s` = Sign of absolute position

`x1...x5` = Five ASCII characters indicating the decimal value of X

`y1...y5` = Five ASCII characters indicating the decimal value of Y

`sb` = Minus sign (–) if key on keyboard was pressed during INPUT statement entry and plus (+) if no key was pressed during INPUT statement entry

`B0` = ASCII space character

`b5` = 1 if button 0 is pressed now and was also pressed during last INPUT statement entry

= 2 if button 0 is pressed now but was not pressed during last INPUT statement entry

= 3 if button 0 is not pressed now but was pressed during last INPUT statement entry

= 4 if button 0 is not pressed now and was not pressed during last INPUT statement entry

`cr` = Carriage return (required as a terminator before control is passed from firmware back to BASIC)

❖ *Note:* The BASIC program must reset the key strobe at $C010 if sb returns to a negative state. POKE 49168,0 resets the strobe.

The mouse is resident in the Apple IIGS internal slot 4. When the mouse is in use, the main memory screen holes for slot 4 hold X and Y absolute position data, the current mode, button 0/1 status, and interrupt status. Eight additional bytes are used for mouse information storage; they hold the maximum and minimum clamps for the mouse's absolute position.

Table 10-3 shows the mouse's screen-hole use when Apple IIGS firmware is used. Figures 10-1 and 10-2 show how the bits of the button interrupt status byte and the mode byte are assigned.

Table 10-3
Position and status information

Address	Use
$47C	Low byte of absolute X position
$4FC	Low byte of absolute Y position
$57C	High byte of absolute X position
$5FC	High byte of absolute Y position
$67C	Reserved and used by firmware
$6FC	Reserved and used by firmware
$77C	Button 0/1 interrupt status byte (see Figure 10-1)
$7FC	Mode byte (see Figure 10-2)

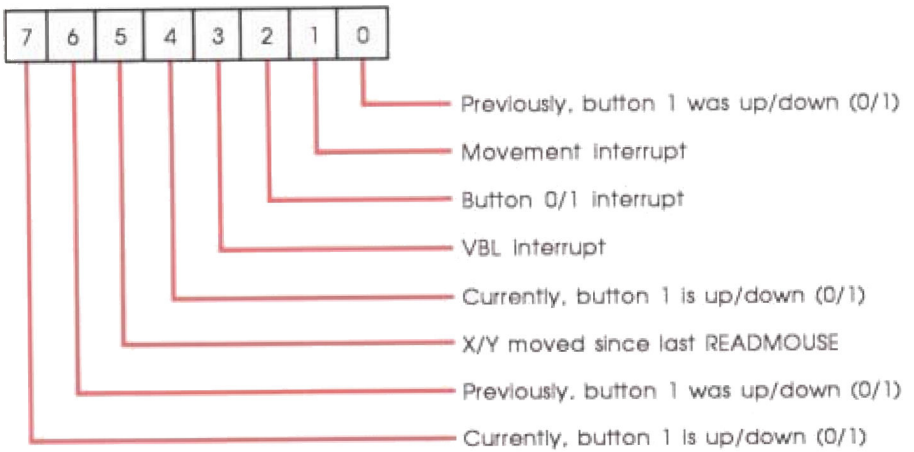


Figure 10-1
Button interrupt status byte, $77C

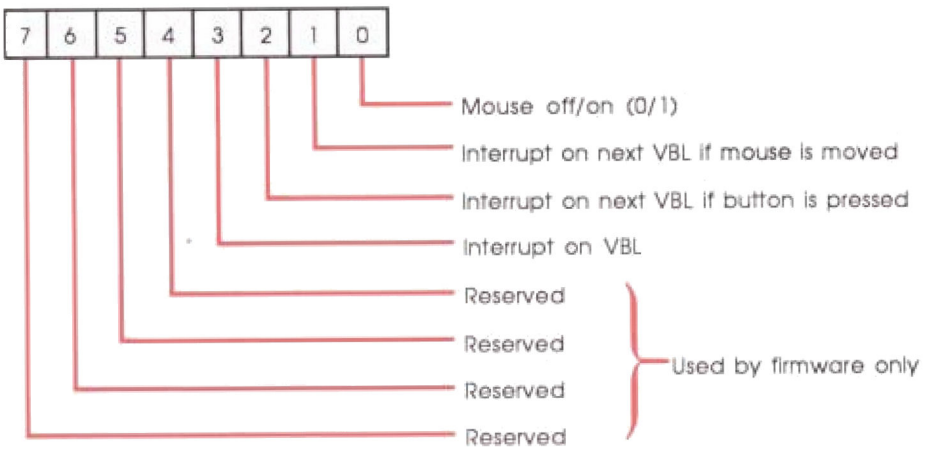


Figure 10-2

Mouse programs in BASIC

Two program examples are provided below. The first example, Mouse.Move, reads and displays the mouse position information. The second example is called Mouse.Draw and allows you to make simple drawings on the screen in low-resolution graphics mode.

Mouse.Move program

```
10  HOME
20  PRINT "MOUSE.MOVE DEMO"
30  PRINT CHR$(4);"PR#4":PRINT CHR$(1)
40  PRINT CHR$(4);"PR#0"
50  PRINT CHR$(4);"IN#4"
60  INPUT "";X,Y,S
70  VTAB 10:PRINT X;"   "Y"   "S"   "
80  IF S > 0 THEN 60
90  PRINT CHR$(4);"IN#0"
100 PRINT CHR$(4);"PR#4":PRINT CHR$(0)
110 PRINT CHR$(4);"PR#0"
```

Comments

Line 10 clears the screen to black.
Line 20 prints a heading message.
Line 30 starts up the mouse's internal program.
Line 40 establishes that subsequent PRINT commands will send information to the monitor screen.
Line 50 establishes that the subsequent INPUT command will read the mouse.
Line 60 transfers mouse position and button status readings to the numeric variables X, Y, and S.
Line 70 displays the numeric variables X, Y, and S on the 10th line of the monitor screen.
Line 80 returns the program for more mouse data if no keyboard key has been pressed. If a key has been pressed, the program drops to line 90.
Line 90 reestablishes your keyboard as the input device.
Line 100 resets the mouse position data to zero.
Line 110 reestablishes the monitor screen as the output device.
Line 120 ends the program.

Mouse.Draw program

```
10    REM MOUSE.DRAW Uses mouse to draw lo-res graphics
100   GOSUB 1000: REM TURN ON THE MOUSE
110   PRINT CHR$(4);"IN#4"
120   INPUT "";X,Y,S:REM READ MOUSE DATA
130   IF S=1 THEN 100:REM CLEAR THE SCREEN
140   IF S<0 THEN 300;REM TIME TO QUIT?
150   REM SCALE MOUSE POSITION
160   X=INT(X/25.575)
170   Y=INT(Y/25.575)
180   PLOT X,Y
190   GOTO 120

300   REM CHECK IF TIME TO QUIT
310   PRINT CHR$(4);"IN#0"
320   VTAB 22:PRINT "PRESS RETURN TO CONT OR ESC TO QUIT"
330   VTAB 22:HTAB 39:GET A$:POKE -16368,0
340   IF A$=CHR$(13) THEN HOME:GOTO 110
350   IF A$<>CHR$(27) THEN 330
360   REM CLEAR SCREEN AND ZERO MOUSE
370   TEXT:HOME
380   PRINT CHR$(4);"PR#4":PRINT CHR$(1)
390   PRINT CHR$(4);"PR#0"
400   END

1000  REM Clear the screen and initialize the mouse
1010  HOME:GR
1020  COLOR = 15
1030  PRINT CHR$(4);"PR#4":PRINT CHR$(1)
1040  PRINT CHR$(4);"PR#0"
1050  RETURN
```

Comments

Line 10 reminds you what the program does.
Line 100 calls the subroutine at lines 1000 through 1050.
Line 110 establishes that the subsequent INPUT command will read the mouse.
Line 120 transfers mouse position and button status data to the numeric variables X, Y, and S.
Line 130 reinitializes the mouse if its button is pressed.
Line 140 sends the program to its exit routine if a key on the Apple keyboard has been pressed.
Line 150 reminds you what the next two lines do.
Lines 160 and 170 convert the range of mouse position numbers (0 to 1023) to the range of low-resolution graphics coordinates (0 to 40).
Line 180 plots a point on the monitor screen.
Line 190 sends the program back for more mouse data.
Line 300 reminds you what lines 310 through 400 do.
Line 310 tells the computer to accept input from its keyboard.
Line 320 prints prompting instructions on line 22 of the screen.
Line 330 fetches your answer to the prompt and changes the button status number back to positive (it becomes negative whenever you press a key on the Apple keyboard).
Line 340 sends the program back to reporting mouse data if you pressed Return.
Line 350 fetches another answer if you press any key except Esc.
Line 360 reminds you what happens next.
Line 370 cancels graphics mode and clears the screen.
Line 380 resets the mouse position data to zero.
Line 390 reestablishes the monitor screen as the output device.
Line 400 ends the program.
Line 1000 reminds you what the following subroutine does.
Line 1010 clears the monitor screen and sets up Apple's low-resolution graphics mode.
Line 1020 establishes that the cursor will be white.
Line 1030 starts up the mouse's internal program.
Line 1040 establishes that subsequent PRINT commands will send information to the monitor screen.
Line 1050 returns to the main program (line 100).

Summary of mouse firmware calls

The firmware calls to enter mouse routines are listed in Table 10-4. These calls conform to Pascal 1.1 protocol for peripheral cards.

Table 10-4
Mouse firmware calls

Location	Routine	Definition
Pascal firmware entry points for the mouse		
$C40D	PINIT	Pascal INIT device (not implemented)
$C40E	PREAD	Pascal READ character (not implemented)
$C40F	PWRITE	Pascal WRITE character (not implemented)
$C410	PSTATUS	Pascal get device status (not implemented)
$C411 = $00		Indicates that more routines follow
Routines implemented on Apple IIGS, Apple II, and AppleMouse card		
$C412	SETMOUSE	Sets mouse mode
$C413	SERVEMOUSE	Services mouse interrupt
$C414	READMOUSE	Reads mouse position
$C415	CLEARMOUSE	Clears mouse position to 0 (for delta mode)
$C416	POSMOUSE	Sets mouse position to user-defined position
$C417	CLAMPMOUSE	Sets mouse bounds in a window
$C418	HOMEMOUSE	Sets mouse to upper-left corner of clamping window
$C419	INITMOUSE	Resets mouse clamps to default values; sets mouse position to 0,0
Entry points compatible with AppleMouse card; they do nothing in Apple IIGS		
$C41A	DIAGMOUSE	Dummy routine; clears c and performs an RTS
$C41B	COPYRIGHT	Dummy routine; clears c and performs an RTS
$C41C	TIMEDATA	Dummy routine; clears c and performs an RTS
$C41D	SETVBLCNTS	Dummy routine; clears c and performs an RTS
$C41E	OPTMOUSE	Dummy routine; clears c and performs an RTS
$C41F	STARTTIMER	Dummy routine; clears c and performs an RTS
Other significant locations		
$C400	BINITENTRY	Initial entry point when coming from BASIC
$C405	BASICINPUT	BASIC input entry point (opcode SEC) Pascal ID byte
$C407	BASICOUTPUT	BASIC output entry point (opcode CLC) Pascal ID byte
$C408 = $01		Pascal generic signature byte
$C40C = $20		Apple technical-support ID byte
$C4FB = $D6		Additional ID byte

Pascal calls

Pascal recognizes the mouse as a valid device; however, Pascal is not supported by the firmware. A Pascal driver for the mouse is available from Apple to interface programs with the mouse. Pascal calls PInit, PRead, PWrite, and PStatus return with the X register set to 3 (Pascal illegal operation error) and the carry flag set to 1. Following is a list of Pascal firmware calls.

PInit

Function Not implemented (just an entry point in case user calls it by mistake).

Input All registers and status bits.

Output X = $03 (error 3 = bad mode: illegal operation). c = 1 (always).
Screen holes: unchanged.

PRead

Function Not implemented (just an entry point in case user calls it by mistake).

Input All registers and status bits.

Output X = $03 (error 3 = bad mode: illegal operation). c = 1 (always).
Screen holes: unchanged.

PWrite

Function Not implemented (just an entry point in case user calls it by mistake).

Input All registers and status bits.

Output X = $03 (error 3 = bad mode: illegal operation). c = 1 (always).
Screen holes: unchanged.

PStatus

Function Not implemented (just an entry point in case user calls it by mistake).

Input All registers and status bits.

Output X = $03 (error 3 = Bad mode: illegal operation). c = 1 (always).
Screen holes: unchanged.

Assembly-language calls

This section describes the assembly-language firmware calls. When you use the mouse from assembly language, you must keep several items in mind.

- For built-in firmware, n = mouse slot number 4.
- The following bits and registers are not changed by mouse firmware:
  - e, m, I, x
  - direct register
  - data bank register
  - program bank register
- Mouse screen holes should not be changed by an application program, with one exception: During execution of the POSMOUSE function, new mouse coordinates are written by the application program directly into the screen holes. No other mouse screen hole can be changed by an application program without adversely affecting the mouse.
- The 65C816 assumes that the mouse firmware is entered in the following machine state:
  - 65C816 is in emulation mode.
  - Direct register = $0000.
  - Data bank register = $00.
  - System speed = fast or slow (does not matter which).
  - Text page 1 shadowing is on to allow access to screen-hole data.

Here are the actual firmware routines. Notice that each is specified by its offset entry address. Recall that the offset entry point is a value at a given location (for example, $C412) to which you add the value of the main entry point (for example, $C400) to obtain the actual address to which the processor must jump to execute the routine.

SETMOUSE, $C412

Function Sets mouse operation mode.

Input
A = mode ($00 to $0F are the only valid modes).
X = Cn for standard interface (Apple IIGS mouse not used).
Y = n0 for standard interface (Apple IIGS mouse not used).

Output
A = mode if illegal mode entered; otherwise, A is scrambled.
X, Y, V, N, Z = scrambled.
c = 0 if legal mode entered (mode is <= $0F).
c = 1 if illegal mode entered (mode is > $0F).
Screen holes: Only mode bytes are updated.

SERVEMOUSE, $C413

Function Tests for interrupt from mouse and resets mouse's interrupt line.

Input
X = Cn for standard interface (Apple IIGS mouse not used).
Y = n0 for standard interface (Apple IIGS mouse not used).

Output
X, Y, V, N, Z = scrambled.
c = 0 if mouse interrupt occurred.
c = 1 if mouse interrupt did not occur.
Screen holes: Interrupt status bits updated to show current status.

READMOUSE, $C414

Function Reads delta (X/Y) positions, updates absolute X/Y positions, and reads button statuses from ADB mouse.

Input
A = not affected.
X = Cn for standard interface (Apple IIGS mouse not used).
Y = n0 for standard interface (Apple IIGS mouse not used).

Output
A, X, Y, V, N, Z = scrambled.
c = 0 (always).
Screen holes: SLO, XHI, YLO, YHI buttons and movement status bits updated; interrupt status bits are cleared.

CLEARMOUSE, $C415

Function Resets buttons, movement, and interrupt status to 0, X, and Y. (This mode is intended for delta mouse positioning instead of the normal absolute positioning.)

Input
A = not affected.
X = Cn for standard interface (Apple IIGS mouse not used).
Y = n0 for standard interface (Apple IIGS mouse not used).

Output
A, X, Y, V, N, Z = scrambled.
c = 0 (always).
Screen holes: SLO, XHI, YLO, YHI buttons and movement status bits updated; interrupt status bits are cleared.

POSMOUSE, $C416

Function Allows user to change current mouse position.

Input User places new absolute X/Y positions directly in appropriate screen holes.
X = Cn for standard interface (Apple IIGS mouse not used).
Y = n0 for standard interface (Apple IIGS mouse not used).

Output A, X, Y, V, N, Z = scrambled.
c = 0 (always).
Screen holes: User changed X and Y absolute positions only; bytes changed.

CLAMPMOUSE, $C417

Function Sets up clamping window for mouse use. Power-up default values are 0 to 1023 ($0000 to $03FF).

Input A = 0 if entering X clamps.
A = 1 if entering Y clamps.

Clamps are entered in slot 0 screen holes by the user as follows:

$478 = low byte of low clamp.
$4F8 = low byte of high clamp.
$578 = high byte of low clamp.
$5F8 = high byte of high clamp.

X = Cn for standard interface (Apple IIGS mouse not used).
Y = n0 for standard interface (Apple IIGS mouse not used).

Output A, X, Y, V, N, Z = scrambled.
c = 0 (always).
Screen holes: X/Y absolute position is set to upper-left corner of clamping window. Clamping RAM values in bank $E0 are updated.

❖ *Note:* The Apple IIGS mouse firmware performs an automatic HOMEMOUSE operation after a CLAMPMOUSE. HOMEMOUSE execution is required because the delta information is being fed to the firmware instead of ±1's, as in the case of the Apple II mouse and the 6805 AppleMouse microprocessor card. The delta information from the Apple IIGS ADB mouse alters the absolute position of the screen pointer, using clamping techniques not used by the other two mouse devices.

HOMEMOUSE, $C418

Function Sets X/Y absolute position to upper-left corner of clamping window.

Input
A = not affected.
X = Cn for standard interface (Apple IIGS mouse not used).
Y = n0 for standard interface (Apple IIGS mouse not used).

Output
A, X, Y, V, N, Z = scrambled.
c = 0 (always)
Screen holes: User changed X and Y absolute positions only; bytes changed.

INITMOUSE, $C419

Function Sets screen holes to default values and sets clamping window to default value of 0000 to 1023 ($0000, $03FF) in both the X and Y directions; resets GLU mouse interrupt capabilities.

Input
A = not affected.
X = Cn for standard interface (Apple IIGS mouse not used).
Y = n0 for standard interface (Apple IIGS mouse not used).

Output
A, X, Y, V, N, Z = scrambled.
c = 0 (always)
Screen holes: X/Y positions, button statuses, and interrupt status are reset.

❖ *Note:* Button and movement statuses are valid only after a READMOUSE operation. Interrupt status bits are valid only after a SERVEMOUSE operation. Interrupt status bits are reset after READMOUSE. Read and use or read and save the appropriate mouse screen-hole data before enabling or reenabling 65C816 interrupts.

Appendix A

Roadmap to the Apple IIGS Technical Manuals

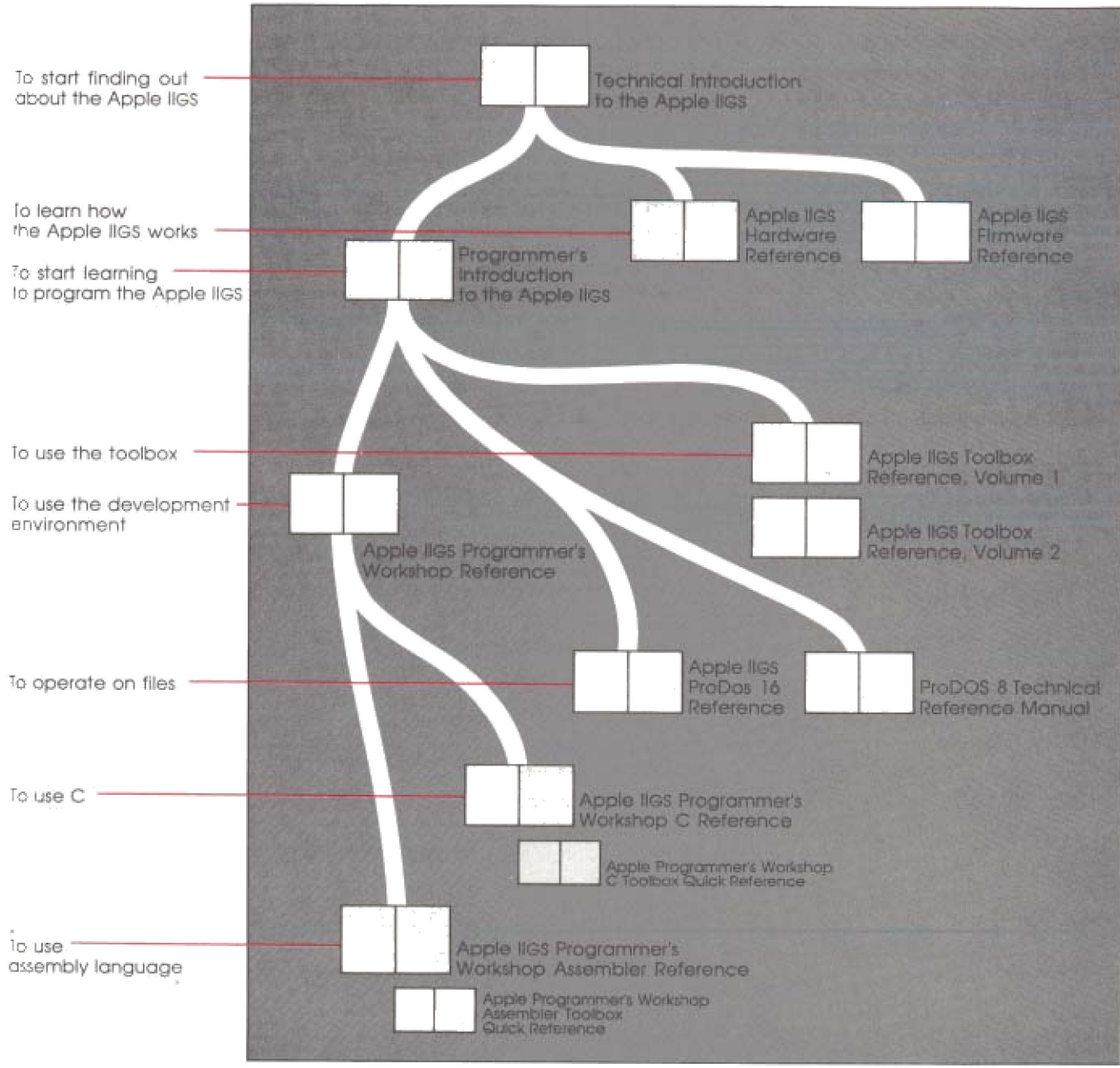
The Apple IIGS personal computer has many advanced features, making it more complex than earlier models of the Apple II. To describe it fully, Apple has produced a suite of technical manuals. Depending on the way you intend to use the Apple IIGS, you may need to refer to a select few of the manuals, or you may need to refer to most of them.

The technical manuals are listed in Table A-1. Figure A-1 is a diagram showing the relationships among the different manuals.

Table A-1
Apple IIGS technical manuals

Title	Subject
Technical Introduction to the Apple IIGS	What the Apple IIGS is
Apple IIGS Hardware Reference	Machine internals—hardware
Apple IIGS Firmware Reference	Machine internals—firmware
Programmer's Introduction to the Apple IIGS	Concepts and a sample program
Apple IIGS Toolbox Reference, Volume 1	How the tools work and some toolbox specifications
Apple IIGS Toolbox Reference, Volume 2	More toolbox specifications
Apple IIGS Programmer's Workshop Reference	The development environment
Apple IIGS Programmer's Workshop Assembler Reference	Using the APW assembler
Apple IIGS Programmer's Workshop C Reference	Using C on the Apple IIGS
ProDOS 8 Technical Reference Manual	Standard Apple II operating system
Apple IIGS ProDOS 16 Reference	Apple IIGS operating system and System Loader
Human Interface Guidelines: The Apple Desktop Interface	Guidelines for the desktop interface
Apple Numerics Manual	Numerics for all Apple computers

Figure A-1
Roadmap to the technical manuals



The introductory manuals

These books are introductory manuals for developers, computer enthusiasts, and other Apple IIGS owners who need technical information. As introductory manuals, their purpose is to help the technical reader understand the features of the Apple IIGS, particularly the features that are different from other Apple computers. Having read the introductory manuals, the reader will refer to specific reference manuals for details about a particular aspect of the Apple IIGS.

The technical introduction

The *Technical Introduction to the Apple IIGS* is the first book in the suite of technical manuals about the Apple IIGS. It describes all aspects of the Apple IIGS, including its features and general design, the program environments, the toolbox, and the development environment.

Where the *Apple IIGS Owner's Guide* is an introduction from the point of view of the user, the technical introduction manual describes the Apple IIGS from the point of view of the program. In other words, it describes the things the programmer has to consider while designing a program, such as the operating features the program uses and the environment in which the program runs.

The programmer's introduction

When you start writing Apple IIGS programs, the *Programmer's Introduction to the Apple IIGS* provides the concepts and guidelines you need. It is not a complete course in programming, only a starting point for programmers writing applications that use the Apple desktop interface (with windows, menus, and the mouse). It introduces the routines in the Apple IIGS Toolbox and the program environment they run under. It includes a sample event-driven program that demonstrates how a program uses the toolbox and the operating system. (An event-driven program waits in a loop until it detects an event such as a click of the mouse button.)

The machine reference manuals

There are two reference manuals for the machine itself: the *Apple IIGS Hardware Reference* and the *Apple IIGS Firmware Reference*. These books contain detailed specifications for people who want to know exactly what's inside the machine.

The hardware reference manual

The *Apple IIGS Hardware Reference* is required reading for hardware developers, and it will also be of interest to anyone else who wants to know how the machine works. Information for developers includes the mechanical and electrical specifications of all connectors, both internal and external. Information of general interest includes descriptions of the internal hardware, which provide a better understanding of the nachine's features.

The firmware reference manual

The *Apple IIGS Firmware Reference* describes the programs and subroutines that are stored in the machine's read-only memory (ROM), with two significant exceptions: Applesoft BASIC and the toolbox, which have their own manuals. The firmware reference manual includes information about interrupt routines and low-level I/O subroutines for the serial ports, the disk port, and the Apple DeskTop Bus interface, which controls the keyboard and the mouse. The manual also describes the Monitor, a low-level programming and debugging aid for assembly-language programs.

The toolbox reference manuals

Like the Macintosh, the Apple IIGS has a built-in toolbox. The *Apple IIGS Toolbox Reference,* Volume 1, introduces concepts and terminology and tells how to use some of the tools. The *Apple IIGS Toolbox Reference,* Volume 2, contains information about the rest of the tools and also tells how to write and install your own tool set..

Of course, you don't have to use the toolbox at all. If you only want to write simple programs that don't use the mouse, or windows, or menus, or other parts of the desktop user interface, then you can get along without the toolbox. However, if you are developing an application that uses the desktop interface or if you want to use the Super Hi-Res graphics display, you'll find the toolbox to be indispensable.

In applications that use the desktop user interface, commands appear as options in pull-down menus, and material being worked on appears in rectangular areas of the screen called *windows*. The user selects commands or other material by using the mouse to move a pointer around on the screen.

The programmer's workshop reference manual

The Apple IIGS Programmer's Workshop (APW) is the development environment for the Apple IIGS computer. APW is a set of programs that enables developers to create and debug application programs on the Apple IIGS. The *Apple IIGS Programmer's Workshop Reference* includes information about the APW Shell, Editor, Linker, Debugger, and utility programs; these are the parts of the workshop that all developers need, regardless of which programming language they use.

The APW reference manual describes the way you use the workshop to create an application and includes examples and illustrations to show how this is done. In addition, this manual documents the APW Shell to provide the information necessary to write an APW utility or a language compiler for the workshop.

Included in the APW reference manual are complete descriptions of two standard Apple IIGS file formats: the text file format and the object module format. The text file format is used for all files written or read as "standard ASCII files" by Apple IIGS programs running under ProDOS 16. The object module format is used for the ouptut of all APW compilers and for all files loadable by the Apple IIGS System Loader.

The programming-language reference manuals

Apple currently provides a 65C816 assembler and a C compiler. Other compilers can be used with the workshop, provided that they follow the standards defined in the *Apple IIGS Programmer's Workshop Reference*.

There is a separate reference manual for each programming language on the Apple IIGS. Each manual includes the specifications of the language and of the Apple IIGS libraries for the language, and describes how to use the assembler or compiler for that language. The manuals for the languages Apple provides are the *Apple IIGS Programmer's Workshop Assembler Reference* and the *Apple IIGS Programmer's Workshop C Reference*.

The *Apple IIGS Programmer's Workshop Reference* and the two programming-language manuals are available through the Apple Programmer's and Developer's Association.

The operating-system reference manuals

There are two operating systems that run on the Apple IIGS: ProDOS 16 and ProDOS 8. Each operating system is described in its own manual: *ProDOS 8 Technical Reference Manual* and *Apple IIGS ProDOS 16 Reference*. ProDOS 16 uses the full power of the Apple IIGS. The ProDOS 16 manual describes its features and includes information about the System Loader, which works closely with ProDOS 16. If you are writing programs for the Apple IIGS, whether as an application programmer or a system programmer, you are almost certain to need the ProDOS 16 reference manual.

ProDOS 8, previously just called *ProDOS*, is the standard operating system for most Apple II computers with 8-bit CPUs (Apple IIc, IIe, and 64K II Plus). It also runs on the Apple IIGS. As a developer of Apple IIGS programs, you need the *ProDOS 8 Technical Reference Manual* only if you are developing programs to run on 8-bit Apple II's as well as on the Apple IIGS.

The all-Apple manuals

In addition to the Apple IIGS manuals mentioned above, there are two manuals that apply to all Apple computers: *Human Interface Guidelines: The Apple Desktop Interface* and *Apple Numerics Manual*. If you develop programs for any Apple computer, you should know about those manuals.

The *Human Interface Guidelines* manual describes Apple's standards for the desktop interface of any program that runs on an Apple computer. If you are writing a commercial application for the Apple IIGS, you should be fully familiar with the contents of this manual.

The *Apple Numerics Manual* is the reference for the Standard Apple Numeric Environment (SANE™), a full implementation of the *IEEE Standard for Binary Floating-Point Arithmetic* (IEEE Std 754-1985). The functions of the Apple IIGS SANE tool set match those of the Macintosh SANE package and of the 6502 assembly-language SANE software. If your application requires accurate or robust arithmetic, you'll probably want to use the SANE routines in the Apple IIGS. The *Apple IIGS Toolbox Reference* tells how to use the SANE routines in your programs. The *Apple Numerics Manual* is the comprehensive reference for the SANE numerics routines.

Appendix B

Firmware ID Bytes

The firmware ID bytes are used to identify the particular hardware system on which you are currently working. Table B-1 lists the locations from which you can read ID information. Each system maintains three separate ID byte locations, as indicated in the table. If all three ID bytes match for a given system type, you will know that your software is running on that particular system.

Table B-1
ID Information locations

System	Main ID ($FBB3)	Sub ID1 (FBC0)	Sub ID2 ($FBBF)
Apple II	$38	$60	$2F
Apple II Plus	$EA	$EA	$EA
Apple IIe	$06	$EA	$C1
Apple IIe Plus	$06	$E0	$00
Apple IIGS	$06	$E0	$00
Apple IIc	$06	$00	$FF
Apple IIc Plus	$06	$00	$00

To distinguish the Apple IIGS from an Apple IIe Plus (the ID bytes are identical), run the following short routine with the ROM enabled in the language card.

```
SEC                 ;c = 1 as a starting point
JSR   $FE1F         ;RTS for Apple II computers
                    ;prior to the Apple IIGS
BCS   ITSAPPLEIIE   ;If c = 1, then the system is an old Apple II
BCC   ITSAppleIIGs  ;If c = 0, then the system is a Apple IIGS or later and the
                     registers are returned with the information in Table B-2.
```

Table B-2
Register bit information

Register	Bit	Information
A	15–7	Reserved
	6	1, if system has a memory expansion slot
	5	1, if system has an IWM port
	4	1, if system has a built-in clock
	3	1, if system has Apple DeskTop Bus
	2	1, if system has SCC
	1	1, if system has external slots
	0	1, if system has internal ports
Y	15–8	Machine ID: 00 Apple IIGS 1–FF Future machines
X	7–0	ROM version number

The Y register contains the machine ID; the X register contains the ROM version number.

❖ *Note:* If the ID call was made in emulation mode, only the low 8 bits of X, A, and Y are returned correctly; however, the c bit is accurate. If the call was made in native mode, the c bit as well as register information is accurate as shown in Table B-2 and is returned in full 16-bit native mode. The c bit is the carry bit in the processor status register. If the value returned in Y is $00, the value in A should be considered to be $7F.

Appendix C

Firmware Entry Points in Bank $00

Apple Computer, Inc. will maintain the entry points described within this document in any future Apple IIGS or Apple II–compatible machine that Apple produces. No other entry points will be maintained. Use of the entry points in this document will ensure compatibility with Apple IIGS and future Apple II–compatible machines. Note that these entry points are specific to Apple IIGS and Apple IIGS–compatible machines and do not necessarily apply to Apple IIe or Apple IIc machines.

As an alternative to using these entry points, note that you can also use the Miscellaneous Tool Set FWENTRY firmware function.

For *all* of the routines defined in this chapter, the following definitions apply:

- A represents the lower 8 bits of the accumulator.
- B represents the upper 8 bits of the accumulator.
- X and Y represent 8-bit index registers.
- DBR represents the data bank register.
- K represents the program bank register.
- P represents the processor status register.
- S represents the processor stack register.
- D represents the direct-page register.
- e represents the emulation-mode bit.
- c represents the carry flag.
- ? represents a value that is undefined.

Warning

For *all* of the routines in this appendix, the following environment variables must be set with the values shown here:

- The e bit must be set to 1.
- The decimal mode must be set to 0.
- K must be set to $00.
- D must be set to $0000.
- DBR must be set to $00.

Following are descriptions of the firmware routines supported as entry points in current and future models of the Apple II family, starting with the Apple IIGS.

$F800 **PLOT** Plot on the low-resolution screen only.

PLOT puts a single block of the color value set by SETCOL on the low-resolution display screen.

Input A = Block's vertical position (0–$2F)
X = ?
Y = Block's horizontal position (0–$27)

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = A/B/P

$F80E **PLOT1** Modify block on the low-resolution screen only.

PLOT puts a single block of the color value set by SETCOL on the low-resolution display screen. The block is plotted at current settings of GBASL/GBASH with current COLOR and MASK settings.

Input A = ?
X = ?
Y = Block's horizontal position (0–$27)

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = A/B/P

$F819 **HLINE** Draw a horizontal line of blocks on low-resolution screen only.

HLINE draws a horizontal line of blocks of the color set by SETCOL on the low-resolution graphics display.

Input
A = Block's vertical position (0–$2F)
X = ?
Y = Block's leftmost horizontal position (0–$27)

H2 = (Address = $2C); block's rightmost horizontal position (0–$27)

Output
Unchanged = X/DBR/K/D/e
Scrambled = A/Y/B/P

$F828 **VLINE** Draw a vertical line of blocks on the low-resolution screen only.

VLINE draws a vertical line of blocks of the color set by SETCOL on the low-resolution display.

Input
A = Block's top vertical position (0–$2F)
X = ?
Y = Block's horizontal position (0–$27)

V2 = (Address = $2D); block's bottom vertical position (0–$2F)

Output
Unchanged = X/DBR/K/D/e
Scrambled = A/Y/B/P

$F832 **CLRSCR** Clear the low-resolution screen only.

CLRSCR clears the low-resolution graphics display to black. If CLRSCR is called while the video display is in text mode, it fills the screen with inverse *at* sign (@) characters.

Input
A = ?
X = ?
Y = ?

Output
Unchanged = X/DBR/K/D/e
Scrambled = A/Y/B/P

$F836 **CLRTOP** Clear the top 40 lines of the low-resolution screen only.

CLRTOP clears the top 40 lines of the low-resolution graphics display (in mixed mode, clears the graphics portion of the screen to black).

Input
A = ?
X = ?
Y = ?

Output
Unchanged = X/DBR/K/D/e
Scrambled = A/Y/B/P

$F847 **GBASCALC** Calculate base address for low-resolution graphics only.

GBASCALC calculates the base address of the line on which a particular pixel is to be plotted.

Input
A = Vertical line to find address for (0–$2F)
X = ?
Y = ?

Output
Unchanged = X/Y/DBR/K/D/e
Scrambled = B/P
Special = A = GBASL

$F85F **NXTCOL** Increment color by 3.

NXTCOL adds 3 to the current color (set by SETCOL) used for low-resolution graphics.

Input
A = ?
X = ?
Y = ?

Output
Unchanged = X/Y/DBR/K/D/e
Scrambled = B/P
Special = A = New color in high or low nibble

$F864 **SETCOL** Set low-resolution graphics color.

SETCOL sets the color used for plotting in low-resolution graphics. The colors are as follows:

$0 = Black
$1 = Deep red
$2 = Dark blue
$3 = Purple
$4 = Dark green
$5 = Dark gray
$6 = Medium blue
$7 = Light blue
$8 = Brown
$9 = Orange
$A = Light gray
$B = Pink
$C = Light green
$D = Yellow
$E = Aquamarine
$F = White

Input A = Low nibble = new color to use; high nibble doesn't matter
X = ?
Y = ?

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = B/P
Special = A = New color in high or low nibble

$F871 **SCRN** Read the low-resolution graphics screen only.

SCRN returns the color value of a single block on the low-resolution graphics display. Call it with the vertical position of the block in the accumulator and horizontal position in the Y register.

Input A = Vertical line to find addr for (0–$2F)
X = ?
Y = ?

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = B/P
Special = A = Color of block specified in low nibble;
high nibble = 0

$F88C **INSDS1.2** Perform LDA (PCL,X); then fall into INSDS2.

INSDS1.2 gets the opcode to determine the instruction length of with an LDA (PCL,X) and falls into INSDS2.

Input
A = ?
X = Offset into buffer at pointer PCL/PCH
Y = ?

PCH = (Address $3B) high byte of buffer address to get opcode from in bank $00

PCL = (Address = $3A) low byte of buffer address to get opcode from in bank $00

Output
Unchanged = DBR/K/D/e
Scrambled = A/X/B/P
Special = Y = $00
LENGTH (address = $2F); contains instruction length 1 of 6502 instructions or = $00 if not a 6502 opcode

$F88E **INSDS2** Calculate length of 6502 instruction.

INSDS2 determines the length 1 of the 6502 instruction denoted by the opcode appearing in the A register.

INSDS2 returns correct instruction length 1 of 6502 opcodes only. All non-6502 opcodes return a length of $00. For compatibility reasons, the BRK opcode returns a length of $00, not $01 as one would expect it to.

Input
A = Opcode for which length is to be determined
X = ?
Y = ?

Output
Unchanged = DBR/K/D/e
Scrambled = A/X/B/P
Special = Y = $00
LENGTH (address = $2F); contains instruction length 1 of 6502 instructions or = $00 if not a 6502 opcode

$F890 **GET816LEN** Calculate length of 65C816 instruction.

GET816LEN determines the length of the 65816 instruction denoted by the opcode appearing in the A register. The BRK opcode returns a length of $01 as one would expect it to.

Input A = Opcode for which length is to be determined
X = ?
Y = ?

Output Unchanged = DBR/K/D/e
Scrambled = A/X/B/P
Special = Y = $00
LENGTH (address = $2F); contains instruction length 1 of 65C816 instructions

$F8D0 **INSTDSP** Display disassembled instruction.

INSTDSP disassembles and displays one instruction pointed to by the program counter PCL/PCH (addresses $3A/$3B) in bank $00.

Input A = ?
X = ?
Y = ?

Output Unchanged = DBR/K/D/e
Scrambled = A/X/Y/B/P

$F940 **PRNTYX** Print contents of Y and X registers in hex format.

PRNTYX prints the contents of the Y and X registers as four-digit hexadecimal values.

Input A = ?
X = Low hex byte to print
Y = High hex byte to print

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = A/B/P

$F941 **PRNTAX** Print contents of A and X registers in hex format.

PRNTYX prints the contents of the A and X registers as four-digit hexadecimal values.

Input A = High hex byte to print
X = Low hex byte to print
Y = ?

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = A/B/P

$F944 **PRNTX** Print contents of X register in hex format.

PRNTYX prints the contents of the X register as a two-digit hexadecimal value.

Input A = ?
X = Hex byte to print
Y = ?

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = A/B/P

$F948 **PRBLNK** Print 3 spaces.

PRBLNK outputs 3 blank spaces to the standard output device.

Input A = ?
X = ?
Y = ?

Output Unchanged = Y/DBR/K/D/e
Scrambled = B/P
Special = X = $00
A = $A0 (space ASCII code)

$F94A **PRBL2** Print X number of blank spaces.

PRBL2 outputs from 1 to 256 blanks to the standard output device.

Input A = ?
X = Number of blanks to print ($00 = 256 blanks)
Y = ?

Output Unchanged = Y/DBR/K/D/e
Scrambled = B/P
Special = X = $00
A = $A0 (space ASCII code)

$F953 **PCADJ** Adjust Monitor program counter.

PCADJ increments the program counter by 1, 2, 3, or 4, depending on the LENGTH (address $2F) byte; 0 = add 1 byte, 1 = add 2 bytes, 2 = add 3 bytes, 3 = add 4 bytes.

Note: PCL/PCH (addresses $3A/$3B) are not changed by this call. The A/Y registers contained the new program counter at the end of this call.

Input
A = ?
X = ?
Y = ?
PCL = (Address $3A) program counter low byte
PCH = (Address $3B) program counter high byte
LENGTH = (Address $2F) length 1 to add to program counter

Output
Unchanged = DBR/K/D/e
Scrambled = X/B/P
Special = A = New PCL
Y = New PCH
PCL/PCH not changed

$F962 **TEXT2COPY** Enable or Disable text Page 2 software-shadowing.

TEXT2COPY toggles the text Page 2 software-shadowing function on and off. The first access to TEXT2COPY enables shadowing, and the next access disables shadowing. When TEXT2COPY is enabled, a heartbeat task is enabled that, on every VBL, copies the information from bank $00 locations $0400–$07FF to bank $E0 locations $0400–$07FF. It then enables VBL interrupts. VBL interrupts remain on until Control-Reset is pressed or until the system is restarted. TEXT2COPY can disable the copy function, but cannot disable VBL interrupts once they are enabled.

Input
A = ?
X = ?
Y = ?

Output
Unchanged = DBR/K/D/e
Scrambled = A/X/Y/B/P

$FA40 **OLDIRQ** Go to emulation-mode interrupt-handling routines.

Jumps to the interrupt-handling routines that handle emulation-mode BRKs and IRQs. All registers are restored after the application performs an RTI at the end of its installed interrupt routines. Location $45 is not destroyed as in the Apple II, Apple II Plus, and original Apple IIe computers.

Input A = ?
X = ?
Y = ?

Output Unchanged = A/X/Y/DBR/P/B/K/D/e
Scrambled = Nothing

$FA4C **BREAK** Old 6502 break handler.

BREAK saves the 6502 registers and the program counter and then jumps indirectly through the user hooks at $03F0/$03F1. Note that this call affects the 6502 registers, not the 65C816 registers. This entry point is obsolete except in very rare circumstances.

Input A = Assumes A was stored at address $45
X = ?
Y = ?

Output Unchanged = DBR/K/D/e
Special = A5H (address $45) = A value
XREG (address $46) = X value
YREG (address $47) = Y value
STATUS (address $48) = P value
SPNT (address $49) = S stack
Pointer value

$FA59 **OLDBRK** New 65C816 break handler.

OLDBRK prints the address of the BRK instruction, disassembles the BRK instruction, and prints the contents of the 65C816 registers and memory configuration at the time the BRK instruction was executed.

Input All 65C816 registers and memory configuration saved by interrupt handler

Output Returns to Monitor after displaying information

$FA62 **RESET** Hardware reset handler.

RESET sets up all necessary warm-start parameters for the Apple IIGS. It is called by the 65C816 reset vector stored in ROM in locations $FFFC/$FFFD. If normal warm start occurs, it then exits through user vectors at $03F2/$03F3. If cold start occurs, it then exits by attempting to start a startup device such as a disk drive or AppleTalk, depending on Control Panel settings. If a program JMPs here, it *must* enter in emulation mode with the direct-page register set to $0000, the data bank register set to $00, and the program bank register set to $00, or RESET will not work.

Input K/DBR/D/e = $00

Output Doesn't return to calling program

$FAA6 **PWRUP** System cold-start routine.

PWRUP performs a partial system reset and then attempts to start the system via a disk drive or AppleTalk. PWRUP also zeros out memory in bank 00 from address $0800–$BFFF. If a program JMPs here, it *must* enter in emulation mode, with the direct-page register set to $0000, the data bank register set to $00, and the program bank register set to $00, or RESET will not work. If no startup device is available, the message `Check Startup Device` appears on the screen.

Input K/DBR/D/e = $00

Output Doesn't return to calling program

$FABA **SLOOP** Disk controller slot search loop.

SLOOP is the disk controller search loop. It searches for a disk controller beginning at the peripheral ROM space (if RAM disk, ROM disk, or AppleTalk has not been selected via the Control Panel as the startup device) pointed to by LOC0 and LOC1 (addresses $00/$01). If a startup device is found, it JMPs to that card's ROM space. If no startup device is found, the message `Check Startup Device` appears on the screen. If RAM disk or ROM disk has been selected, then the firmware JMPs to the SmartPort code that handles those startup devices. If slot 7 was selected and AppleTalk is enabled in port 7, the firmware JMPs to the AppleTalk boot code in slot 7.

Input A = ?
X = ?
Y = ?
LOC0 = (Address $00); must be $00, or startup will not occur
LOC1 = (Address $01); contains $Cn, where n = next slot number to test for a startup device

Output Doesn't return to calling program

$FAD7 **REGDSP** Display contents of registers.

REGDSP displays all 65C816 register contents stored by the firmware and Apple IIGS memory-state information, including shadowing and system speed. Displayed values include A/X/Y/K/DBR/S/D/P/M/Q/m/x/e/L. A/X/Y/S are always saved and displayed as 16-bit values, even if emulation mode or 8-bit native mode is selected.

Input A = ?
X = ?
Y = ?

Output Unchanged = DBR/K/D/e
Scrambled = A/X/Y/B/P

$FB19 **RTBL** Register names table for 6502 registers only.

This is not a callable routine. It is a fixed ASCII string. The fixed string is 'AXYPS'. Some routines require this string here, or they will not execute properly. The most significant bit of each ASCII character is set to 1.

Input No input (not a callable routine)

Output No output (not a callable routine)

$FB1E **PREAD** Read a hand controller.

PREAD returns a number that represents the position of the specified hand controller.

Input A = ?
X = 0, 1, 2, or 3 only = Paddle to read
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = A/B/P
Special = Y = Paddle count

$FB21 **PREAD4** Check timeout paddle; then read the hand controller.

PREAD4 verifies that the paddle (hand controller) is in timeout mode and then reads the paddle the same as PREAD does, returning a number that represents the position of the specified hand controller.

Input A = ?
X = 0, 1, 2, or 3 only = Paddle to read
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = A/B/P
Special = Y = Paddle count

$FB2F **INIT** Initialize text screen.

INIT sets up the screen for full window display and text Page 1.

Input A = ?
X = ?
Y = ?

Output Unchanged = DBR/K/D/e
Scrambled = X/Y/B/P
Special = A = BASL

$FB39 **SETTXT** Set text mode.

SETTXT sets screen for full text window, but does not force text Page 1.

Input A = ?
X = ?
Y = ?

Output Unchanged = DBR/K/D/e
Scrambled = X/Y/B/P
Special = A = BASL

$FB40 **SETGR** Set graphics mode.

SETGR sets screen for mixed graphics mode and clears the graphics portion of the screen. It then sets the top of the window to line 20 for four lines of text space below the graphics screen.

Input A = ?
X = ?
Y = ?

Output Unchanged = DBR/K/D/e
Scrambled = X/Y/B/P
Special = A = BASL

$FB4B **SETWND** Set text window size.

SETWND sets window to the following:

WNDLFT (address = $20) = $00
WNDWDTH (address = $21) = $28/$50 (40/80 columns)
WNDTOP (address $22) = A on entry
WNDBTM (address $23) = $18

Input A = New WNDTOP
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = Y/B/P
Special = A = BASL

$FB51 **SETWND2** Set text window width and bottom size.

SETWND2 sets window to the following:

WNDWDTH (address = $21) = $28/$50 (40/80 columns)
WNDBTM (address $23) = $18

Input A = ?
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = Y/B/P
Special = A = BASL

$FB5B **TABV** Vertical tab.

TABV stores the value in A in CV (address $25) and then calculates a new base address for storing data to the screen.

Input A = New vertical position (line number)
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = Y/B/P
Special = A = BASL

$FB60 **APPLEII** Clears screen and displays Apple IIGS logo.

APPLEII clears the screen and displays the startup ASCII string 'Apple IIGS' on the first line of the screen.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = A/Y/B/P

$FB6F **SETPWRC** Create power-up byte.

SETPWRC calculates the "funny" complement of the high byte of the RESET vector and stores it at PWREDUP (address $03F4).

Input A = ?
X = ?
Y = ?

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = B/P
Special = A = PWREDUP

$FB78 **VIDWAIT** Check for a pause (Control-S) request.

VIDWAIT checks the keyboard for a Control-S if it is called with an $8D (carriage return) in the accumulator. If a Control-S is found, the system falls through to KBDWAIT. If it is not, control is sent to VIDOUT, where the character is printed and the cursor advanced.

Input A = Output character
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = A/Y/B/P

$FB88 **KBDWAIT** Wait for a keypress.

KBDWAIT waits for a keypress. The keyboard is cleared (unless the keypress is a Control-C), and then control is sent to VIDOUT, where the character is printed and the cursor advanced.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = A/Y/B/P

$FBB3 **VERSION** One of the monitor ROM's main identification bytes.

This is not a callable routine. It is a fixed hex value. The fixed value is $06. This is the identification byte that indicates whether this is an Apple IIe or a later system. This byte is the same in the Apple IIc, the enhanced Apple IIc, the Apple IIe, the enhanced Apple IIe, and the Apple IIGS.

Input No input (not a callable routine)

Output No output (not a callable routine)

$FBBF **ZIDBYTE2** One of the monitor ROM's main identification bytes.

This is not a callable routine. It is a fixed hex value. The fixed value is $00. This is the identification byte that indicates this is an enhanced Apple IIe or a later system.

Input No input (not a callable routine)

Output No output (not a callable routine)

$FBC0 **ZIDBYTE** One of the Monitor ROM's main identification bytes.

This is not a callable routine. It is a fixed hex value. The fixed value is $E0. This is the identification byte that indicates this is an enhanced Apple IIe or a later system.

Input No input (not a callable routine)

Output No output (not a callable routine)

$FBC1 **BASCALC** Text base-address calculator.

BASCALC calculates the base address of the line for the next text character on the 40-column screen. The values calculated are stored at BASL/BASH (addresses $0028/$0029).

Input A = Line number to calculate base for
X = ?
Y = ?

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = B/P
Special = A = BASL

$FBDD **BELL1** Generate user-selected bell tone.

BELL1 generates the user-selected (via the Control Panel) bell tone. There is a delay prior to the tone being generated to prevent rapid calls to BELL1 from causing distorted bell sounds.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = A/B/P
Special = Y = $00

$FBE2 **BELL1.2** Generate user-selected bell tone.

BELL1.2 generates the user-selected (via the Control Panel) bell tone. There is a delay prior to the tone being generated to prevent rapid calls to BELL1.2 from causing distorted bell sounds.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = A/B/P
Special = Y = $00

$FBE4 **BELL2** Generate user-selected bell tone.

BELL2 generates the user-selected (via the Control Panel) bell tone. There is a delay prior to the tone being generated to prevent rapid calls to BELL2 from causing distorted bell sounds.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = A/B/P
Special = Y = $00

$FBF0 **STORADV** Place a printable character on the screen.

STORADV stores the value in the accumulator at the next position in the text buffer (screen location) and advances to the next screen location position.

Input A = Character to display in line
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = A/Y/B/P

$FBF4 **ADVANCE** Increment the cursor position.

ADVANCE advances the cursor by one position. If the cursor is at the window limit, this call issues a carriage return to go to the next line on the screen.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = A/Y/B/P

$FBFD **VIDOUT** Place a character on the screen.

VIDOUT sends printable characters to STORADV. Return, line feed, forward, reverse space, and so on are sent to the vector of appropriate special routines.

Input A = Character to output
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = Y/B/P
Special = A = Output character

$FC10 **BS** Backspace.

BS decrements the cursor one position. If the cursor is at the beginning of the window, the horizontal cursor position is set to the right edge of the window, and the routine goes to the UP subroutine.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = A/Y/B/P

$FC1A **UP** Move up a line.

UP decrements the cursor vertical location by one line unless the cursor is currently on the first line.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = A/B/P

$FC22 **VTAB** Vertical tab.

VTAB loads the value at CV (address $25) into the accumulator and goes to VTABZ.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = B/P
Special = A = BASL
BASL/BASH (addresses $28/$29) = New base address

$FC24 **VTABZ** Vertical tab (alternate entry).

VTABZ uses the value in the accumulator to update the base address used for storing values in the text screen buffer.

Input A = Line to calculate base address for
X = ?
Y = ?

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = B/P
Special = A = BASL
BASL/BASH (addresses $28/$29) = New base address

$FC42 **CLREOP** Clear to end of page.

CLREOP clears the text window from the cursor position to the bottom of the window.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = A/Y/B/P

$FC58 **HOME** Home cursor and clear to end of page.

HOME moves the cursor to the top of screen column 0 and then clears from there to the bottom of the screen window.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = A/Y/B/P

$FC62 **CR** Begin a new line.

CR sets the cursor horizontal position at the left edge of the window and then goes to LF to move to the next line on the screen.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = A/Y/B/P

$FC66 **LF** Line feed.

LF increments the vertical position of the cursor. If the cursor vertical position is not past the bottom line, the base address is updated; otherwise, the routine goes to SCROLL to scroll the screen.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = A/Y/B/P

$FC70 **SCROLL** Scroll the screen up one line.

SCROLL moves all characters up one line within the current text window. The cursor postion is maintained.

Input
A = ?
X = ?
Y = ?

Output
Unchanged = X/DBR/K/D/e
Scrambled = A/Y/B/P

$FC9C **CLREOL** Clear to end of line.

CLREOL clears a text line from the cursor position to the right edge of the window.

Input
A = ?
X = ?
Y = ?

Output
Unchanged = X/DBR/K/D/e
Scrambled = A/Y/B/P

$FC9E **CLREOLZ** Clear to end of line.

CLREOLZ clears from Y on the current line to the right edge of the text window.

Input
A = ?
X = ?
Y = Horizontal position to start clearing from

Output
Unchanged = X/DBR/K/D/e
Scrambled = A/Y/B/P

$FCA8 **WAIT** Delay loop (system-speed independent).

WAIT delays for a specific amount of time and then returns to the program that called it. The length of the delay is specified by the contents of the accumulator. With A the contents of the accumulator, the delay is 1/2(26+27A+5A^2)*14/14.31818 microseconds. WAIT should be used as a minimum delay time, not as the absolute delay time.

Input
A = ?
X = ?
Y = ?

Output
Unchanged = X/Y/DBR/K/D/e
Scrambled = B/P
Special = A = $00

$FCB4 **NXTA4** Increment pointer at A4L/A4H (addresses $42/$43).

NXTA4 increments the 16-bit pointer at A4L/A4H and then goes to NXTA1.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = A/B/P

$FCBA **NXTA1** Compare A1L/A1H (addresses $3C/$3D) with A2L/A2H (addresses $3E/$3F) and then increment A1L/A1H.

NXTA1 performs a 16-bit comparison of A1L/A1H with A2L/A2H and increments the 16-bit pointer A1L/A1H.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = A/B/P

$FCC9 **HEADR** Write a header to cassette tape (obsolete).

HEADR is an obsolete entry point for the Apple IIGS. It does nothing except perform an RTS back to the calling routine.

Input A = ?
X = ?
Y = ?

Output Unchanged = A/X/Y/P/B/DBR/K/D/e

$FDOC **RDKEY** Get an input character and display old inverse flashing cursor.

RDKEY is a character-input subroutine. It places the old Apple II inverse character flashing cursor on the display at the current cursor position and jumps to subroutine $FD10.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = Y/B/P
Special = A = Key pressed (entered character)

$FD10 **FD10** Get an input character and don't display inverse flashing character cursor.

FD10 is a character-input subroutine. It jumps to the subroutine whose address is stored in KSWL/KSWH (addresses $38/$39), usually the standard input subroutine KEYIN, which displays the normal cursor and returns with a character in the accumulator. $FD10 returns only after a key has been pressed or an input character has been placed in the accumulator.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = Y/B/P
Special = A = Key pressed (entered character)

$FD18 **RDKEY1** Get an input character.

RDKEY1 jumps to the subroutine whose address is stored in KSWL/KSWH (addresses $38/$39), usually the standard input subroutine KEYIN, which returns with a character in the accumulator. RDKEY1 returns only after a key has been pressed or an input character has been placed in the accumulator.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = Y/B/P
Special = A = Key pressed (entered character)

$FD1B **KEYIN** Read the keyboard.

KEYIN is a keyboard-input subroutine. It tests the Event Manager to see if it is active. If it is active, KEYIN reads the key pressed from the Event Manager; otherwise, it reads the Apple keyboard directly. In any case, it randomizes the random-number seed RNDL/RNDH (addresses $4E/$4F). When a key is pressed, KEYIN removes the cursor from the display and returns with the keycode in the accumulator.

Input A = Character below cursor
X = ?
Y = ?

Output Unchanged = X/DBR/K/D/e
Scrambled = Y/B/P
Special = A = Key pressed (entered character)

$FD35 **RDCHAR** Get an input character and process escape codes.

RDKEY is a character-input subroutine; it also interprets the standard Apple escape sequences. It places an appropriate cursor on the display at the cursor position and jumps to the subroutine whose address is stored in KSWL/KSWH (addresses $38/$39), usually the standard input subroutine KEYIN, which returns with a character in the accumulator. RDCHAR returns only after a non-e escape-sequence key has been pressed or an input character has been placed in the accumulator.

Input
A = ?
X = ?
Y = ?

Output
Unchanged = X/DBR/K/D/e
Scrambled = Y/B/P
Special = A = Key pressed (entered character)

$FD67 **GETLNZ** Get an input line after issuing a carriage return.

GETLNZ is an alternate entry point for GETLN that sends a carriage return to the standard output and then continues in GETLN. The calling program must call GETLN with the prompt character at PROMPT (address $33).

Input
A = ?
X = ?
Y = ?
PROMPT = (Address $33) = Prompt character

Output
Unchanged = DBR/K/D/e
Scrambled = A/Y/B/P
Special = $200–$2xx contains input line
X = Length of input line

$FD6A **GETLN** Get an input line with a prompt.

GETLN is a standard input subroutine for entire lines of characters. The calling program must call GETLN with the prompt character at PROMPT (address $33).

Input
A = ?
X = ?
Y = ?
PROMPT = (Address $33) = Prompt character

Output
Unchanged = DBR/K/D/e
Scrambled = A/Y/B/P
Special = $200–$2xx contains input line
X = Length of input line

$FD6C **GETLN0** Get an input line with a prompt (alternate entry).

GETLN0 outputs the contents of the accumulator as the prompt. If the user cancels the input line with Control-X or by entering too many backspaces, the contents of PROMPT (address $33) will be issued as the prompt when it gets another line.

Input
A = prompt character
X = ?
Y = ?
PROMPT = (Address $33) = Prompt character

Output
Unchanged = DBR/K/D/e
Scrambled = A/Y/B/P
Special = $200–$2xx contains input line
X = Length of input line

$FD6F **GETLN1** Get an input line with no prompt (alternate entry).

GETLN1 is an alternate entry point for GETLN that does not issue a prompt before it accepts the input line. If the user cancels the input line with Control-X or by entering too many backspaces, the contents of PROMPT (address $33) will be issued as the prompt when it gets another line.

Input
A = ?
X = ?
Y = ?
PROMPT = (Address $33) = Prompt character

Output
Unchanged = DBR/K/D/e
Scrambled = A/Y/B/P
Special = $200–$2xx contains input line
X = Length of input line

$FD8B **CROUT1** Clear to end on line; then issue a carriage return.

CROUT1 clears the current line from the current cursor position to the right edge of the text window. It then goes to CROUT to issue a carriage return.

Input
A = ?
X = ?
Y = ?

Output
Unchanged = X/DBR/K/D/e
Scrambled = Y/B/P
Special = A = $8D (carriage return)

$FD8E **CROUT** Issue a carriage return.

CROUT issues a carriage return to the output device pointed to by CSWL/CSWH (addresses $36/$37).

Input A = ?
X = ?
Y = ?

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = B/P
Special = A = $8D (carriage return)

$FD92 **PRA1** Print a carriage return and A1L/A1H (addresses $3C/$3D).

PRA1 sends a carriage return character ($8D) to the current output device, followed by the contents of the 16-bit pointer A1L/A1H (addresses ($3C/$3D) in hex, followed by a colon (:).

Input A = ?
X = ?
Y = ?

Output Unchanged= DBR/K/D/e
Scrambled = X/B/P
Special = A = $BA (colon)
Y = $00

$FDDA **PRBYTE** Print a hexadecimal byte.

PRBYTE outputs the contents of the accumulator in hexadecimal format to the current output device.

Input A = Hex byte to print
X = ?
Y = ?

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = A/B/P

$FDE3 **PRHEX** Print a hexadecimal digit.

PRHEX outputs the lower nibble of the accumulator as a single hexadecimal digit to the current output device.

Input A = Lower nibble is digit to output
X = ?
Y = ?

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = A/B/P

$FDED **COUT** Output a character.

COUT calls the current output subroutine. The character to output should be in the accumulator. COUT calls the subroutine whose address is stored in CSWL/CSWH (addresses $36/$37), which is usually the standard character-output routine COUT1.

Input A = Character to print
X = ?
Y = ?

Output Unchanged = A/X/Y/DBR/K/D/e
Scrambled = B/P

$FDF0 **COUT1** Output a character to the screen.

COUT1 displays the character in the accumulator on the Apple screen at the current output cursor position and advances the output cursor. It places the character using the settings of the normal/inverse location INVFLG (address $32). It handles the control characters for return ($8D), line feed ($8C), Backspace/Left Arrow ($88), Right Arrow ($95), and bell ($87) and the Change Cursor command (Control-^ = $9E).

Input A = Character to print
X = ?
Y = ?

Output Unchanged = A/X/Y/DBR/K/D/e
Scrambled = B/P

$FDF6 **COUTZ** Output a character to the screen without masking it with the inverse flag.

COUTZ outputs the character in the accumulator without masking it with the inverse flag INVFLG (address $32). Output goes to the screen.

Input A = Character to print
X = ?
Y = ?

Output Unchanged = A/X/Y/DBR/K/D/e
Scrambled = B/P

$FE1F **IDROUTINE** Returns identification information about the system.

IDROUTINE is called with c (carry) set. If it returns with c (carry) clear, then the system is an Apple IIGS or a later system, and the registers A/X/Y contain identification information about the system.

Input A = ?
X = ?
Y = ?

Output Unchanged = DBR/K/D/e
Scrambled = B/P
Special = c (carry) = 0 if Apple IIGS or later. If c = 0, then A/X/Y contain identification information. If c = 1, then A/X/Y are unchanged.

$FE2C **MOVE** Original Monitor Move routine.

MOVE copies the contents of memory from one range of locations to another. This subroutine is *not* the same as the Monitor Move (M) command. The destination address must be in A4L/A4H (addresses $42/$43), the starting source address in A1L/A1H (addresses $3C/$3D), and the ending source address in A2L/A2H (addresses $3E/$3F) when MOVE is called. Y must contain the starting offset into the source/destination buffers.

Input A = ?
X = ?
Y = Starting offset into source/destination buffers (normally $00)
A1L/A1H = (Addresses $3C/$3D) = Start of source buffer
A2L/A2H = (Addresses $3E/$3F) = End of source buffer
A4L/A4H = (Addresses $42/$43) = Start of destination buffer

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = A/B/P
Special = A1L/A1H = (Addresses $3C/$3D) = End of source buffer + 1
A2L/A2H = (Addresses $3E/$3F) = End of source buffer
A4L/A4H = (Addresses $42/$43) = End of destination buffer + 1

$FE5E **"LIST"** Old list entry point (*not* supported under Apple IIGS).

$FE80 **SETINV** Set inverse text mode.

SETINV sets INVFLG (address $32) so that subsequent text output to the screen will appear in inverse mode.

Input
A = ?
X = ?
Y = ?

Output
Unchanged = A/X/DBR/K/D/e
Scrambled = Y/B/P
Special = INVFLG (address $32) = $3F
Y = $3F

$FE84 **SETNORM** Set normal text mode.

SETNORM sets INVFLG (address $32) so that subsequent text output to the screen will appear in normal mode.

Input
A = ?
X = ?
Y = ?

Output
Unchanged = A/X/DBR/K/D/e
Scrambled = Y/B/P
Special = INVFLG (address $32) = $FF
Y = $FF

$FE89 **SETKBD** Reset input to keyboard.

SETKBD resets input hooks KSWL/KSWH (addresses $38/$39) to point to the keyboard.

Input
A = ?
X = ?
Y = ?

Output
Unchanged = DBR/K/D/e
Scrambled = A/X/Y/B/P

$FE8B **INPORT** Reset input to a slot.

INPORT resets input hooks KSWL/KSWH (addresses $38/$39) to point to the ROM space reserved for a peripheral card (or port) in the slot (or port) designated by the value in the accumulator.

Input
A = Slot number to set hooks to
X = ?
Y = ?

Output
Unchanged = DBR/K/D/e
Scrambled = A/X/Y/B/P

$FE93 **SETVID** Reset output to screen.

SETVID resets output hooks CSWL/CSWH (addresses $36/$37) to the screen display routines.

Input
A = ?
X = ?
Y = ?

Output
Unchanged = DBR/K/D/e
Scrambled = A/X/Y/B/P

$FE95 **OUTPORT** Reset output to a slot.

OUTPORT resets output hooks CSWL/CSWH (addresses $36/$37) to point to the ROM space reserved for a peripheral card (or port) in the slot (or port) designated by the value in the accumulator.

Input
A = Slot number to reset hooks to
X = ?
Y = ?

Output
Unchanged = DBR/K/D/e
Scrambled = A/X/Y/B/P

$FEB6 **GO** Original Apple II Go entry point.

GO begins execution of the code pointed to by A1L/A2L (addresses $3C/$3D).

Input
A = ?
X = $01 (required)
Y = ?
A1L/A1H (addresses $3C/$3D) = Start address of program to run
A5H (address $45) = A value to set up before running program
XREG (address $46) = X value to set up before running program
YREG (address $47) = Y value to set up before running program
STATUS (address $48) = P status to set up before running program

Output
Unchanged = DBR/K/D/e
Scrambled = A/X/Y/B/P

$FECD **WRITE** Write a record to cassette tape (obsolete).

WRITE is an obsolete entry point under Apple IIGS. It does nothing except perform an RTS back to the calling routine.

Input A = ?
X = ?
Y = ?

Output Unchanged = A/X/Y/P/BDBR/K/D/e

$FEFD **READ** Read data from a cassette tape (obsolete).

READ is an obsolete entry point under Apple IIGS. It does nothing except perform an RTS back to the calling routine.

Input A = ?
X = ?
Y = ?

Output Unchanged = A/X/Y/P/B/DBR/K/D/e

$FF2D **PRERR** Print ERR on output device.

PRERR sends ERR to the output device and goes to BELL.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = B/P
Special = A = $87 (bell character)

$FF3A **BELL** Send a bell character to the output device.

BELL writes a bell (Control-G) character to the current output device.

Input A = ?
X = ?
Y = ?

Output Unchanged = X/Y/DBR/K/D/e
Scrambled = B/P
Special = A = $87 (bell character)

$FF3F **RESTORE** Restore A/X/Y/P registers.

Restore 6502 register information from locations $45–$48.

Input
A = ?
X = ?
Y = ?
A5H (address $45) = New value for A
XREG (address $46) = New value for X
YREG (address $47) = New value for Y
STATUS (address $48) = New value for P

Output
Unchanged = DBR/K/D/e
Scrambled = B
Special = A = New value
X = New value
Y = New value
P = New value

$FF4A **SAVE** Save A/X/Y/P/S registers and clear decimal mode.

SAVE saves 6502 register information in locations $45–$49 and clears decimal mode.

Input
A = ?
X = ?
Y = ?

Output
Unchanged = Y/DBR/K/D/e
Scrambled = A/X/B/P
Special = A5H (address $45) = Value of A
XREG (address $46) = Value of X
YREG (address $47) = Value of Y
STATUS (address $48) = Value of P
SPNT (address $49) = Value of stack pointer 2
Decimal mode is cleared.

$FF58 **IORTS** Known RTS instruction.

IORTS is used by peripheral cards to determine which slot a card is in. This RTS is fixed and will never be changed.

Input
A = ?
X = ?
Y = ?

Output
Unchanged = A/X/Y/DBR/K/D/e
Scrambled = Nothing

$FF59 **OLDRST** Old Monitor entry point.

OLDRST sets up the video display and keyboard as output and input devices. It sets hex mode, does not beep, and enters the Monitor at MONZ2. It does not return to caller. All Monitor 65C816 register storage locations are reset to standard values.

Input A = ?
X = ?
Y = ?

Output Does not return to caller

$FF65 **MON** Standard Monitor entry point, with beep.

MON clears decimal mode, beeps bell, and enters the Monitor at MONZ. All Monitor 65816 register storage locations are reset to standard values.

Input A = ?
X = ?
Y = ?

Output Does not return to caller

$FF69 **MONZ** Standard Monitor entry point (Call –151).

All Monitor 65816 register storage locations are reset to standard values. MONZ displays the * prompt and sends control to the Monitor input parser.

Input A = ?
X = ?
Y = ?

Output Does not return to caller

$FF6C **MONZ2** Standard Monitor entry point (alternate).

MONZ2 does not change Monitor 65816 register storage locations. MONZ2 displays the * prompt and sends control to the Monitor input parser.

Input A = ?
X = ?
Y = ?

Output Does not return to caller

$FF70 **MONZ4** No prompt Monitor entry point.

MONZ4 does not change Monitor 65816 register storage locations. No prompt is displayed. Control is sent to the Monitor input parser.

Input A = ?
X = ?
Y = ?

Output Does not return to caller

$FF8A **DIG** Shift hex digit into A2l/A2H (addresses $3E/$3F).

DIG shifts an ASCII representation of a hex digit in the accumulator into A2L/A2H (addresses $3E/$3F) and the exits into NXTCHR.

Input A = ASCII character EORed with $B0
X = ?
Y = Entry point in input buffer $2xx at which to continue decoding characters

Output Unchanged = DBR/K/D/e
Scrambled = A/B/P/X
Special = Y = Points to next character in input buffer at $2xx

$FFA7 **GETNUM** Transfer hex input into A2l/A2H (addresses $3E/$3F).

GETNUM scans the input buffer ($2xx) starting at position Y. It shifts hex digits into A2L/A2H (addresses $3E/$3F) until it encounters a nonhex digit. It then exits into NXTCHR.

Input A = ?
X = ?
Y = Entry point in input buffer $2xx at which to start decoding characters

Output Unchanged = DBR/K/D/e
Scrambled = A/B/P/X
Special = Y = Points to next character in input buffer at $2xx

$FFAD **NXTCHR** Translate next character.

NXTCHR is the loop used by GETNUM to parse each character in the input buffer and convert it to a value in A2L/A2H (address $3E/$3F). It also upshifts any lowercase ASCII values that appear in the input buffer (addresses $2xx).

Input A = ?
X = ?
Y = Entry point in input buffer $2xx at which to start decoding characters

Output Unchanged = DBR/K/D/e
Scrambled = A/B/P/X
Special = Y = Points to next character in input buffer at $2xx

$FFBE **TOSUB** Transfer control to a Monitor function.

TOSUB pushes an execution address onto the stack and then performs an RTS to the routine. It is of very limited use to any program.

Input A = ?
X = ?
Y = Offset into subroutine table

Output Unchanged = DBR/K/D/e
Scrambled = A/B/P/X/Y

$FFC7 **ZMODE** Zero out Monitor's mode byte MONMODE (address $31).

ZMODE zeroes out MONMODE (address $31).

Input A = ?
X = ?
Y = ?

Output Unchanged = A/X/DBR/K/D/e
Scrambled = P/B
Special = Y = $00

Appendix D

Vectors

This appendix lists the Apple IIGS vectors. A vector is usually either a 2-byte address in page $00 or (possibly) a 4-byte jump instruction in a different bank of memory. Vectors are used to ensure a common interface point between externally developed programs and system-resident routines. External software jumps directly or indirectly through these vectors instead of attempting to locate and jump directly to the routines themselves. When a new version of the system is released, the vector contents change, thereby maintaining system integrity.

For *all* of the vectors defined in this chapter, the following definitions apply:

- A represents the lower 8 bits of the accumulator.
- B represents the upper 8 bits of the accumulator.
- X and Y represent 8-bit index registers.
- DBR represents the data bank register.
- K represents the program bank register.
- P represents the processor status register.
- S represents the processor stack register.
- D represents the direct-page register.
- e represents the emulation-mode bit.
- c represents the carry flag.
- v represents the overflow flag.
- ? represents a value that is undefined.

Bank $00 page 3 vectors

$03F0–$03F1 **BRKV** User BRK vector.

Address of subroutine that handles BRK interrupts. Normally points to OLDBRK (address $FA59) in Monitor ROM.

$03F2–$03F3 **SOFTEV** User soft-entry vector for RESET.

Address of subroutine that handles warm start (RESET pressed). Normally points to BASIC or operating system.

$03F4 **PWREDUP** EOR of high byte of SOFTEV address.

PWREDUP = SOFTEV + 1 EORed with constant $A5. If PWREDUP does *not* equal SOFTEV + 1 EORed with constant $A5, system performs cold start. If PWREDUP equals SOFTEV + 1 EORed with constant $A5, system performs warm start.

$03F5–$03F6–$3F7 **AMPERV** Applesoft & JMP vector.

Address of subroutine that handles Applesoft & (ampersand) commands. Normally points to IORTS (address $FA58) in Monitor. Address $03F5 contains a JMP ($4C) opcode.

$03F8–$03F9–$3FA **USRADR** User Control-Y and Applesoft. USR function JMP vector.

Address of subroutine that handles user Control-Y and Applesoft USR function commands. Normally points to MON (address $FF65) in Monitor; points to BASIC.SYSTEM warm-start address if ProDOS 8 is loaded. Address $03F8 contains a JMP ($4C) opcode.

$03FB–$03FC–$3FD **NMI** User NMI vector.

Address of subroutine that operating systems or applications can change to gain access to NMI interrupts. Normally points to OLDRST (address $FF59) in Monitor ROM or to operating system if one is loaded. Address $03FB contains a JMP ($4C) opcode.

$03FE–$03FF **IRQLOC** User IRQ vector.

Address of subroutine that operating systems or applications can change to gain access to IRQ interrupts. Normally points to MON (address $FF65) in Monitor ROM or to operating system if one is loaded.

Bank $00 page C3 routines

$C311 **AUXMOVE** Move data blocks between main and auxiliary 48K memory.

AUXMOVE is used by the Apple IIe and Apple IIc to move data blocks between main and auxiliary memory. For compatibility reasons, Apple IIGS also supports this entry point if the 80-column firmware is enabled via the Control Panel.

Input

A = ?
X = ?
Y = ?
c = 1 = Move from main to auxiliary memory
c = 0 = Move from auxiliary to main memory
A1L = (Address $3C); source starting address, low-order byte
A1H = (Address $3D); source starting address, high-order byte
A2L = (Address $3E); source ending address, low-order byte
A2H = (Address $3F); source ending address, high-order byte
A4L = (Address $42); destination starting address, low-order byte
A4H = (Address $43); destination starting address, high-order byte

Output

Unchanged = A/X/Y/DBR/K/D/e
Changed = B/P
A1L/A1H = (Addresses $3C/$3D)=16-bit source ending address +1
A2L/A2H = (Addresses $3E/$3F)=16-bit source ending address
A4L/A4H = (Addresses $42/$43)=16-bit original destination address + number of bytes moved + 1

$C314 **XFER** Transfer program control between main and auxiliary 48K memory.

XFER is used by the Apple IIe and Apple IIc to transfer control between main and auxiliary memory. For compatibility reasons, the Apple IIGS also supports this entry point if the 80-column firmware is enabled via the Control Panel. XFER assumes that the programmer has saved the current stack pointer at $0100 in auxiliary memory and the alternate stack pointer at $0101 in auxiliary memory before calling XFER and restores them after regaining control. Failure to restore these pointers causes program errors and incorrect interrupt handling.

Input
A = ?
X = ?
Y = ?
c = 1 = Transfer control from main to auxiliary memory
c = 0 = Transfer control from auxiliary to main memory
v = 1 = Use page zero and stack in auxiliary memory
v = 0 = Use page zero and stack in main memory
$03ED = Program starting address, low-order byte
$03EE = Program starting address, high-order byte

Output
Unchanged = A/X/Y/DBR/K/D/e
Changed = B/P

Bank $00 page Fx vectors

$FFE4–$FFE5 **NCOP** Native-mode COP vector.

This is not a callable routine. It is a 16-bit value that changes with each ROM release. Its value is not guaranteed. No program should use this value. This vector is pulled from the ROM and used whenever a native-mode COP is executed.

$FFE6–$FFE7 **NBREAK** Native-mode BRK vector.

This is not a callable routine. It is a 16-bit value that changes with each ROM release. Its value is not guaranteed. No program should use this value. This vector is pulled from the ROM and used whenever a native-mode BRK is executed.

$FFE8–$FFE9 **NABORT** Native-mode ABORT vector.

This is not a callable routine. It is a 16-bit value that changes with each ROM release. Its value is not guaranteed. No program should use this value. This vector is pulled from the ROM and used whenever a native-mode ABORT is executed.

$FFEA–$FFEB **NNMI** Native-mode NMI vector.

This is not a callable routine. It is a 16-bit value that changes with each ROM release. Its value is not guaranteed. No program should use this value. This vector is pulled from the ROM and used whenever a native-mode NMI is executed.

$FFEE–$FFEF **NIRQ** Native-mode IRQ vector.

This is not a callable routine. It is a 16-bit value that changes with each ROM release. Its value is not guaranteed. No program should use this value. This vector is pulled from the ROM and used whenever a native-mode IRQ is executed.

$FFF4–$FFF5 **ECOP** Emulation-mode COP vector.

This is not a callable routine. It is a 16-bit value that changes with each ROM release. Its value is not guaranteed. No program should use this value. This vector is pulled from the ROM and used whenever an emulation-mode COP is executed.

$FFF8–$FFF9 **EABORT** Emulation-mode ABORT vector.

This is not a callable routine. It is a 16-bit value that changes with each ROM release. Its value is not guaranteed. No program should use this value. This vector is pulled from the ROM and used whenever an emulation-mode ABORT is executed.

$FFFA–$FFFB **ENMI** Emulation-mode NMI vector.

This is not a callable routine. It is a 16-bit value that changes with each ROM release. Its value is not guaranteed. No program should use this value. This vector is pulled from the ROM and used whenever an emulation-mode NMI is executed.

$FFFC–$FFFD **ERESET** RESET vector.

This is not a callable routine. It is a 16-bit value that changes with each ROM release. Its value is not guaranteed. No program should use this value. This vector is pulled from the ROM and used whenever a RESET is executed.

$FFFE–$FFFF **EBRKIRQ** Emulation-mode BRK/IRQ vector.

This is not a callable routine. It is a 16-bit value that changes with each ROM release. Its value is not guaranteed. No program should use this value. This vector is pulled from the ROM and used whenever an emulation-mode BRK or IRQ is executed.

Bank $E1 vectors

The vectors DISPATCH1 through SYSMGRV are guaranteed to be in the given locations in this and all future Apple IIGS–compatible machines.

$E1/0000–0003 **DISPATCH1** Jump to tool locator entry type 1.

Unconditional jump to tool locator entry type 1. JSL from user's code directly to the tool locator with this entry point. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0004–0007 **DISPATCH2** Jump to tool locator entry type 2.

Unconditional jump to tool locator entry type 2. JSL to a JSL from user's code to the tool locator with this entry point. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0008–000B **UDISPATCH1** Jump to tool locator entry type 1.

Unconditional jump to user-installed tool locator entry type 1. JSL from user's code directly to the user-installed tool locator with this entry point. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/000C–000F **UDISPATCH2** Jump to tool locator entry type 2.

Unconditional jump to user-installed tool locator entry type 2. JSL to a JSL from user's code to the user-installed tool locator with this entry point. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0010–0013 **INTMGRV** Jump to system interrupt manager.

Unconditional jump to the main system interrupt manager. If the application patches out this vector, the application must be able to handle all interrupts in the same fashion as the built-in ROM interrupt manager. Otherwise, the system will not, in most circumstances, run. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0014–0017 **COPMGRV** Jump to COP manager.

Unconditional jump to COP (coprocessor) manager. Currently points to code that causes the Monitor to print a COP instruction disassembly, similar to the BRK disassembly. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0018–001B **ABORTMGRV** Jump to abort manager.

Unconditional jump to abort manager. Currently points to code that causes the Monitor to print the disassembly of the instruction being executed, similar to the BRK disassembly. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/001C–001F **SYSDMGRV** Jump to system failure manager.

Unconditional jump to the system failure manager. This call assumes the following:

- Entry is in 16-bit native mode.
- c (carry) = 0 if user-defined message is pointed to on stack; c = 1 if the default value is used.
- The stack is set up as follows:

  9,S = Error high byte
  8,S = Error low byte
  7,S = Null byte of message address
  6,S = Bank byte of message address
  5,S = High byte of message address
  4,S = Low byte of message address
  3,S = Unused return address
  2,S = Unused return address
  1,S = Unused return address

The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

IRQ.APTALK and IRQ.SERIAL vectors

Vectors IRQ.APTALK and IRQ.SERIAL are normally set up to point to the internal interrupt handler or to code that sets carry and then performs an RTL back to the interrupt manager. All the routines are called in 8-bit native mode and at high speed. The data bank register, the direct register, MSLOT ($7F8), and the stack pointer are not preset or set as for other interrupt vectors. The called routine must return carry clear if the routine handled the interrupt and carry set if it did not handle the interrupt. Carry clear tells the interrupt manager not to call the application or operating system. Carry set tells the interrupt manager that the application or the operating system must be notified of the current interrupt. The called routines must preserve the DBR, speed, 8-bit native mode, D register, stack pointer (or just use current stack), and MSLOT for proper operation. A/X/Y need not be preserved. Interrupts are disabled on entry to all interrupt handlers. The user's interrupt handler must not reenable interrupts from within the handler. AppleTalk and the Desk Manager are allowable exceptions. These vectors should be accessed only via the Miscellaneous Tool Set. Their location in memory is not guaranteed.

$E1/0020–0023 **IRQ.APTALK** Jump to AppleTalk interrupt handler.

Unconditional jump to the AppleTalk LAP (link access protocol) interrupt handler. Handles SCC interrupts intended for AppleTalk. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0024–0027 **IRQ.SERIAL** Jump to serial-port interrupt handler.

Unconditional jump to serial-port interrupt handler. Handles interrupts intended for serial ports. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

IRQ.SCAN through IRQ.OTHER vectors

Vectors IRQ.SCAN through IRQ.OTHER are normally set up to point to the internal interrupt handler or to code that sets carry and then performs an RTL back to the interrupt manager. All the routines are called in 8-bit native mode and with the high speed at data bank register set to $00 and the direct register set to $0000. The called routine must return carry clear if it handled the interrupt and carry set if it did not handle the interrupt. Carry clear tells the interrupt manager not to call the application or operating system. Carry set tells the interrupt manager that the application or the operating system must be notified of the current interrupt. The called routines must preserve the DBR, speed, 8-bit native mode, and D register for proper operation. A/X/Y need not be preserved. Interrupts are disabled on entry to all interrupt handlers. The handler must not reenable interrupts from within the interrupt handler. AppleTalk and the Desk Manager are allowable exceptions. These vectors should be accessed only via the Miscellaneous Tool Set. Their location in memory is not guaranteed.

$E1/0028–002B **IRQ.SCAN** Jump to scan-line interrupt handler.

Unconditional jump to the scan-line interrupt handler. Used by the Cursor Update routine. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/002C–002F **IRQ.SOUND** Jump to sound interrupt handler.

Unconditional jump to the sound interrupt handler. Handles all interrupts from the Ensoniq sound chip. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0030–0033 **IRQ.VBL** Jump to VBL handler.

Unconditional jump to the vertical blanking (VBL) interrupt handler. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0034–0037 **IRQ.MOUSE** Jump to mouse interrrupt handler.

Unconditional jump to the mouse interrupt handler. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0038–003B **IRQ.QTR** Jump to quarter-second interrupt handler.

Unconditional jump to the quarter-second interrupt handler. Used by AppleTalk. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/003C–003F **IRQ.KBD** Jump to keyboard interrupt handler.

Unconditional jump to the keyboard interrupt handler. Currently the keyboard has no hardware interrupt. Keyboard interrupts are still available by making a call to the Miscellaneous Tool Set, telling it to install a heartbeat task that interrupts every time VBL polls the keyboard. If a key is pressed, the heartbeat task will JSL through this vector. This forms a quasi-keyboard interrupt. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0040–0043 **IRQ.RESPONSE** Jump to ADB response interrupt handler.

Unconditional jump to the ADB (Apple DeskTop Bus) response interrupt handler. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0044–0047 **IRQ.SRQ** Jump to SRQ interrupt handler.

Unconditional jump to the ADB (Apple DeskTop Bus) SRQ (service request) interrupt handler. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0048–004B **IRQ.DSKACC** Jump to Desk Manager interrupt handler.

Unconditional jump to the Desk Manager interrupt handler. Invoked by the user pressing Control-Ó-Esc. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/004C–004F **IRQ.FLUSH** Jump to keyboard FLUSH interrupt handler.

Unconditional jump to the keyboard FLUSH interrupt handler. Invoked by the user pressing Control-Ⓒ-Backspace. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0050–0053 **IRQ.MICRO** Jump to keyboard micro abort interrupt handler.

Unconditional jump to the keyboard micro abort recovery routine. This interrupt occurs only when the keyboard micro has a catastrophic failure. If such a failure does occur, the firmware will try to resynchronize up to the keyboard micro and initialize. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0054–0057 **IRQ.1SEC** Jump to 1-second interrupt handler.

Unconditional jump to the 1-second interrupt handler. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0058–005B **IRQ.EXT** Jump to VGC external interrupt handler.

Unconditional jump to the VGC (video graphics chip) external interrupt handler. Currently, the pin that generates this interrupt is forced high so that no interrupt can be generated. This interrupt handler is for future system expansion and currently cannot be used. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/005C–005F **IRQ.OTHER** Jump to other interrupt handler.

Unconditional jump to an installed interrupt handler that handles interrupts other than the ones handled by the internal firmware. This is a general-purpose vector. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0060–0063 **CUPDATE** Cursor Update vector.

Unconditional jump to the Cursor Update routine in QuickDraw II. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0064–0067 **INCBUSYFLG** Increment busy flag vector.

Unconditional jump to the increment busy flag routine. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0068–006B **DECBUSYFLG** Decrement busy flag vector.

Unconditional jump to the decrement busy flag routine. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/006C–006F **BELLVECTOR** Monitor bell vector intercept routine.

Unconditional jump to a user-installed BELL routine. The Monitor calls this routine whenever a BELL character ($87) is output through the output hooks (CSWL/CSWH $36/$37) and whenever BELL1, BELL1.2, and BELL2 are called. The routine is called in 8-bit native mode and must return to the Monitor in 8-bit native mode. The data bank register and direct register must be preserved. Carry must be returned clear, or the Monitor will generate its own bell sound. For compatibility with existing programs, the X register must be preserved during this call, and Y must be = $00 on exit from this call. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0070–0073 **BREAKVECTOR** Break vector.

Unconditional jump to a user-installed break vector. The user's routine is called in 8-bit native mode at high speed, with the data bank register set to $00 and the direct register set to $0000. The user's routine must preserve the data bank register, direct register, and speed and return in 8-bit native mode with an RTL. The user's routine must also clear carry, or the normal break routine pointed to by the vector at $00/03F0.03F1 will be called. If carry comes back clear, the break interrupt is processed and the application program is resumed 2 bytes past the BRK opcode. This vector is set up for use by debuggers such as the Apple IIGS debugger. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0074–0077 **TRACEVECTOR** Trace vector.

Unconditional jump to a trace vector. The user's routine is called in 8-bit native mode at high speed, with the data bank register set to $00 and the direct register set to $0000. The user's routine must preserve the data bank register, direct register, and speed and return in 8-bit native mode with an RTL. If the user's routine clears carry, the Monitor firmware resumes where it left off. If the user sets carry, the Monitor firmware currently will print Trace on the screen and continue where it left off. This vector is set up for use by future system firmware and by current debuggers. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0078–007B **STEPVECTOR** Step vector.

Unconditional jump to a step vector. The user's routine is called in 8-bit native mode at high speed, with the data bank register set to $00 and the direct register set to $0000. The user's routine must preserve the data bank register, direct register, and speed and return in 8-bit native mode with an RTL. If the user clears carry, the Monitor firmware resumes where it left off. If the user's routine sets carry, the Monitor firmware currently will print Step on the screen and continue where it left off. This vector is set up for use by future system firmware and by current debuggers. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/007C–007F Reserved for future expansion.

This vector is reserved for future system expansion and is not available to the user. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

TOWRITEBR through MSGPOINTER vectors

Vectors TOWRITEBR through MSGPOINTER are guaranteed to stay in the same memory locations in all Apple IIGS–compatible systems. These vectors are for convenience and are not to be altered by any application.

$E1/0080–0083 **TOWRITEBR** Write BATTERYRAM routine.

This vector points to a routine that copies the BATTERYRAM buffer in bank $E1 to the clock chip BATTERYRAM with proper checksums. This routine is called by the Miscellaneous Tool Set and by the Control Panel. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0084–0087 **TOREADBR** Read BATTERYRAM routine.

This vector points to a routine that copies the clock chip BATTERYRAM to the BATTERYRAM buffer in bank $E1, compares the checksums, and if the checksums match, returns to the caller. If the checksums do not match or if one of the values in the BATTERYRAM is out of limits, the system default parameters are written into the BATTERYRAM buffer in bank $E1 and then into the clock chip BATTERYRAM with proper checksums. This routine is called by the Miscellaneous Tool Set and by the Control Panel. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0088–008B **TOWRITETIME** Write time routine.

This vector points to a routine that writes to the seconds registers in the clock chip. It transfers the values in the CLKWDATA buffer in bank $E1 to the clock chip. This routine is called by the Miscellaneous Tool Set only. It returns carry clear if the write operation was successful and carry set if it was unsuccessful. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/008C–008F **TOREADTIME** Read time routine.

This vector points to a routine that reads from the seconds registers in the clock chip. It transfers the values to the CLKRDATA buffer in bank $E1 to the clock chip. This routine is called by the Miscellaneous Tool Set only. It returns carry clear if the read operation was successful and carry set if it was unsuccessful. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0090–0093 **TOCTRL.PANEL** Show Control Panel.

This vector points to the Control Panel program. It assumes it was called from the Desk Manager. It uses most of zero page. It RTLs back to the Desk Manager when Quit is chosen. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0094–0097 **TOBRAMSETUP** Set up system to BATTERYRAM parameters routine.

This vector points to a routine that sets up the system parameters to match the values in the BATTERYRAM buffer. In addition, if it is called with carry clear, it sets up the slot configuration (internal versus external). If it is called with carry set, it does *not* set up the slot configuration (internal versus external). BATTERYRAM buffer $E1 values can be set via the Miscellaneous Tool Set only. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/0098–009B **TOPRINTMSG8** Print ASCII string designated by the 8-bit accumulator.

This vector points to a routine that displays ASCII strings pointed to by multiplying the 8-bit accumulator times 2 (shifting it left 1 bit) and then indexing into the address pointer table pointed to by MSGPOINTER (address $E1/00C0; 3-byte pointer). It then uses that address to get the string to display. This routine is used by the built-in Control Panel, by any text-based RAM Control Panel, and by the Monitor (to display messages). The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/009C–009F **TOPRINTMSG16** Print ASCII string designated by the 16-bit accumulator.

This vector points to a routine that displays ASCII strings pointed to by the 16-bit A register. The accumulator is used to index into the address pointer table pointed to by MSGPOINTER (address $E1/00C0; 3-byte pointer). It then uses that address to get the string to display. This routine is used by the built-in Control Panel, by any text-based RAM Control Panel, and by the Monitor (to display messages). The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/00A0–00A3 **CTRLYVECTOR** User Control-Y vector.

Unconditional jump to a user-defined Control-Y vector. The user's routine is called in 8-bit native mode, with the data bank register set to $00 and the direct register set to $0000. The user's routine must preserve the data bank register, direct register, and speed and return in emulation mode with an RTS from bank $00. If no debugger vector is installed, the Monitor firmware will go to the user's routine via the normal Control-Y vector in bank $00 (USRADR 00/03F8.03F9.03FA). This vector is set up to be used by debuggers. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/00A4–00A7 **TOTEXTPG2DA** Point to Alternate Display Mode desk accessory.

This vector points to the Alternate Display Mode program. It assumes it was called from the Desk Manager. It RTLs back to the Desk Manager when a key is pressed. The form of the call in memory is as follows:

JMP abslong ($5C/low byte/high byte/bank byte)

$E1/00A8–00BF **PRO16MLI** ProDOS 16 MLI vectors.

This vector points to the ProDOS 16 routines. Consult ProDOS 16 documents for information about these calls.

$E1/00C0–00C2 **MSGPOINTER** Pointer to all strings used in Control Panel, Alternate Display Mode, and Monitor system messages.

This 3-byte vector points to the address pointer table that points to ASCII strings used by the Control Panel, Alternate Display Mode, and Monitor system messages. It is not useful for users. The form of the call in memory is as follows:

low byte/high byte/bank byte

Appendix E

Soft Switches

This appendix contains a list of the Apple IIGS soft switches—the locations at which various program-definable system control options may be accessed and changed. Note that this listing of soft switches is provided for reference only. You should change the contents of a soft switch only by using the appropriate tool from the toolbox. Refer to the *Apple IIGS Toolbox Reference* for more information.

Important

If you choose to change the contents of any of the soft switches (*not* recommended other than by using the toolbox routines) for any bit that is listed herein as undefined, you should mask that bit. In other words, read the current contents of the data byte, modify only the bits that are defined, and write the contents back to the switch location.

Tables E-1 and E-2 are symbol tables sorted by symbol and address.

```
C000:   C000    20 IOADR           EQU *   ;All I/O is at $Cxxx
C000:   C000    21 KBD             EQU *   ;Bit 7 = 1 if keystroke
                                           ;Bits 6-0 = Key pressed
C000:00         22 CLR80COL        DFB 0   ;Disable 80-column store
C001:00         23 SET80COL        DFB 0   ;Enable 80-column store
C002:00         24 RDMAINRAM       DFB 0   ;Read from main 48K RAM
C003:00         25 RDCARDRAM       DFB 0   ;Read from alternate 48K RAM
C004:00         26 WRMAINRAM       DFB 0   ;Write to main 48K RAM
C005:00         27 WRCARDRAM       DFB 0   ;Write to alternate 48K RAM
C006:00         28 SETSLOTCXROM    DFB 0   ;Use ROM on cards
C007:00         29 SETINTCXROM     DFB 0   ;Use internal ROM
C008:00         30 SETSTDZP        DFB 0   ;Use main zero page/stack
C009:00         31 SETALTZP        DFB 0   ;Use alternate zero page/stack
```

```
C00A:00      32 SETINTC3ROM    DFB 0   ;Enable internal slot 3 ROM
C00B:00      33 SETSLOTC3ROM   DFB 0   ;Enable external slot 3 ROM
C00C:00      34 CLR80VID       DFB 0   ;Disable 80-column hardware
C00D:00      35 SET80VID       DFB 0   ;Enable 80-column hardware
C00E:00      36 CLRALTCHAR     DFB 0   ;Normal LC, flashing UC
C00F:00      37 SETALTCHAR     DFB 0   ;Normal inverse, LC; no flash
C010:00      38 KBDSTRB        DFB 0   ;Turn off keypressed flag
C011:00      39 RDLCBNK2       DFB 0   ;Bit 7 = 1 if LC bank 2 is enabled
C012:00      40 RDLCRAM        DFB 0   ;Bit 7 = 1 if LC RAM read enabled
C013:00      41 RDRAMRD        DFB 0   ;Bit 7 = 1 if reading alternate 48K
C014:00      42 RDRAMWRT       DFB 0   ;Bit 7 = 1 if writing alternate 48K
C015:00      43 RDCXROM        DFB 0   ;Bit 7 = 1 if using internal ROM
C016:00      44 RDALTZP        DFB 0   ;Bit 7 = 1 if slot zp enabled
C017:00      45 RDC3ROM        DFB 0   ;Bit 7 = 1 if slot c3 space enabled
C018:00      46 RD80COL        DFB 0   ;Bit 7 = 1 if 80-column store
C019:00      47 RDVBLBAR       DFB 0   ;Bit 7 = 1 if not VBL
C01A:00      48 RDTEXT         DFB 0   ;Bit 7 = 1 if text (not graphics)
C01B:00      49 RDMIX          DFB 0   ;Bit 7 = 1 if mixed mode on
C01C:00      50 RDPAGE2        DFB 0   ;Bit 7 = 1 if TXTPAGE2 switched in
C01D:00      51 RDHIRES        DFB 0   ;Bit 7 = 1 if HIRES is on
C01E:00      52 ALTCHARSET     DFB 0   ;Bit 7 = 1 if alternate character
                                       set in use
C01F:00      53 RD80VID        DFB 0   ;Bit 7 = 1 if 80-column hardware on
C020:00      54                DFB 0   ;Reserved for future system
                                       expansion
```

```
C021:        56 *___7_____6_____5_____4_____3_____2_____1_____0___
C021:        57 *|       |     |     |     |     |     |     |     |
C021:        58 *|Enable |     |     |     |     |     |     |     |
C021:        59 *|color/ |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
C021:        60 *|mono   |     |     |     |     |     |     |     |
C021:        61 *|_______|_____|_____|_____|_____|_____|_____|_____|
C021:        62 *      ^^^^^  MONOCOLOR status byte ^^^^^

C021:        64 *   MONOCOLOR bits defined as follows:
C021:        65 *   Bit 7 = 0 enables color, 1 disables color
C021:        66 *   Bits 6, 5, 4, 3, 2, 1, 0 must be 0
C021:00      68     MONOCOLOR  DFB 0   ;Monochrome/color selection register
```

```
C022:        70 *___7_____6_____5_____4_____3_____2_____1_____0___
C022:        71 *|       |     |     |     |     |     |     |     |
C022:        72 *|                       |                       |
C022:        73 *|      Text color bits   |  Background color bits |
C022:        74 *|                       |                       |
C022:        75 *|_______|_____|_____|_____|_____|_____|_____|_____|

C022:        76 *      ^^^^^  TBCOLOR byte ^^^^^
```

```
C022:          78 *   TBCOLOR bits defined as follows:
C022:          79 *   Bits 7, 6, 5, 4 = Text color bits
C022:          80 *   Bits 3, 2, 1, 0 = Background color bits
C022:          81 *
C022:          82 *   Color bits =
C022:          83 *   $0 = Black
C022:          84 *   $1 = Deep red
C022:          85 *   $2 = Dark blue
C022:          86 *   $3 = Purple
C022:          87 *   $4 = Dark green
C022:          88 *   $5 = Dark gray
C022:          89 *   $6 = Medium blue
C022:          90 *   $7 = Light blue
C022:          91 *   $8 = Brown
C022:          92 *   $9 = Orange
C022:          93 *   $A = Light gray
C022:          94 *   $B = Pink
C022:          95 *   $C = Green
C022:          96 *   $D = Yellow
C022:          97 *   $E = Aquamarine
C022:          98 *   $F = White
C022:00       100      TBCOLOR    DFB 0   ;Text/background color selection
                                           register
```

```
C023:         102 *___7_______6_____5_______4_____3_______2_____1_____0___
C023:         103 *|       |       |       |       |       |       |       |
C023:         104 *|VGC    |1sec   |Scan   |Ext    |       |1sec   |Scan   |Ext    |
C023:         105 *|int    |int    |int    |int    |  0    |int    |int    |int    |
C023:         106 *|active |active|active|       |       |enable|enable|enable|
C023:         107 *|_______|_____|_____|_____|_____|_____|_____|_____|
C023:         108 *       ^^^^^  VGCINT status byte ^^^^^
```

```
C023:         110 *   VGCINT bits defined as follows:
C023:         111 *   Bit 7 = 1 if interrupt generated by VGC
C023:         112 *   Bit 6 = 1 if 1-second timer interrupt
C023:         113 *   Bit 5 = 1 if scan-line interrupt
C023:         114 *   Bit 4 = 1 if external interrupt (forced low in
                            Apple IIGS)
C023:         115 *   Bit 3 must be 0
C023:         116 *   Bit 2 = 1-second timer interrupt enable
C023:         117 *   Bit 1 = scan-line interrupt enable
C023:         118 *   Bit 0 = ext int enable (can't cause an int in
                            Apple IIGS)
C023:00       120      VGCINT     DFB 0   ;VGC interrupt register
```

```
C024:        122 *___7_____6_____5_____4_____3_____2_____1_____0___
C024:        123 *|      |     |     |     |     |     |     |     |
C024:        124 *|Button|     |                                     |
C024:        125 *|status|Delta|           Delta movement            |
C024:        126 *|now   |sign |                                     |
C024:        127 *|______|_____|_____|_____|_____|_____|_____|_____|
C024:        128 *      ^^^^^  MOUSEDATA byte ^^^^^

C024:        130 *   MOUSEDATA bits defined as follows:
C024:        131 *   Bit 7 = button 1 status if reading X data
C024:        132 *           button 0 status if reading Y data
C024:        133 *   Bit 6 = sign of delta 0 = '+' - 1 = '-'
C024:        134 *   Bits 5, 4, 3, 2, 1, 0 = Delta movement
C024:00      136     MOUSEDATA  DFB 0   ;X or Y mouse data register

C025:        138 *___7_____6_____5_____4_____3_____2_____1_____0___
C025:        139 *|      |     |Update|     |     |     |     |     |
C025:        140 *|Open  |Closed|mod   |Keypad|Repeat|Caps |Ctrl |Shift|
C025:        141 *|Apple |Apple |no key|key   |active|lock |key  |key  |
C025:        142 *|key   |key   |press |active|      |active|active|active|
C025:        143 *|______|______|______|______|______|_____|_____|_____|
C025:        144 *      ^^^^^  KEYMODREG status byte ^^^^^

C025:        146 *   KEYMODREG bits defined as follows:
C025:        147 *   Bit 7 = ⍁ key active
C025:        148 *   Bit 6 =  key active
C025:        149 *   Bit 5 = Updated modifier latch without keypress
C025:        150 *   Bit 4 = Keypad key active
C025:        151 *   Bit 3 = Repeat active
C025:        152 *   Bit 2 = Caps lock active
C025:        153 *   Bit 1 = Control key active
C025:        154 *   Bit 0 = Shift key active
C025:00      156     KEYMODREG  DFB 0   ;Key modifier register

C026:        158 *___7_____6_____5_____4_____3_____2_____1_____0___
C026:        159 *|      |     |     |     |     |     |     |     |
C026:        160 *|                                                 |
C026:        161 *|            Data to/from keyboard micro          |
C026:        162 *|                                                 |
C026:        163 *|______|_____|_____|_____|_____|_____|_____|_____|
C026:        164 *      ^^^^^  DATAREG byte ^^^^^
```

```
C026:          166 *   DATAREG bits defined as follows:
C026:          167 *   Bits 7, 6, 5, 4, 3, 2, 1, 0 = Data to/from keyboard
                                                      micro
C026:          168 *
C026:          169 *   Data at interrupt time in this register defined as
                       follows:
C026:          170 *   Bit 7 = Response byte if set; otherwise, status byte
C026:          171 *   Bit 6 = ABORT valid if set, and all other bits reset
C026:          172 *   Bit 5 = Desktop Manager key sequence pressed
C026:          173 *   Bit 4 = Flush buffer key sequence pressed
C026:          174 *   Bit 3 = SRQ valid if set
C026:          175 *   Bits 2, 1, 0; if all bits clear, then no FDB data
                       valid; otherwise the bits indicate the number of valid
C026:          176 *   bytes received minus 1 (2-8 bytes total)
C026:          177 *
C026:00        179     DATAREG    DFB 0   ;Data register in GLU chip

C027:          181 *___7_______6______5_______4______3_______2______1______0___
C027:          182 *|       |      |       |      |       |      |      |      |
C027:          183 *|Mouse  |Mouse |Data   |Data  |Key    |Key   |Mouse |Cmd   |
C027:          184 *|reg    |int   |reg    |int   |data   |int   |X/Yreg|reg   |
C027:          185 *|full   |enable|full   |enable|full   |enable|data  |full  |
C027:          186 *|_______|______|_______|______|_______|______|______|______|
C027:          187 *       ^^^^^  KMSTATUS byte ^^^^^

C027:          189 *   KMSTATUS bits defined as follows:
C027:          190 *   Bit 7 = 1 if mouse register full
C027:          191 *   Bit 6 = mouse interrupt disable/enable
C027:          192 *   Bit 5 = 1 if data register full
C027:          193 *   Bit 4 = data interrupt enable
C027:          194 *   Bit 3 = 1 if key data full (never use, won't work)
C027:          195 *   Bit 2 = key data interrupt enable (never use, won't
                         work)
C027:          196 *   Bit 1 = 0 = mouse 'X' register data available
C027:          197 *           1 = mouse 'Y' register data available
C027:          198 *   Bit 0 = Command register full
C027:00        200     KMSTATUS   DFB 0   ;Keyboard/mouse status register
C028:00        201     ROMBANK    DFB 0   ;ROM bank select toggle (not used in
                                           Apple IIGS)
```

```
C029:          203 *___7_______6_____5_____4_____3_____2_____1_____0___
C029:          204 *|       |       |       |       |       |       |       |
C029:          205 *|Enable |Linear|B/W   |       |       |       |       |Enable|
C029:          206 *|super  |video |Color|  0    |  0    |  0    |  0    |bank 1|
C029:          207 *|hi-res |       |DHires|       |       |       |       |batch |
C029:          208 *|_______|______|______|______|______|______|______|______|
C029:          209 *       ^^^^^  NEWVIDEO byte ^^^^^

C029:          211 *   NEWVIDEO bits defined as follows:
C029:          212 *   Bit 7 = 1 = Disable Apple IIe video (enables super
                           hi-res)
C029:          213 *   Bit 6 = 1 to linearize for super hi-res
C029:          214 *   Bit 5 = 0 for color double hi-res; 1 for B/W hi-res
C029:          215 *   Bits 4, 3, 2, 1 must be 0
C029:          216 *   Bit 0 = Enable bank 1 latch to allow long instructions
                           to access bank 1 directly; set by Monitor
C029:          217 *           only; a programmer must not change this bit.
C029:00        219     NEWVIDEO   DFB 0   ;Video/enable read alternate mem
                                          with long instructions
C02A:00        220                DFB 0   ;Reserved for future system
                                          expansion

C02B:          222 *___7_______6_____5_____4_____3_____2_____1_____0___
C02B:          223 *|       |       |       |       |       |       |       |
C02B:          224 *| Character Generator | NTSC/|Lang  |       |       |       |
C02B:          225 *|   language select   |  PAL |select|  0    |  0    |  0    |
C02B:          226 *|                     |      |bit   |       |       |       |
C02B:          227 *|_____|_____|_____|_____|_____|_____|_____|_____|
C02B:          228 *       ^^^^^  LANGSEL byte ^^^^^

C02B:          230 *   LANGSEL bits defined as follows:
C02B:          231 *   Bits 7, 6, 5 = Character-generator language selector
C02B:          232 *   Primary language    Secondary language
C02B:          233 *   $0 = English (USA)  Dvorak
C02B:          234 *   $1 = English (UK)   USA
C02B:          235 *   $2 = French         USA
C02B:          236 *   $3 = Danish         USA
C02B:          237 *   $4 = Spanish        USA
C02B:          238 *   $5 = Italian        USA
C02B:          239 *   $6 = German         USA
C02B:          240 *   $7 = Swedish        USA
C02B:          241 *   Bit 4 = 0 if NTSC video mode, 1 if PAL video mode
C02B:          242 *   Bit 3 = LANGUAGE switch bit 0 if primary lang set
                           selected
C02B:          243 *   Bits 2, 1, 0 must be 0
C02B:00        245     LANGSEL    DFB 0   ;Language/PAL/NTSC select register
C02C:00        246     CHARROM    DFB 0   ;Addr for tst mode read of character
                                          ROM
```

```
C02D:        248 *___7_______6_____5_______4_____3_______2_____1_____0___
C02D:        249 *|       |       |       |       |       |       |       |
C02D:        250 *|Slot7  |Slot6 |Slot5 |Slot4 |       |Slot2 |Slot1 |       |
C02D:        251 *|intext |intext|intext|intext|  0    |intext|intext|  0    |
C02D:        252 *|enable |enable|enable|enable|       |enable|enable|       |
C02D:        253 *|_______|______|______|______|______|______|______|______|
C02D:        254 *       ^^^^^  SLTROMSEL byte ^^^^^

C02D:        256 *   SLTROMSEL bits defined as follows:
C02D:        257 *   Bit 7 = 0 enables internal slot 7, 1 enables slot ROM
C02D:        258 *   Bit 6 = 0 enables internal slot 6, 1 enables slot ROM
C02D:        259 *   Bit 5 = 0 enables internal slot 5, 1 enables slot ROM
C02D:        260 *   Bit 4 = 0 enables internal slot 4, 1 enables slot ROM
C02D:        261 *   Bit 3 must be 0
C02D:        262 *   Bit 2 = 0 enables internal slot 2, 1 enables slot ROM
C02D:        263 *   Bit 1 = 0 enables internal slot 1, 1 enables slot ROM
C02D:        264 *   Bit 0 must be 0
C02D:00      266     SLTROMSEL  DFB 0   ;Slot ROM select
C02E:00      267     VERTCNT    DFB 0   ;Addr for read of video cntr bits
                                        V5-VB
C02F:00      268     HORIZCNT   DFB 0   ;Addr for read of video cntr bits
                                        VA-H0
C030:00      269     SPKR       DFB 0   ;Clicks the speaker

C031:        271 *___7_______6_____5_______4_____3_______2_____1_____0___
C031:        272 *|       |       |       |       |       |       |       |
C031:        273 *|3.5"   |3.5"  |       |       |       |       |       |
C031:        274 *|head   |drive |  0    |  0    |  0    |  0    |  0    |  0    |
C031:        275 *|Select |enable|       |       |       |       |       |
C031:        276 *|_______|______|______|______|______|______|______|______|
C031:        277 *       ^^^^^  DISKREG status byte ^^^^^

C031:        279 *   DISKREG bits defined as follows:
C031:        280 *   Bit 7 = 1 to select head on 3.5" drive to use
C031:        281 *   Bit 6 = 1 to enable 3.5" drive
C031:        282 *   Bits 5, 4, 3, 2, 1, 0 must be 0
C031:00      284     DISKREG    DFB 0   ;Used for 3.5" disk drives

C032:        286 *___7_______6_____5_______4_____3_______2_____1_____0___
C032:        287 *|       |       |       |       |       |       |       |
C032:        288 *|       |Clear |Clear |       |       |       |       |
C032:        289 *|  0    |1 sec |scan  |  0    |  0    |  0    |  0    |  0    |
C032:        290 *|       |int   |1n int|       |       |       |       |
C032:        291 *|_______|______|______|______|______|______|______|______|
C032:        292 *       ^^^^^  SCANINT byte ^^^^^
```

```
C032:          294 *   SCANINT bits defined as follows:
C032:          295 *   Bit 7 must be 0
C032:          296 *   Bit 6 = Write 0 here to reset 1-second interrupt
C032:          297 *   Bit 5 = Write 0 here to clear scan-line interrupt
C032:          298 *   Bit 4 must be 0
C032:          299 *   Bit 3 must be 0
C032:          300 *   Bit 2 must be 0
C032:          301 *   Bit 1 must be 0
C032:          302 *   Bit 0 must be 0
C032:00        304     SCANINT    DFB 0   ;Scan-line interrupt register

C033:          306 *___7_______6_______5_______4_______3_______2_______1_______0___
C033:          307 *|       |       |       |       |       |       |       |       |
C033:          308 *|                                                               |
C033:          309 *|                   Clock data register                         |
C033:          310 *|                                                               |
C033:          311 *|_______|_______|_______|_______|_______|_______|_______|_______|
C033:          312 *       ^^^^^  CLOCKDATA byte ^^^^^

C033:          314 *   CLOCKDATA bits defined as follows:
C033:          315 *   Bits 7, 6, 5, 4, 3, 2, 1, 0 = Data passed to/from clock
                                                      chip
C033:00        317     CLOCKDATA  DFB 0   ;Clock data register

C034:          319 *___7_______6_______5_______4_______3_______2_______1_______0___
C034:          320 *|       |       |       |       |       |       |       |       |
C034:          321 *|Clock  |Read/  |Chip   |       |                               |
C034:          322 *|xfer   |Write  |enable|  0    |         Border color          |
C034:          323 *|       |chip   |assert|       |                               |
C034:          324 *|_______|_______|_______|_______|_______|_______|_______|_______|
C034:          325 *       ^^^^^  CLOCKCTL byte ^^^^^

C034:          327 *   CLOCKCTL bits defined as follows:
C034:          328 *   Bit 7 = Set = 1 to start transfer to clock
C034:          329 *   Read = 0 when transfer to clock is complete
C034:          330 *   Bit 6 = 0 = Write to clock chip, 1 = Read from clock
                                                        chip
C034:          331 *   Bit 5 = Clk chip enable asserted after transfer
                           0 = no/1 = yes
C034:          332 *   Bit 4 must be 0
C034:          333 *   Bits 3, 2, 1, 0 = Select border color (see TBCOLOR for
                                          values)
C034:00        335     CLOCKCTL   DFB 0   ;Clock control register
```

```
C035:        337 *___7_____6_____5_____4_____3_____2_____1_____0___
C035:        338 *|      |     |     |     |     |     |     |     |
C035:        339 *|      |Stop |     |Stop |Stop |Stop |Stop |Stop |
C035:        340 *|  0   |I/O/LC|  0  |auxh-r|suprhr|hires2|hires1|txpg |
C035:        341 *|      |shadow|     |shadow|shadow|shadow|shadow|shadow|
C035:        342 *|______|_____|_____|_____|_____|_____|_____|_____|
C035:        343 *       ^^^^^  SHADOW byte ^^^^^

C035:        345 *   SHADOW bits defined as follows:
C035:        346 *   Bit 7 must write 0
C035:        347 *   Bit 6 = 1 to inhibit I/O and language-card operation
C035:        348 *   Bit 5 must write 0
C035:        349 *   Bit 4 = 1 to inhibit shadowing aux hi-res page
C035:        350 *   Bit 3 = 1 to inhibit shadowing 32K video buffer
C035:        351 *   Bit 2 = 1 to inhibit shadowing hi-res page 2
C035:        352 *   Bit 1 = 1 to inhibit shadowing hi-res page 1
C035:        353 *   Bit 0 = 1 to inhibit shadowing text pages
C035:00      355     SHADOW     DFB 0   ;Shadow register

C036:        357 *___7_____6_____5_____4_____3_____2_____1_____0___
C036:        358 *|      |     |     |     |     |     |     |     |
C036:        359 *|Slow/ |     |     |Shadow|Slot 7|Slot 6|Slot 5|Slot 4|
C036:        360 *|fast  |  0  |  0  |in all|motor |motor |motor |motor |
C036:        361 *|speed |     |     |RAM   |detect|detect|detect|detect|
C036:        362 *|______|_____|_____|_____|_____|_____|_____|_____|
C036:        363 *       ^^^^^  CYAREG byte ^^^^^

C036:        365 *   CYAREG bits defined as follows:
C036:        366 *   Bit 7 = 0 = Slow system speed, 1 = Fast system speed
C036:        367 *   Bit 6 must write 0
C036:        368 *   Bit 5 must write 0
C036:        369 *   Bit 4 = Shadow in all RAM banks (never use)
C036:        370 *   Bit 3 = Slot 7 disk motor on detect (set by Monitor
                           only)
C036:        371 *   Bit 2 = Slot 6 disk motor on detect (set by Monitor
                           only)
C036:        372 *   Bit 1 = Slot 5 disk motor on detect (set by Monitor
                           only)
C036:        373 *   Bit 0 = Slot 4 disk motor on detect (set by Monitor
                           only)
C036:00      375     CYAREG      DFB 0   ;Speed and motor on detect
C037:00      376     DMAREG      DFB 0   ;Used during DMA as bank address
C038:00      377     SCCBREG     DFB 0   ;SCC channel B cmd register
C039:00      378     SCCAREG     DFB 0   ;SCC channel A cmd register
C03A:00      379     SCCBDATA    DFB 0   ;SCC channel B data register
C03B:00      380     SCCADATA    DFB 0   ;SCC channel A data register
```

```
C03C:        382 *___7_______6_____5_______4_____3_______2_____1_____0___
C03C:        383 *|       |       |       |       |       |       |       |
C03C:        384 *|Busy   |Auto   |Access|       |                       |
C03C:        385 *|flag   |doc/   |inc   |  0    |          Volume DAC   |
C03C:        386 *|       |RAM    |adrptr|       |                       |
C03C:        387 *|_______|_______|______|_______|_______|_______|_______|
C03C:        388 *       ^^^^^  SOUNDCTL byte ^^^^^

C03C:        390 *   SOUNDCTL bits defined as follows:
C03C:        391 *   Bit 7 = 0 if not busy, 1 if busy
C03C:        392 *   Bit 6 = 0 = Access doc, 1 = Access RAM
C03C:        393 *   Bit 5 = 0 = Disable auto incrementing of address
                             pointer
C03C:        394 *           1 = Enable auto incrementing of address pointer
C03C:        395 *   Bit 4 must be 0
C03C:        396 *   Bits 3, 2, 1, 0 = Volume DAC-$0/$F = Low/full volume
                                       (write only)
C03C:00      398     SOUNDCTL   DFB 0   ;Sound control register

C03D:        400 *___7_______6_____5_______4_____3_______2_____1_____0___
C03D:        401 *|       |       |       |       |       |       |       |
C03D:        402 *|                                                       |
C03D:        403 *|                 Sound data read/written               |
C03D:        404 *|                                                       |
C03D:        405 *|_______|_______|______|_______|_______|_______|_______|
C03D:        406 *       ^^^^^  SOUNDDATA byte ^^^^^

C03D:        408 *   SOUNDDATA bits defined as follows:
C03D:        409 *   Bits 7, 6, 5, 4, 3, 2, 1, 0 = Data read from/written to
                                                    sound RAM
C03D:00      411     SOUNDDATA  DFB 0   ;Sound data register

C03E:        413 *___7_______6_____5_______4_____3_______2_____1_____0___
C03E:        414 *|       |       |       |       |       |       |       |
C03E:        415 *|                                                       |
C03E:        416 *|            Low byte of sound address pointer          |
C03E:        417 *|                                                       |
C03E:        418 *|_______|_______|______|_______|_______|_______|_______|
C03E:        419 *       ^^^^^  SOUNDADRL byte ^^^^^

C03E:        421 *   SOUNDADRL bits defined as follows:
C03E:        422 *   Bits 7, 6, 5, 4, 3, 2, 1, 0 = Address into sound RAM
                                                    low byte
C03E:00      424     SOUNDADRL  DFB  0  ;Sound address pointer, low byte
```

```
C03F:        426 *   7       6       5       4       3       2       1       0
C03F:        427 *|       |       |       |       |       |       |       |       |
C03F:        428 *|                                                               |
C03F:        429 *|           High byte of sound address pointer                  |
C03F:        430 *|                                                               |
C03F:        431 *|_______|_______|_______|_______|_______|_______|_______|_______|
C03F:        432 *       ^^^^^  SOUNDADRH byte ^^^^^

C03F:        434 *   SOUNDADRH bits defined as follows:
C03F:        435 *   Bits 7, 6, 5, 4, 3, 2, 1, 0 = Address into sound RAM
                                                    high byte
C03F:00      437     SOUNDADRH  DFB 0   ;Sound address pointer, high byte
C040:00      438                DFB 0   ;Reserved for future system
                                        expansion
```

❖ *Note:* The Mega II mouse is not used under Apple IIGS as a mouse, but the soft switches and functions are used. Therefore, the programmer may not use the Mega II mouse soft switches.

```
C041:        440 *   7       6       5       4       3       2       1       0
C041:        441 *|       |       |       |       |       |       |       |       |
C041:        442 *|       |       |       |Enable|Enable|Enable|Enable|Enable|
C041:        443 *|  0    |  0    |  0    |1/4sec|VBL   |switch|move  |mouse |
C041:        444 *|       |       |       |ints  |ints  |ints  |ints  |      |
C041:        445 *|_______|_______|_______|______|______|______|______|______|
C041:        446 *       ^^^^^  INTEN byte ^^^^^

C041:        448 *   INTEN bits defined as follows:
C041:        449 *   Bit 7 must be 0
C041:        450 *   Bit 6 must be 0
C041:        451 *   Bit 5 must be 0
C041:        452 *   Bit 4 = 1 to enable quarter-second interrupts
C041:        453 *   Bit 3 = 1 to enable VBL interrupts
C041:        454 *   Bit 2 = 1 to enable Mega II mouse switch interrupts
C041:        455 *   Bit 1 = 1 to enable Mega II mouse movement interrupts
C041:        456 *   Bit 0 = 1 to enable Mega II mouse operation
C041:00      458     INTEN      DFB 0   ;Interrupt-enable register (firmware
                                        use only)
C042:00      459                DFB 0   ;Reserved for future system
                                        expansion
C043:00      460                DFB 0   ;Reserved for future system
                                        expansion
```

```
C044:          462 *   7       6       5       4       3       2       1       0
C044:          463 *|       |       |       |       |       |       |       |       |
C044:          464 *|                                                               |
C044:          465 *|            Mega II Mouse delta movement byte                  |
C044:          466 *|                                                               |
C044:          467 *|_______|_______|_______|_______|_______|_______|_______|_______|
C044:          468 *       ^^^^^  MMDELTAX byte ^^^^^

C044:          470 *   MMDELTAX bits defined as follows:
C044:          471 *   Bits 7, 6, 5, 4, 3, 2, 1, 0 = Delta movement in 2's
                                                      complement notation
C044:00        473     MMDELTAX   DFB 0   ;Mega II mouse delta X register

C045:          475 *   7       6       5       4       3       2       1       0
C045:          476 *|       |       |       |       |       |       |       |       |
C045:          477 *|                                                               |
C045:          478 *|            Mega II Mouse delta movement byte                  |
C045:          479 *|                                                               |
C045:          480 *|_______|_______|_______|_______|_______|_______|_______|_______|
C045:          481 *       ^^^^^  MMDELTAY byte ^^^^^

C045:          483*    MMDELTAY bits defined as follows:
C045:          484 *   Bits 7, 6, 5, 4, 3, 2, 1, 0 = Delta movement in 2's
                                                      complement notation
C045:00        486     MMDELTAY   DFB 0   ;Mega II mouse delta Y register

C046:          488 *   7       6       5       4       3       2       1       0
C046:          489 *|       |       |       |       |       |       |       |       |
C046:          490 *|Self/  |MMouse|Status|Status|Status|Status|Status|Status|
C046:          491 *|burnin |last  |AN3   |1/4sec|VBL   |switch|move  |system|
C046:          492 *|diags  |button|      |int   |int   |int   |int   |      |
C046:          493 *|_______|______|______|______|______|______|______|______|
C046:          494 *       ^^^^^  DIAGTYPE byte ^^^^^

C046:          496 *   DIAGTYPE bits defined as follows:
C046:          497 *   Bit 7 = 0 if self-diagnostics get used if BUTN0 =
                           1/BUTN1 = 1
C046:          498 *   Bit 7 = 1 if burn-in diagnostics get used if BUTN0 =
                           1/BUTN1 = 1
C046:          499 *   Bits 6-0 = Same as INTFLAG
```

```
C046:          501 *___7_____6_____5_____4_____3_____2_____1_____0___
C046:          502 *|      |     |     |     |     |     |     |     |
C046:          503 *|MMouse|MMouse|Status|Status|Status|Status|Status|Status|
C046:          504 *|now   |last  |AN3   |1/4sec|VBL   |switch|move  |system|
C046:          505 *|button|button|      |int   |int   |int   |int   |IRQ   |
C046:          506 *|______|______|______|______|______|______|______|______|
C046:          507 *      ^^^^^  INTFLAG byte ^^^^^

C046:          509 *   INTFLAG bits defined as follows:
C046:          510 *   Bit 7 = 1 if mouse button currently down
C046:          511 *   Bit 6 = 1 if mouse button was down on last read
C046:          512 *   Bit 5 = Status of AN3
C046:          513 *   Bit 4 = 1 if quarter-second interrupted
C046:          514 *   Bit 3 = 1 if VBL interrupted
C046:          515 *   Bit 2 = 1 if Mega II mouse switch interrupted
C046:          516 *   Bit 1 = 1 if Mega II mouse movement interrupted
C046:          517 *   Bit 0 = 1 if system IRQ line is asserted
C046:  C046    519     DIAGTYPE   EQU *   ;0/1 Self/burn-in diagnostics
C046:00        520     INTFLAG    DFB 0   ;Interrupt flag register
C047:00        521     CLRVBLINT  DFB 0   ;Clear the VBL/3.75Hz interrupt
                                          flags
C048:00        522     CLRXYINT   DFB 0   ;Clear Mega II mouse interrupt flags
C049:00        523                DFB 0   ;Reserved for future system
                                          expansion
C04A:00        524                DFB 0   ;Reserved for future system
                                          expansion
C04B:00        525                DFB 0   ;Reserved for future system
                                          expansion
C04C:00        526                DFB 0   ;Reserved for future system
                                          expansion
C04D:00        527                DFB 0   ;Reserved for future system
                                          expansion
C04E:00        528                DFB 0   ;Reserved for future system
                                          expansion
C04F:00        529                DFB 0   ;Reserved for future system
                                          expansion
C050:00        530     TXTCLR     DFB 0   ;Switch in graphics (not text)
C051:00        531     TXTSET     DFB 0   ;Switch in text (not graphics)
C052:00        532     MIXCLR     DFB 0   ;Clear mixed mode
C053:00        533     MIXSET     DFB 0   ;Set mixed mode (4 lines text)
C054:00        534     TXTPAGE1   DFB 0   ;Switch in text page 1
C055:00        535     TXTPAGE2   DFB 0   ;Switch in text page 2
C056:00        536     LORES      DFB 0   ;Low-resolution graphics
C057:00        537     HIRES      DFB 0   ;High-resolution graphics
C058:00        538     SETAN0     DFB 0   ;Clear annunciator 0
```

```
C059:00      539     CLRAN0     DFB 0   ;Set annunciator 0
C05A:00      540     SETAN1     DFB 0   ;Clear annunciator1
C05B:00      541     CLRAN1     DFB 0   ;Set annunciator 1
C05C:00      542     SETAN2     DFB 0   ;Clear annunciator 2
C05D:00      543     CLRAN      DFB 0   ;Set annunciator 2
C05E:00      544     SETAN3     DFB 0   ;Clear annunciator 3
C05F:00      545     CLRAN3     DFB 0   ;Set annunciator 3
C060:00      546     BUTN3      DFB 0   ;Read switch 3
C061:00      547     BUTN0      DFB 0   ;Read switch 0 (Ó key)
C062:00      548     BUTN1      DFB 0   ;Read switch 1 (◆ key)
C063:00      549     BUTN2      DFB 0   ;Read switch 2
C064:00      550     PADDL0     DFB 0   ;Read paddle 0
C065:00      551                DFB 0   ;Read paddle 1
C066:00      552                DFB 0   ;Read paddle 2
C067:00      553                DFB 0   ;Read paddle 3

C068:        555 *___7_____6_____5_____4_____3_____2_____1_____0____
C068:        556 *|      |     |     |     |     |     |     |     |
C068:        557 *|ALTZP |PAGE2|RAMRD|RAMWRT|RDROM|LCBNK2|ROMB |INTCX |
C068:        558 *|status|status|status|status|status|status|status|status|
C068:        559 *|      |     |     |     |     |     |     |     |
C068:        560 *|______|_____|_____|_____|_____|_____|_____|_____|
C068:        561 *      ^^^^^  STATEREG status byte ^^^^^

C068:        563 *   STATEREG bits defined as follows:
C068:        564 *   Bit 7 = ALTZP status
C068:        565 *   Bit 6 = PAGE2 status
C068:        566 *   Bit 5 = RAMRD status
C068:        567 *   Bit 4 = RAMWRT status
C068:        568 *   Bit 3 = RDROM status (read only RAM/ROM (0/1))
C068:        570 *   Important note: Perform two reads to $C083; then change
C068:        571 *   STATEREG to change LCRAM/ROM banks (0/1); keep the
C068:        572 *   language card write enabled.
C068:        573 *
C068:        575 *   Bit 2 = LCBNK2 status 0 = LC bank 0, 1 = LC bank 1
C068:        576 *   Bit 1 = ROMBANK status
C068:        577 *   Bit 0 = INTCXROM status
C068:00      579     STATEREG   DFB 0   ;State register
C069:00      580                DFB 0   ;Reserved for future system
                                        expansion
C06A:00      581                DFB 0   ;Reserved for future system
                                        expansion
```

```
C06B:00     582                 DFB 0   ;Reserved for future system
                                        expansion
C06C:00     583                 DFB 0   ;Reserved for future system
                                        expansion
C06D:00     584     TESTREG     DFB 0   ;Test mode bit register
C06E:00     585     CLRTM       DFB 0   ;Clear test mode
C06F:00     586     ENTM        DFB 0   ;Enable test mode
C070:00     587     PTRIG       DFB 0   ;Trigger the paddles
C071:       588                 DS 15,0 ;ROM interrupt code jump table
C080:00     590                 DFB 0   ;Sel LC RAM bank2 rd, wrt protect LC
                                        RAM
C081:00     591     ROMIN       DFB 0   ;Enable ROM read, 2 reads wrt enb LC
                                        RAM
C082:00     592                 DFB 0   ;Enable ROM read, wrt protect LC RAM
C083:00     593     LCBANK2     DFB 0   ;Sel LC RAM bank2, 2 rds wrt enb LC
                                        RAM
C084:00     595                 DFB 0   ;Sel LC RAM bank2 rd, wrt protect LC
                                        RAM
C085:00     596                 DFB 0   ;Enable ROM read, 2 reads wrt enb LC
                                        RAM
C086:00     597                 DFB 0   ;Enable ROM read, wrt protect LC RAM
C087:00     598                 DFB 0   ;Sel LC RAM bank2, 2 rds wrt enb LC
                                        RAM
C088:00     600                 DFB 0   ;Sel LC RAM bank1 rd, wrt protect LC
                                        RAM
C089:00     601                 DFB 0   ;Enable ROM read, 2 reads wrt enb LC
                                        RAM
C08A:00     602                 DFB 0   ;Enable ROM read, wrt protect LC RAM
C08B:00     603     LCBANK1     DFB 0   ;Sel LC RAM bank1, 2 rds wrt enb LC
                                        RAM
C08C:00     605                 DFB 0   ;Sel LC RAM bank1 rd, wrt protect LC
                                        RAM
C08D:00     606                 DFB 0   ;Enable ROM read, 2 reads wrt enb LC
                                        RAM
C08E:00     607                 DFB 0   ;Enable ROM read, wrt protect LC RAM
C08F:00     608                 DFB 0   ;Sel LC RAM bank1, 2 rds wrt enb LC
                                        RAM

0000:610    DEND

0000:612    CLRROM  EQU         $CFFF   ;Switch out $C8 ROMs
```

Table E-1
Symbol table sorted by symbol

C01E ALTCHARSET	C061 BUTN0	C062 BUTN1	C063 BUTN2
C060 BUTN3	C02C CHARROM	C034 CLOCKCTL	C033 CLOCKDATA
C000 CLR80COL	C00C CLR80VID	C00E CLRALTCHAR	C059 CLRAN0
C05B CLRAN1	C05D CLRAN2	C05F CLRAN3	CFFF CLRROM
C06E CLRTM	C047 CLRVBLINT	C048 CLRXYINT	C036 CYAREG
C026 DATAREG	C046 DIAGTYPE	C031 DISKREG	C037 DMAREG
C06F ENTM	C057 HIRES	C02F HORIZCNT	C041 INTEN
C046 INTFLAG	C000 IOADR	C010 KBDSTRB	C000 KBD
C025 KEYMODREG	C027 KMSTATUS	C02B LANGSEL	C08B LCBANK1
C083 LCBANK2	C056 LORES	C052 MIXCLR	C053 MIXSET
C044 MMDELTAX	C045 MMDELTAY	C021 MONOCOLOR	C024 MOUSEDATA
C029 NEWVIDEO	C064 PADDL0	C070 PTRIG	C018 RD80COL
C01F RD80VID	C016 RDALTZP	C017 RDC3ROM	C003 RDCARDRAM
C015 RDCXROM	C01D RDHIRES	C011 RDLCBNK2	C012 RDLCRAM
C002 RDMAINRAM	C01B RDMIX	C01C RDPAGE2	C013 RDRAMRD
C014 RDRAMWRT	C01A RDTEXT	C019 RDVBLBAR	C028 ROMBANK
C081 ROMIN	C032 SCANINT	C03B SCCADATA	C039 SCCAREG
C03A SCCBDATA	C038 SCCBREG	C001 SET80COL	C00D SET80VID
C00F SETALTCHAR	C009 SETALTZP	C058 SETAN0	C05A SETAN1
C05C SETAN2	C05E SETAN3	C00A SETINTC3ROM	C007 SETINTCXROM
C00B SETSLOTC3ROM	C006 SETSLOTCXROM	C008 SETSTDZP	C035 SHADOW
C02D SLTROMSEL	C03F SOUNDADRH	C03E SOUNDADRL	C03C SOUNDCTL
C03D SOUNDDATA	C030 SPKR	C068 STATEREG	C022 TBCOLOR C06
C06D TESTREG	C050 TXTCLR	C054 TXTPAGE1	C055 TXTPAGE2
C051 TXTSET	C02E VERTCNT	C023 VGCINT	C005 WRCARDRAM
C004 WRMAINRAM			

Table E-2
Symbol table sorted by address

C000	IOADR	C000	KBD	C000	CLR80COL	C001	SET80COL
C002	RDMAINRAM	C003	RDCARDRAM	C004	WRMAINRAM	C005	WRCARDRAM
C006	SETSLOTCXROM	C007	SETINTCXROM	C008	SETSTDZP	C009	SETALTZP
C00A	SETINTC3ROM	C00B	SETSLOTC3ROM	C00C	CLR80VID	C00D	SET80VID
C00E	CLRALTCHAR	C00F	SETALTCHAR	C010	KBDSTRB	C011	RDLCBNK2
C012	RDLCRAM	C013	RDRAMRD	C014	RDRAMWRT	C015	RDCXROM
C016	RDALTZP	C017	RDC3ROM	C018	RD80COL	C019	RDVBLBAR
C01A	RDTEXT	C01B	RDMIX	C01C	RDPAGE2	C01D	RDHIRES
C01E	ALTCHARSET	C01F	RD80VID	C021	MONOCOLOR	C022	TBCOLOR
C023	VGCINT	C024	MOUSEDATA	C025	KEYMODREG	C026	DATAREG
C027	KMSTATUS	C028	ROMBANK	C029	NEWVIDEO	C02B	LANGSEL
C02C	CHARROM	C02D	SLTROMSEL	C02E	VERTCNT	C02F	HORIZCNT
C030	SPKR	C031	DISKREG	C032	SCANINT	C033	CLOCKDATA
C034	CLOCKCTL	C035	SHADOW	C036	CYAREG	C037	DMAREG
C038	SCCBREG	C039	SCCAREG	C03A	SCCBDATA	C03B	SCCADATA
C03C	SOUNDCTL	C03D	SOUNDDATA	C03E	SOUNDADRI	C03F	SOUNDADRH
C041	INTEN	C044	MMDELTAX	C045	MMDELTAY	C046	DIAGTYPE
C046	INTFLAG	C047	CLRVBLINT	C048	CLRXYINT	C050	TXTCLR
C051	TXTSET	C052	MIXCLR	C053	MIXSET	C054	TXTPAGE1
C055	TXTPAGE2	C056	LORES	C057	HIRES	C058	SETAN0
C059	CLRAN0	C05A	SETAN1	C05B	CLRAN1	C05C	SETAN2
C05D	CLRAN2	C05E	SETAN3	C05F	CLRAN3	C060	BUTN3
C061	BUTN0	C062	BUTN1	C063	BUTN2	C064	PADDL0
C068	STATEREG	C06D	TESTREG	C06E	CLRTM	C06F	ENTM
C070	PTRIG	C081	ROMIN	C083	LCBANK2	C08B	LCBANK1
CFFF	CLRROM						

Appendix F

Disassembler/ Mini-Assembler Opcodes

This appendix lists all of the 65C816 instructions and the instruction formats that the disassembler uses to define the contents of the disassembly. You may wish to hand-assemble various short routines. This listing provides you with a ready reference for the 65C816 instructions and addressing modes. Sometimes as the table begins a new alphabetic item in the name field, a line break is inserted for readability. For cases where the instructions are closely related to each other (such as branch instructions, push instructions, and pull instructions), the line break is omitted.

In the table that follows, the addressing modes of the processor are abbreviated as shown on the following page.

Abbreviation for addressing mode	Actual addressing mode represented
#	Immediate
(a)	Absolute indirect
(a,x)	Absolute indexed indirect
(d)	Direct indirect
(d),y	Direct indirect indexed
(d,x)	Direct indexed indirect
(r,s),y	Stack relative indirect indexed
a	Absolute
a,x	Absolute indexed (with x)
a,y	Absolute indexed (with y)
Acc	Accumulator
al	Absolute long
al,x	Absolute indexed long
d	Direct
d,x	Direct indexed (with x)
d,y	Direct indexed (with y)
i	Implied
r	Program counter relative
r,s	Stack relative
rl	Program counter relative long
s	Stack
xya	Block move
[d]	Direct indirect long
[d],y	Direct indirect indexed long

Name	Mode	Bytes	Opcode number
ADC	(d)	2	72
ADC	(d),y	2	71
ADC	(d,x)	2	61
ADC	(r,s),y	2	73
ADC	d	2	65
ADC	d,x	2	75
ADC	r,s	2	63
ADC	[d]	2	67
ADC	[d],y	2	77
ADC	#	2 (3)	69
ADC	a	3	6D
ADC	a,x	3	7D
ADC	a,y	3	79
ADC	al	4	6F
ADC	al,x	4	7F
AND	(d)	2	32
AND	(d),y	2	31
AND	(d,x)	2	21
AND	(r,s),y	2	33
AND	d	2	25
AND	d,x	2	35
AND	r,s	2	23
AND	[d]	2	27
AND	[d],y	2	37
AND	#	2 (3)	29
AND	a	3	2D
AND	a,x	3	3D
AND	a,y	3	39
AND	al	4	2F
AND	al,x	4	3F
ASL	Acc	1	0A
ASL	d	2	06
ASL	d,x	2	16
ASL	a	3	0E
ASL	a,x	3	1E
BCC	r	2	90
BCS	r	2	B0
BEQ	r	2	F0

Name	Mode	Bytes	Opcode number
BIT	d	2	24
BIT	d,x	2	34
BIT	#	2 (3)	89
BIT	a	3	2C
BIT	a,x	3	3C
BMI	r	2	30
BNE	r	2	D0
BPL	r	2	10
BRA	r	2	80
BRK	i	2	00
BRL	rl	3	82
BVC	r	2	50
BVS	r	2	70
CLC	i	1	18
CLD	i	1	D8
CLI	i	1	58
CLV	i	1	B8
CMP	(d)	2	D2
CMP	(d),y	2	D1
CMP	(d,x)	2	C1
CMP	(r,s),y	2	D3
CMP	d	2	C5
CMP	d,x	2	D5
CMP	r,s	2	C3
CMP	[d]	2	C7
CMP	[d],y	2	D7
CMP	#	2 (3)	C9
CMP	a	3	CD
CMP	a,x	3	DD
CMP	a,y	3	D9
CMP	al	4	CF
CMP	al,x	4	DF
COP	i	2	02
CPX	d	2	E4
CPX	#	2 (3)	E0
CPX	a	3	EC

Name	Mode	Bytes	Opcode number
CPY	d	2	C4
CPY	#	2 (3)	C0
CPY	a	3	CC
DEC	Acc	1	3A
DEC	d	2	C6
DEC	d,x	2	D6
DEC	a	3	CE
DEC	a,x	3	DE
DEX	i	1	CA
DEY	i	1	88
EOR	(d)	2	52
EOR	(d),y	2	51
EOR	(d,x)	2	41
EOR	(r,s),y	2	53
EOR	d	2	45
EOR	d,x	2	55
EOR	r,s	2	43
EOR	[d]	2	47
EOR	[d],y	2	57
EOR	#	2 (3)	49
EOR	a	3	4D
EOR	a,x	3	5D
EOR	a,y	3	59
EOR	al	4	4F
EOR	al,x	4	5F
INC	Acc	1	1A
INC	d	2	E6
INC	d,x	2	F6
INC	a	3	EE
INC	a,x	3	FE
INX	i	1	E8
INY	i	1	C8
JML	(a)	3	DC
JMP	(a)	3	6C
JMP	(a,x)	3	7C
JMP	a	3	4C
JMP	al	4	5C
JSL	al	4	22
JSR	(a,x)	3	FC
JSR	a	3	20
LDA	(d)	2	B2
LDA	(d),y	2	B1
LDA	(d,x)	2	A1
LDA	(r,s),y	2	B3
LDA	d	2	A5
LDA	d,x	2	B5
LDA	r,s	2	A3
LDA	[d]	2	A7
LDA	[d],y	2	B7
LDA	#	2 (3)	A9
LDA	a	3	AD
LDA	a,x	3	BD
LDA	a,y	3	B9
LDA	al	4	AF
LDA	al,x	4	BF
LDX	d	2	A6
LDX	d,y	2	B6
LDX	#	2 (3)	A2
LDX	a	3	AE
LDX	a,y	3	BE
LDY	d	2	A4
LDY	d,x	2	B4
LDY	#	2 (3)	A0
LDY	a	3	AC
LDY	a,x	3	BC
LSR	Acc	1	4A
LSR	d	2	46
LSR	d,x	2	56
LSR	a	3	4E
LSR	a,x	3	5E
MVN	xya	3	54
MVP	xya	3	44
NOP	i	1	EA

Name	Mode	Bytes	Opcode number
ORA	(d)	2	12
ORA	(d),y	2	11
ORA	(d,x)	2	01
ORA	(r,s),y	2	13
ORA	d	2	05
ORA	d,x	2	15
ORA	r,s	2	03
ORA	[d]	2	07
ORA	[d],y	2	17
ORA	#	2 (3)	09
ORA	a	3	0D
ORA	a,x	3	1D
ORA	a,y	3	19
ORA	al	4	0F
ORA	al,x	4	1F
PEA	s	3	F4
PEI	s	2	D4
PER	s	3	62
PHA	s	1	48
PHB	s	1	8B
PHD	s	1	0B
PHK	s	1	4B
PHP	s	1	08
PHX	s	1	DA
PHY	s	1	5A
PLA	s	1	68
PLB	s	1	AB
PLD	s	1	2B
PLP	s	1	28
PLX	s	1	FA
PLY	s	1	7A
REP	#	2	C2
ROL	Acc	1	2A
ROL	d	2	26
ROL	d,x	2	36
ROL	a	3	2E
ROL	a,x	3	3E

Name	Mode	Bytes	Opcode number
ROR	Acc	1	6A
ROR	d	2	66
ROR	d,x	2	76
ROR	a	3	6E
ROR	a,x	3	7E
RTI	s	1	40
RTL	s	1	6B
RTS	s	1	60
SBC	(d)	2	F2
SBC	(d),y	2	F2
SBC	(d,x)	2	E1
SBC	(r,s),y	2	F3
SBC	d	2	E5
SBC	d,x	2	F5
SBC	r,s	2	E3
SBC	[d]	2	E7
SBC	[d],y	2	F7
SBC	#	2 (3)	E9
SBC	a	3	ED
SBC	a,x	3	FD
SBC	a,y	3	F9
SBC	al	4	EF
SBC	al,x	4	FF
SEC	i	1	38
SED	i	1	F8
SEI	i	1	78
SEP	#	2	E2
STA	(d)	2	92
STA	(d),y	2	91
STA	(d,x)	2	81
STA	(r,s),y	2	93
STA	d	2	85
STA	d,x	2	95
STA	r,s	2	83
STA	[d]	2	87
STA	[d],y	2	97
STA	a	3	8D
STA	a,x	3	9D
STA	a,y	3	99
STA	al	4	8F
STA	al,x	4	9F
STP	i	1	DB

Name	Mode	Bytes	Opcode number
STX	d	2	86
STX	d,y	2	96
STX	a	3	8E
STY	d	2	84
STY	d,x	2	94
STY	a	3	8C
STZ	d	2	64
STZ	d,x	2	74
STZ	a	3	9C
STZ	a,x	3	9E
TAX	i	1	AA
TAY	i	1	A8
TCD	i	1	5B
TCS	i	1	1B
TDC	i	1	7B

Appendix G

The Control Panel

The Control Panel firmware allows you to experiment with different system configurations and change the system time. You can also permanently store any changes in the battery-powered RAM (called *Battery RAM*). The Battery RAM is a Macintosh clock chip that has 256 bytes of battery-powered RAM for system-parameter storage.

The Control Panel program is a ROM-resident hardware configuration program. It is invoked when the system is powered up if you press the Option key. An alternate means of invoking the Control Panel is to perform a cold start by pressing Control and the Option key at the same time and then Reset. The Desk Manager can also call the Control Panel and affect the values specified in this appendix.

Control Panel parameters

The following are the selections and options available for each Control Panel menu. A checkmark (√) indicates the default value for each option.

Printer port

Sets up all related functions for the printer port (slot 1). Options are as follows:

Option	Choices
Device connect	√ Printer Modem
Line length	√ Unlimited 40 72 80 132
Delete first LF after CR	√ No Yes
Add LF after CR	√ Yes No
Echo	√ No Yes
Buffering	√ No Yes
Baud	50 75 110 134.5 150 300 600 1200 1800 2400 3600 4800 7200 √ 9600 19,200
Data bits	√ 8 7 6 5
Stop bits	√ 2 1
Parity	Odd Even √ None
DCD handshake	√ Yes No
DSR/DTR handshake	√ Yes No
XON/XOFF handshake	Yes √ No

Modem port

Sets up all related functions for the modem port (slot 2). Options are as follows:

Option	Choices
Device connected	√ Modem Printer
Line length	√ Unlimited 40 72 80 132
Delete first LF after CR	√ No Yes
Add LF after CR	√ Yes No
Echo	√ No Yes
Buffering	√ No Yes
Baud	50 75 110 134.5 150 300 600 √ 1200 1800 2400 3600 4800 7200 19,200
Data bits	√ 8 7 6 5
Stop bits	√ 2 1
Parity	Odd Even √ None
DCD handshake	No √ Yes
DSR/DTR handshake	√ Yes No
XON/XOFF handshake	Yes √ No

Display

Selects all video-specific options. Choosing Type automatically causes color or monochrome selections to appear on the rest of the screen. Options are as follows:

Line option	Choices
Type	√ Color
	Mono
Columns	√ 40
	80
Hertz	√ 60
	50

Color/ monochrome options	Choices	
Text color	(*Color name is displayed.*)	
	Black	Orange
	Dark blue	Light gray
	Purple	Pink
	Dark green	Light green
	Dark gray	Yellow
	Medium blue	Aquamarine
	Light blue	√ White
	Brown	
Text background	(*Color name is displayed.*)	
	Black	Brown
	Deep red	Orange
	Dark blue	Light gray
	Purple	Pink
	Dark green	Light green
	Dark gray	Yellow
	√ Medium blue	Aquamarine
	Light blue	White

Color/ monochrome options	Choices	
Border color	(*Color name is displayed.*)	
	Black	Brown
	Deep red	Orange
	Dark blue	Light gray
	Purple	Pink
	Dark green	Light green
	Dark gray	Yellow
	√ Medium blue	Aquamarine
	Light blue	White
Standard colors	No	
	√ Yes	

The *Standard colors* option indicates whether your chosen colors match the Apple standard values. If you select Yes, the current colors are switched to Apple standard colors.

Sound

Allows system volume and pitch to be altered via an indicator bar. The default value is in the middle of each range.

Speed

Allows default system speed of either normal speed (1 MHz) or fast speeds (2.6/2.8 RAM/ROM MHz). Available options are as follows:

Option	Choices
System speed	√ Fast
	Normal

RAM disk

Allows default amount of free RAM to be used for RAM disk. Options are as follows:

Minimum free RAM for RAM disk: (minimum)
Maximum free RAM for RAM disk: (maximum)

Graduations between minimum and maximum are determined by adding or subtracting 32K from the RAM size that is displayed. Limited to zero or the largest selectable size. Default RAM disk size is 0 bytes minimum, 0 bytes maximum. RAM disk size ranges from 0 bytes to largest selectable RAM disk size.

The amount of free RAM (in kilobytes) for the RAM disk is displayed on the screen in the format xxxxxK. Free RAM equals the total system RAM minus 256K.

The current RAM disk size is also displayed on the screen. The current RAM disk size can be determined by one of the RAM disk driver commands.

The following message will be displayed on the screen:

```
Total RAM in use: xxxxxK
```

Total RAM in use equals total system RAM minus total free RAM.

The total free RAM disk space will be displayed on the screen. You can determine the amount of total free RAM by calling the Memory Manager.

Slots

Allows you to select either built-in device or peripheral card for slots 1, 2, 3, 4, 5, 6, and 7. Also allows you to select startup slot or to scan slots at startup time. Options available are as follows:

Option	Choices
Slot 1	√ Printer port Your card
Slot 2	√ Modem port Your card
Slot 3	√ Built-in text display Your card
Slot 4	√ Mouse port Your card
Slot 5	√ SmartPort Your card
Slot 6	√ Disk port Your card
Slot 7	Built-in AppleTalk √ Your card
Startup slot	√ Scan 1 2 3 4 5 6 7 RAM disk ROM disk

Options

Allows you to select the keyboard layout, text display language, key repeat speed, and delay to key repeat to use advanced features. Layouts and languages are displayed that correspond to the hardware. Layouts and languages not available with your hardware (keyboard micro and Mega II) are not displayed. The information about the layouts and languages that are available comes from the keyboard micro at power-up time. Options are as follows:

Option	Choices
Display language	Chosen from Table G-1
Keyboard layout	Chosen from Table G-1
Keyboard buffering	√ No Yes
Repeat speed	4 char/sec 8 char/sec 11 char/sec 15 char/sec √ 20 char/sec 24 char/sec 30 char/sec 40 char/sec
Repeat delay	.25 sec .50 sec √ .75 sec 1.00 sec No repeat

Option	Choices
Double-click time	1 tick = 1/60 sec 50 ticks (slow) 40 ticks √ 30 ticks 20 ticks 10 ticks (fast)
Cursor flash rate	1 tick = 1/60 sec 0 ticks (no flash) 60 ticks √ 30 ticks 15 ticks 10 ticks (fast)

Advanced features

Shift caps/ lowercase	√ No Yes
Fast space/ delete keys	√ No Yes
Dual speed keys	√ Normal Fast
High-speed mouse	√ No Yes

Table G-1
Language options

Number	ASCII	Number	ASCII
0	English (U.S.A.)	10	Finnish
1	English (U.K.)	11	Portuguese
2	French	12	Tamil
3	Danish	13	Hindi
4	Spanish	14	T1
5	Italian	15	T2
6	German	16	T3
7	Swedish	17	T4
8	Dvorak	18	T5
9	French Canadian	19	T6
A	Flemish	1A	L1
B	Hebrew	1B	L2
C	Japanese	1C	L3
D	Arabic	1D	L4
E	Greek	1E	L5
F	Turkish	1F	L6

For the language options, items 0–7 are available to control the display language. Items 8 and 9 control the keyboard layout.

(The keyboard microprocessor provides the pointer for the appropriate ASCII value listed in Table G-1.)

Clock

Allows you to set the time and date and time/date formats. Options are as follows:

Option	Choices	Option	Choices
Month	1–12	Hour	1–12 or 0–23 (depends on Format selected)
Day	1–31	Minute	0–59
Year	1904–2044	Second	0–59
Format √	MM/DD/YY DD/MM/YY YY/MM/DD	Format √	AM–PM 24-hour

Quit

Returns to calling application or, if called from keyboard, performs a startup function.

Battery-powered RAM

The Battery RAM is a Macintosh clock chip that has 256 bytes of battery-powered RAM used for system-parameter storage. The AppleTalk node number is stored in the Battery RAM, set by the AppleTalk firmware.

❖ *Note:* The Battery RAM is not for application program use.

The Battery RAM must include encoded bytes for all options that can be selected from the Control Panel. Standard setup values are placed into Battery RAM during manufacturing. However, the keyboard layout and display language are determined by the keyboard used.

Items that can be changed by manufacturing and the Control Panel program can also be changed by your application program; however, only the Miscellaneous Tool Set Battery RAM routines or another Apple-approved utility program can make changes to Battery RAM. If the changing program is not an Apple-approved utility, Battery RAM will be severely damaged and the system will become inoperative. If Battery RAM is damaged and inoperative (or the battery dies), the firmware will automatically use the Apple standard values to bring up the system. The battery can be replaced, and you can enter the Control Panel program to restore the system to its prior configuration.

Control Panel at power-up

At power-up, the Battery RAM is checksummed. If the Battery RAM fails its checksum test, the system assumes a U.S. keyboard configuration and English language. Further, U.S. standard parameters are checksummed and moved to the Battery RAM storage buffer in bank $E1. The system continues running using U.S. standard parameters.

Appendix H

Banks $E0 and $E1

A special section of Apple IIGS memory is dedicated to the Mega II chip. The Mega II, also called the Apple-II-on-a-chip, is a separate coprocessor that runs at 1 MHz and provides the display that the Apple IIGS produces on the video screen.

To communicate with the Mega II, the Apple IIGS either writes directly into bank $E0 or $E1 or enables a special soft switch, named *shadowing*. When shadowing is enabled, whenever the Apple IIGS writes into bank $00 (or bank $01), the system automatically synchronizes with the Mega II and writes the same data into bank $E0 (or bank $E1).

Figure H-1 depicts the layout of the memory in these banks of memory. Some of this memory is dedicated to display areas, some of it is reserved for firmware use, and some of it is declared as free space and is managed by the Memory Manager.

Figure H-1 shows the location of the various functions of Apple IIGS banks $E0 and $E1. In the figure, the notation *K* means a decimal value of 1024 bytes, and the notation *page* means hex $100 bytes.

❖ *Note:* In Figure H-1, the memory segments called *free space* are available through the Memory Manager only.

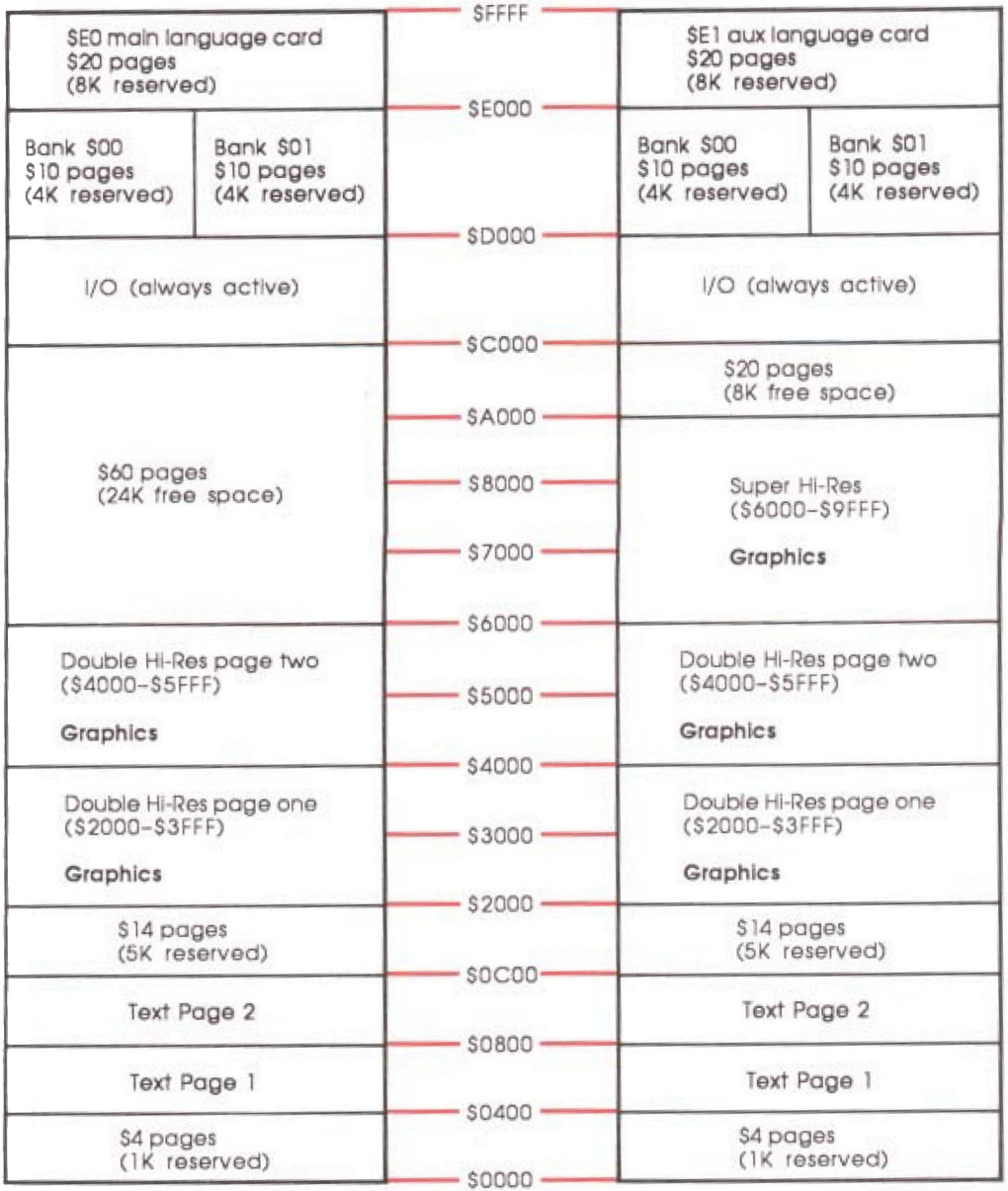


Figure H-1
Memory map of banks $E0 and $E1

Using banks $E0 and $E1

You can use graphics memory located in memory banks $E0 and $E1 or the free space via the Memory Manager; however, you must exercise caution to ensure that you don't use areas that are reserved for machine use.

Free space

Eighty hexadecimal pages, or 32K bytes, in the area labeled *free space* can be used; however, this area must be accessed through the Memory Manager. (The Memory Manager can be called through the Apple IIGS Toolbox.) If you try to use this space without first calling the Memory Manager, you will cause a system failure.

Video buffers not needed for screen display may be used for your applications.

❖ *Note:* Video buffers are used by firmware only for video displays because there is no way to determine which video modes are needed by your applications.

Language-card area

The language-card area is switched by the same soft switches used to switch Apple II simulation language cards in banks $00 and $01. Before switching language-card banks (or ROM for RAM or RAM for ROM), the current configuration must be saved. The configuration must be restored after your subroutine is finished accessing the switched area.

Shadowing

The shadowing ability of the Apple IIGS can be used by applications to display overlay data on the screen. Normally, if an application wants to display an overlay on an existing screen, it must save the data in the area that is overwritten. Because of the shadowing capabilities of the Apple IIGS, this task is simplified.

When shadowing is turned on, you draw your original screen display into banks $00 and $01. To display the overlay, turn shadowing off and write directly into banks $E0 and $E1. This affects only the display and not the original screen data that is also present in banks $00 and $01. When you are finished with the overlay, enable shadowing again and simply read and write the screen data (use MVN or MVP for speed) into the current screen area using banks $00 and $01. This will have no effect on banks $00 or $01, but it will restore the display to its appearance before the overlay data was written.

Glossary

accumulator: The register in a computer's central processor or microprocessor where most computations are performed.

ACIA: Abbreviation for *Asynchronous Communications Interface Adapter,* a type of communications IC used in some Apple computers. An ACIA converts data from parallel to serial form and vice versa. It handles serial transmission and reception and RS-232-C signals under the control of its internal registers, which can be set and changed by firmware or software. Compare **SCC.**

ADB: See **Apple DeskTop Bus.**

address: A number that specifies the location of a single byte of memory. Addresses can be given as decimal or hexadecimal integers. The Apple IIGS has addresses ranging from 0 to 16,777,215 (decimal) or from $00 00 00 to $FF FF FF (hexadecimal). A complete address consists of a 4-bit **bank** number ($00 to $FF) followed by a 16-bit address within that bank ($00 00 to $FF FF).

Apple DeskTop Bus (ADB): A low-speed serial input port that supports the keyboard, the ADB mouse, and additional input devices, such as hand controls and graphics tablets.

Apple key: A modifier key on the Apple IIGS keyboard, marked with both an Apple icon and a spinner, the icon used on the equivalent key on some Macintosh keyboards. It performs the same functions as the ⌘ key on standard Apple II computers.

AppleTalk: Apple's local-area network for Apple II and Macintosh personal computers and the LaserWriter and ImageWriter II printers. Like the Macintosh, the Apple IIGS has the AppleTalk interface built in.

AppleTalk connector: A piece of equipment consisting of a connection box, a short cable, and an 8-pin miniature **DIN** connector that enables an Apple IIGS to be part of an AppleTalk network.

Apple II: A family of computers, including the original Apple II, the Apple II Plus, the Apple IIe, the Apple IIc, and the Apple IIGS. Compare **standard Apple II.**

Apple IIGS Programmer's Workshop (APW): The development environment for the Apple IIGS computer. It consists of a set of programs that facilitate the writing, compiling, and debugging of Apple IIGS applications.

APW: See **Apple IIGS Programmer's Workshop.**

assembler: A program that produces **object files** (programs that contain machine-language code) from **source files** written in assembly language. The opposite of **disassembler.**

background printing: Printing from one application while another application is running.

bank: A 64K (65,536-byte) portion of the Apple IIGS internal memory. An individual bank is specified by the value of one of the 65C816 microprocessor's bank registers.

bank-switched memory: On Apple II computers, that part of the **language-card memory** in which two 4K portions of memory share the same address range ($D000 to $DFFF).

BASICOUT: The routine that outputs a character when the 80-column firmware is active.

Battery RAM: RAM memory on the Apple IIGS clock chip. A battery preserves the clock settings and the RAM contents when the power is off. Control Panel settings are kept in the Battery RAM.

baud rate: The rate at which serial data is transferred, measured in signal transitions per second. It takes approximately 10 signal transitions to transmit a single character.

bit: A contraction of *binary digit*. The smallest unit of information a computer can hold. The value of a bit (1 or 0) represents a simple two-way choice, such as on or off.

block: (1) A unit of data storage or transfer, typically 512 bytes. (2) A contiguous, page-aligned region of computer memory of arbitrary size, allocated by the Memory Manager. Also called a *memory block*.

block device: A device that transfers data to or from a computer in multiples of one block (512 bytes) of characters at a time. Disk drives are block devices. Also called *block I/O device*.

boot: Another way to say *start up*. A computer boots by loading a program into memory from an external storage medium such as a disk. The word *boot* is short for *bootstrap load*. Starting up is often accomplished by first loading a small program, which then reads a larger program into memory. The program is said to "pull itself up by its own bootstraps."

buffer: A holding area in the computer's memory (for example, a print buffer) where information can be stored by one program or device and then read at a different rate by another.

byte: A unit of information consisting of a sequence of 8 bits. A byte can take any value between 0 and 255 ($0 and $FF hexadecimal). The value can represent an instruction, a number, a character, or a logical state.

carry flag: A status bit in the microprocessor, used as an additional high-order bit with the accumulator bits in addition, subtraction, rotation, and shift operations.

central processing unit (CPU): The part of the computer that performs the actual computations in machine language. See also **microprocessor.**

character: Any symbol that has a widely understood meaning and thus can convey information. Some characters, such as letters, numbers, and punctuation, can be displayed on the monitor screen and printed on a printer. Most characters are represented in the computer as 1-byte values.

clamp: A memory location that contains the maximum and minimum excursion positions of the mouse cursor when the desktop is in use.

CMOS: Acronym for *complementary metal oxide semiconductor,* one of several methods of making integrated circuits out of silicon. CMOS devices are characterized by low power consumption.

controller card: A peripheral card that connects a device such as a printer or disk drive to a computer's main logic board and controls the operation of the device.

Control Panel: A **desk accessory** that lets the user change certain system parameters, such as speaker volume, display colors, and configuration of slots and ports.

control register: A special register that programs can read and write, similar to a **soft switch.** The control registers are specific locations in the I/O space ($Cxxx) in bank $E0; they are accessible from bank $00 if I/O shadowing is on.

Control-Reset: A combination keystroke on Apple II computers that usually causes an Applesoft BASIC program or command to stop immediately.

COUT: The firmware entry point for the Apple II character-output subroutine. COUT is actually an I/O **link** located in RAM rather than in ROM, and so can be modified to contain the address of the presently active character-output subroutine.

COUT1: An entry point within the Apple II character-output subroutine.

C3COUT1: Also called **BASICOUT,** this is the routine that **COUT** jumps to when the 80-column firmware is active.

data: Information transferred to or from, or stored in, a computer or other mechanical communications or storage device.

DCD: Abbreviation for *Data Carrier Detect,* a modem signal indicating that a communication connection has been established.

Delete key: A key on the upper-right corner of the Apple IIe, Apple IIc, and Apple IIGS keyboards that erases the character immediately preceding (to the left of) the cursor. Similar to the Macintosh Backspace key.

delta: The difference from something the program already knows. For example, mouse moves are represented as deltas compared to previous mouse locations. The name comes from the way mathematicians use the Greek letter delta (Δ) to represent a difference.

desk accessory: A small, special-purpose program available to the user regardless of which application is running. The **Control Panel** is an example of a desk accessory.

desktop: The visual interface between the computer and the user—the menu bar and the gray area on the screen.

device: A piece of hardware used in conjunction with a computer and under the computer's control. Also called a *peripheral device* because such equipment is often physically separate from (but attached to) the computer.

device driver: A program that manages the transfer of information between the computer and a peripheral device.

Digital Oscillator Chip (DOC): An integrated circuit in the Apple IIGS that contains 32 digital oscillators, each of which can generate a sound from stored digital waveform data.

DIN: Acronym for *Deutsche Industrie Normal,* a European standards organization.

DIN connector: A type of connector with multiple pins inside a round outer shield.

direct page: A page (256 bytes) of bank $00 of Apple IIGS memory, any part of which can be addressed with a short (1-byte) address because its high-order byte of the address is always $00 and its middle byte of the address is the value of the 65C816 **direct register.** Coresident programs or routines can have their own direct pages at different locations. The direct page corresponds to the 6502 processor's **zero page.** The term *direct page* is often used informally to refer to any part of the lower portion of the **direct-page/stack space.**

direct-page/stack space: A portion of bank $00 of Apple IIGS memory reserved for a program's **direct page** and **stack.** Initially, the 65C816 processor's **direct register** contains the base address of the space, and its **stack register** contains the highest address. In use, the stack grows downward from the top of the direct-page/stack space, and the lower part of the space contains direct-page data.

direct register: A hardware register in the 65C816 processor that specifies the start of the direct page.

disassembler: A program that examines data in memory and interprets it as a set of assembly-language instructions. Assuming the data is object code, a disassembler gives the user the source code that could have generated that object code.

disk operating system: An operating system whose principal function is to manage files and communication with one or more disk drives. **DOS** and **ProDOS** are two families of Apple II disk operating systems.

Disk II drive: A type of disk drive made and sold by Apple Computer for use with the Apple II, Apple II Plus, and Apple IIe computers. It uses 5.25-inch disks.

DOC: See **Digital Oscillator Chip.**

DOS: An Apple II disk operating system. Acronym for *Disk Operating System.*

Double Hi-Res: A high-resolution graphics display mode on Apple II computers with at least 128K of RAM, consisting of an array of points 560 wide by 192 high with 16 colors.

DSR: Abbreviation for *Data Set Ready,* a signal indicating that a modem has established a connection.

DTR: Abbreviation for *Data Terminal Ready,* a signal indicating that a terminal is ready to transmit or receive data.

e flag: One of three flag bits in the 65C816 processor that programs use to control the processor's operating modes. The setting of the e flag determines whether the processor is in **native mode** or **emulation mode.** See also **m flag** and **x flag.**

8-bit Apple II: Another way of saying **standard Apple II;** that is, any Apple II with an 8-bit microprocessor (6502 or 65C02).

80-column text card: A peripheral card that allows the Apple II, Apple II Plus, and Apple IIe computers to display text in 80 columns (in addition to the standard 40 columns).

emulate: To operate in a way identical to a different system. For example, the 65C816 microprocessor in the Apple IIGS can carry out all the instructions in a program originally written for an Apple II that uses a 6502 microprocessor, thus emulating the 6502.

emulation mode: The 8-bit configuration of the 65C816 processor in which the processor functions like a 6502 processor in all respects except clock speed.

environment: The complete set of machine registers associated with a running program. Saving the environment allows a program to be restored to its original operating mode with all of its registers intact as though nothing had happened. Saving and restoring an environment is most often associated with calling system functions or processing interrupts.

error: The state of a computer after it has detected a fault in one or more commands sent to it. Also called **error condition.**

escape code: A key sequence formed by pressing the Esc (Escape) key, followed by pressing another key. Escape codes are used to control the video firmware.

escape mode: The mode of video-firmware operation activated by pressing the Esc (Escape) key. It allows for moving the cursor, picking up characters from the screen, and performing other special operations.

extended SmartPort call: A SmartPort call that allows data transfer to or from anywhere in the Apple IIGS system memory space. Compare **standard SmartPort call.**

field: A string of ASCII characters or a value that has a specific meaning to some program. Fields may be of fixed length or may be separated from other fields by field delimiters. For example, each parameter in a segment header constitutes a field.

firmware: Programs stored permanently in ROM; most provide an interface to system hardware. Such programs (for example, the **Monitor program**) are built into the computer at the factory. They can be executed at any time, but cannot be modified or erased from main memory.

format: (n) The form in which information is organized or presented. (v) To divide a disk into tracks and sectors where information can be stored; synonymous with *initialize*. Blank disks must be formatted before the user can save information on them for the first time.

frequency: The rate at which a repetitive event recurs. In alternating current (AC) signals, the number of cycles per second. Frequency is usually expressed in **hertz** (cycles per second), **kilohertz,** or **megahertz.**

GETLN: The firmware routine that a program calls to obtain an entire line of characters from the currently active input device.

GLU: Acronym for *general logic unit,* a class of custom integrated circuits used as interfaces between different parts of the computer.

handshaking: The exchange of status information between two data terminals used to control the transfer of data between them. The status information can be the state of a signal connecting the two terminals, or it can be in the form of a character transmitted with the rest of the data.

hertz (Hz): The unit of frequency of vibration or oscillation, defined as the number of cycles per second. Named for the physicist Heinrich Hertz. See also **kilohertz** and **megahertz.**

hexadecimal: The base-16 system of numbers, using the ten digits 0 through 9 and the six letters A through F. Hexadecimal numbers can be converted easily and directly to binary form, because each hexadecimal digit corresponds to a sequence of 4 bits. In Apple manuals, hexadecimal numbers are usually preceded by a dollar sign ($).

high order: The most significant part of a numerical quantity. In normal representation, the *high-order bit* of a binary value is in the leftmost position; likewise, the *high-order byte* of a binary **word** or **longword** quantity consists of the leftmost 8 bits.

Hi-Res: A high-resolution graphics display mode on the Apple II family of computers, consisting of an array of points 280 wide by 192 high with 6 colors.

Human Interface Guidelines: A set of software development guidelines designed by Apple Computer to support the **desktop** concept and to promote uniform user interfaces in Apple II and Macintosh applications.

icon: An image that graphically represents an object, a concept, or a message.

index register: A register in a computer processor that holds an index for use in indexed addressing. The 6502 and 65C816 microprocessors used in the Apple II family of computers have two index registers, called the *X register* and the *Y register.*

initialize: See **format** (v).

intelligent device: A device containing a microprocessor and a program that allows the device to interpret data sent to it as commands that the device is to perform.

interpreter: A program that interprets its **source files** on a statement-by-statement or character-by-character basis.

interrupt handler: A program, associated with a particular external device, that executes whenever that device sends an interrupt signal to the computer. The interrupt handler performs its tasks during the interrupt, then returns control to the computer so it may resume program execution.

IRQ: A 65C816 signal line that, when activated, causes an interrupt request to be generated.

IWM: Abbreviation for *Integrated Woz Machine,* the custom chip used in built-in disk ports on Apple computers.

KEYIN: The firmware entry point that a program calls to obtain a keystroke from the currently active input device (normally the keyboard).

kilobit: A unit of measurement, 1024 bits, commonly used in specifying the capacity of memory integrated circuits. Not to be confused with **kilobyte.**

kilobyte: A unit of measurement, 1024 bytes, commonly used in specifying the capacity of memory or disk storage systems.

kilohertz (kHz): A unit of measurement of frequency, equal to 1000 **hertz.** Compare **megahertz.**

language-card memory: Memory with addresses between $D000 and $FFFF on any Apple II–family computer. It includes two RAM banks in the $Dxxx space, called **bank-switched memory.** The language card was originally a peripheral card for the 48K Apple II or Apple II Plus computer that expanded the computer's memory capacity to 64K and provided space for an additional dialect of BASIC.

last-changeable location: The last location whose value the user inquired about through the Monitor.

link: An area in memory that contains an address and a jump instruction. Programs are written to jump to the link address. Other programs can modify this address to make everything behave differently. **COUT** and **KEYIN** are examples of I/O links.

longword: A double-length word. For the Apple IIGS, a long word is 32 bits (4 bytes) long.

Lo-Res: The lowest resolution graphics display mode on the Apple II family of computers, consisting of an array of blocks 48 high by 40 wide with 16 colors.

low order: The least significant part of a numerical quantity. In normal representation, the *low-order bit* of a binary number is in the rightmost position; likewise, the *low-order byte* of a binary **word** or **longword** quantity consists of the rightmost 8 bits.

megabit: A unit of measurement equal to 1,048,576 (2^{16}) bits, or 1024 **kilobits.** Megabits are commonly used in specifying the capacity of memory integrated circuits. Not to be confused with **megabyte.**

megabyte: A unit of measurement equal to 1,048,576 (2^{16}) bytes, or 1024 **kilobytes.** Megabytes are commonly used in specifying the capacity of memory or disk storage systems.

megahertz (MHz): A unit of measurement of frequency, equal to 1,000,000 **hertz.** Compare **kilohertz.**

Mega II: A custom large-scale integrated circuit that incorporates most of the timing and control circuits of the standard Apple II. It addresses 128K of RAM organized as 64K main and auxiliary banks and provides the standard Apple II video display modes, both text (40-column and 80-column) and graphics (Lo-Res, Hi-Res, and Double Hi-Res).

memory block: See **block** (2).

Memory Manager: A program in the Apple IIGS Toolbox that manages memory use. The Memory Manager keeps track of how much memory is available and allocates memory **blocks** to hold program segments or data.

memory-mapped I/O: The method used for I/O operations in Apple II computers. Certain memory locations are attached to I/O devices, and I/O operations are just memory load and store instructions.

m flag: One of three flag bits in the 65C816 processor that programs use to control the processor's operating modes. In **native mode,** the setting of the m flag determines whether the accumulator is 8 or 16 bits wide. See also **e flag** and **x flag.**

microprocessor: A central processing unit that is contained in a single integrated circuit. The Apple IIGS uses a 65C816 microprocessor.

mini-assembler: A part of the Apple IIGS **Monitor program** that allows the user to create small assembly-language test routines. See also **assembler.**

Monitor program: A program built into the firmware of Apple II computers, used for directly inspecting or changing the contents of main memory and for operating the computer at the machine-language level.

MOS: Acronym for *metal oxide semiconductor,* one of several methods of making integrated circuits.

native mode: The 16-bit configuration of the 65C816 microprocessor.

next-changeable location: The memory location that is next to have its value changed.

NTSC: (1) Abbreviation for *National Television Standards Committee,* which defined the standard format used for transmitting broadcast video signals in the United States. (2) The standard video format defined by the NTSC; also called *composite* because it combines all video information, including color, into a single signal.

object file: The output from an assembler or a compiler, and the input to a linker. It contains machine-language intructions. Also called *object program* or *object code.* Compare **source file.**

op code: See **operation code.**

Apple: A modifier key on some Apple II keyboards. On the Apple IIGS keyboard, the equivalent key is called simply the **Apple key;** it is marked with both an Apple icon and a spinner, the icon used on some Macintosh keyboards.

operand: An item on which an *operator* (such as + or AND) acts.

operation code: The part of a machine-language instruction that specifies the operation to be performed. Often called *op code.*

page: (1) A portion of memory 256 bytes long and beginning at an address that is an even multiple of 256. Memory blocks whose starting addresses are an even multiple of 256 are said to be *page aligned.* (2) (usually capitalized) An area of main memory containing text or graphic information being displayed on the screen.

palette: The set of colors from which the user can choose a color to apply to a pixel on the screen.

parameter: A value passed to or from a function or other routine.

parameter block: A set of contiguous memory locations set up by a calling program to pass parameters to and receive results from an operating-system function that the program calls. Every call to SmartPort must include a pointer to a properly constructed parameter block.

parity bit: A bit that is sometimes transmitted along with the other bits that define a serial character. It is used to check the accuracy of the transmission of the character. *Even parity* means that the total number of 1 bits transmitted, including the parity bit itself, is even. *Odd parity* means that the total number is odd. The parity bit is generated individually for each character and checked, a character at a time, at the receiving end.

peripheral device: See **device.**

pixel: Short for *picture element.* The smallest dot that can be drawn on the screen. Also a location in video memory that corresponds to a point on the graphics screen when the viewing window includes that location. In the Macintosh display, each pixel can be either black or white, so it can be represented by a bit; thus, the display is said to be a *bit map.* In the Super Hi-Res display on the Apple IIGS, each pixel is represented by either 2 or 4 bits; the display is not a bit map, but rather a **pixel map.**

pixel map: A set of values that represents the positions and states of the set of **pixels** making up an image.

ProDOS: Acronym for *Professional Disk Operating System,* a family of disk operating systems developed for the Apple II family of computers. ProDOS includes both **ProDOS 8** and **ProDOS 16.**

ProDOS 8: A disk operating system developed for standard Apple II computers. It runs on 6502-series microprocessors and on the Apple IIGS when the 65C816 processor is in 6502 **emulation mode.**

ProDOS 16: A disk operating system developed for 65C816 **native-mode** operation on the Apple IIGS. It is functionally similar to ProDOS 8, but more powerful.

prompt: A message on the screen that a program provides when it needs a response from the user. A prompt is usually in the form of a symbol, a dialog box, or a menu of choices.

Quagmire register: On the Apple IIGS, the name given to the 8 bits comprising the speed-control bit and the shadowing bits. From the Monitor program, the user can read from or write to the Quagmire register to access those bits, even though they are actually in separate registers.

RAM: See **random-access memory.**

RAM disk: A portion of RAM that appears to the operating system to be a disk volume. Files in a RAM disk can be accessed much faster than the same files on a disk. See also **ROM disk.**

random-access memory (RAM): Memory in which information can be referred to in an arbitrary or random order. RAM usually means the part of memory available for programs from a disk; the programs and other data are lost when the computer is turned off. (Technically, the read-only memory is also *random access,* and what's called RAM should correctly be termed *read-write memory.*) Compare **read-only memory.**

RDKEY: The firmware routine that a program uses to read a single keystroke from the keyboard.

read-only memory (ROM): Memory whose contents can be read, but not changed; used for storing **firmware.** Information is placed into read-only memory once, during manufacture; it then remains there permanently, even when the computer's power is turned off. Compare **random-access memory.**

recharge routine: The function that supplies data to the output device when **background printing** is taking place.

RGB: Abbreviation for *red-green-blue*. A method of displaying color video by transmitting the three primary colors as three separate signals. There are two ways of using RGB with computers: *TTL RGB*, which allows the color signals to take on only a few discrete values; and *analog RGB*, which allows the color signals to take on any values between their upper and lower limits for a wide range of colors.

ROM: See **read-only memory.**

ROM disk: A feature of some operating systems that permits the use of ROM as a disk volume. Often used for making applications permanently resident. See also **RAM disk.**

RS-232: A common standard for serial data communication interfaces.

RS-422: A standard for serial data communication interfaces, different from the RS-232 standard in its electrical characteristics and in its use of differential pairs for data signals. The serial ports on the Apple IIGS use RS-422 devices modified so as to be compatible with RS-232 devices.

SCC: Abbreviation for *Serial Communications Controller*, a type of communications IC used in the Apple IIGS. The SCC can run synchronous data transmission protocol and thus transmit data at faster rates than the **ACIA.**

screen holes: Locations in the text display buffer (text Page 1) used for temporary storage either by I/O routines running in peripheral-card ROM or by firmware routines addressed as if they were in card ROM. Text Page 1 occupies memory from $0400 to $07FF; the screen holes are locations in that area that are neither displayed nor modified by the display firmware.

sector: See **track.**

shadowing: The process whereby any changes made to one part of the Apple IIGS memory are automatically and simultaneously copied into another part. When shadowing is on, information written to bank $00 or $01 is automatically copied into equivalent locations in bank $E0 or $E1. Likewise, any changes to bank $E0 or $E1 are immediately reflected in bank $00 or $01.

64K Apple II: Any standard Apple II that has at least 64K of RAM. This includes the Apple IIc, the Apple IIe, and an Apple II or Apple II Plus with 48K of RAM and the language card installed.

6502: The microprocessor used in the Apple II, the Apple II Plus, and early models of the Apple IIe. The 6502 is a **MOS** device with 8-bit data registers and 16-bit address registers.

65C02: A **CMOS** version of the 6502, this is the microprocessor used in the Apple IIc and the enhanced Apple IIe.

65C816: The microprocessor used in the Apple IIGS. The 65C816 is a **CMOS** device with 16-bit data registers and 24-bit address registers.

SmartPort: A set of firmware routines supporting multiple block devices connected to the Apple IIGS disk port. See also **extended SmartPort call** and **standard SmartPort call.**

soft switch: A location in memory that produces a specific effect whenever its contents are read or written.

source file: An ASCII file consisting of instructions written in a particular language, such as Pascal or assembly language. An assembler or a compiler converts a source file into an **object file.**

SSC: Abbreviation for *Super Serial Card*, a peripheral card that enables an Apple II to communicate with serial devices.

stack: A list in which entries are added (pushed) and removed (pulled) at one end only (the top of the stack), causing them to be removed in last-in, first-out (LIFO) order. *The stack* usually refers to the particular stack pointed to by the 65C816's **stack register.**

stack register: A hardware register in the 65C816 processor that contains the address of the top of the processor's **stack.**

standard Apple II: Any computer in the Apple II family except the Apple IIGS. This includes the Apple II, the Apple II Plus, the Apple IIe, and the Apple IIc.

standard SmartPort call: A SmartPort call that allows data transfer to or from anywhere in standard Apple II memory, or the lowest 64K of Apple IIGS memory. Compare **extended SmartPort call.**

start up: To get the system running. See **boot.**

Super Hi-Res: A high-resolution graphics display mode on the Apple IIGS, consisting of an array of points 320 wide by 200 high with 16 colors or 640 wide by 200 high with 16 colors (with restrictions).

synthesizer: A hardware device capable of creating sound digitally and converting it into an analog waveform that can be heard.

system disk: A disk that contains the operating system and other system software needed to run applications.

system software: The components of a computer system that support application programs by managing system resources such as memory and I/O devices.

terminal mode: The mode of operation in which the Apple IIGS acts like an intelligent terminal.

text window: The portion of the Apple II screen that is reserved for text. At startup, the firmware initializes the entire display to text. However, applications can restrict text to any rectangular portion of the display.

tool: See **tool set.**

toolbox: A collection of built-in routines on the Apple IIGS that programs can call to perform many commonly needed functions. Functions within the toolbox are grouped into **tool sets.**

tool set: A group of related routines (usually in firmware) that perform necessary functions or provide programming convenience. They are available to applications and system software. The Memory Manager, the System Loader, and QuickDraw II are tool sets.

track: One of a series of concentric circles that are magnetically drawn on the recording surface of a disk when the disk is formatted. Tracks are further divided into *sectors.*

vector: A location containing a value that, when added to a base address value, provides the address that is the entry point of a specific kind of routine.

word: A group of bits that is treated as a unit. For the Apple IIGS, a word is 16 bits (2 bytes) long.

x flag: One of three flag bits in the 65C816 processor that programs use to control the processor's operating modes. In **native mode,** the setting of the x flag determines whether the index registers are 8 or 16 bits wide. See also **e flag** and **m flag.**

XON: A special character (value $13) used for controlling the transfer of data between two pieces of equipment. See also **handshaking** and **XOFF.**

XOFF: A special character (value $11) used for controlling the transfer of data between two pieces of equipment. When one piece of equipment receives an XOFF character from the other, it stops transmitting characters until it receives an XON. See also **handshaking** and **XON.**

zero page: The first page (256 bytes) of memory in a standard Apple II computer (or in the Apple IIGS when running a standard Apple II program). Because the high-order byte of any address in this part of memory is zero, only a single byte is needed to specify a zero-page address. Compare **direct page.**

Index

S

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using the Apple Macintosh™ Plus and Microsoft Word. Proof and final pages were created on the Apple LaserWriter® Plus. POSTSCRIPT™, the LaserWriter page-description language, was developed by Adobe Systems Incorporated.

Text type is ITC Garamond® (a downloadable font distributed by Adobe Systems). Display type is ITC Avant Garde Gothic®. Bullets are ITC Zapf Dingbats®. Program listings are set in Apple Courier, a monospaced font.




The Apple Technical Library

The Official Publications from Apple Computer, Inc.

The Apple Technical Library offers programmers, developers, and enthusiasts the most complete technical information available on Apple® computers, peripherals, and software. The library consists of technical manuals for the Apple II family of computers, the Macintosh™ family of computers, and their key peripherals and programming environments.

Manuals for the Apple II family include technical references to the Apple IIe, Apple IIc, and Apple IIgs™ computers, with detailed descriptions of the hardware, firmware, ProDOS® operating systems, and built-in programming tools that programmers and developers can draw upon. In addition to a technical introduction and programmer's guide to the Apple IIgs, there are tutorials and references for Applesoft BASIC and Instant Pascal programmers.

Manuals for the Macintosh family, known collectively as the Inside Macintosh Library, provide complete technical references to the Macintosh 512K, Macintosh 512K Enhanced, Macintosh Plus, Macintosh SE, and Macintosh II computers. Individual volumes provide technical introductions and programmer's guides to the Macintosh, as well as detailed information on hardware, firmware, system software, and programming tools. The Inside Macintosh Library offers the most detailed and complete source of information available for the Macintosh family of computers.

In addition, titles in the Apple Technical Library offer references to the wide range of important printers, communications standards, and programming environments—such as the Standard Apple Numerics Environment (SANE™)—to help programmers and experienced users get the most out of their computer systems.

The Official Publication from Apple Computer, Inc.

Now programmers and designers have a comprehensive guide to the inner workings of the popular Apple IIGS™ computer.

With its impressive 256K base memory, expandable to well over 4 megabytes, and its enhanced color graphics and sound capabilities, the Apple IIGS is destined to become the new standard in the educational computer market, and the choice of software developers. As the Apple IIGS user base grows, more and more programmers need the important technical information found only in this manual.

The *Apple IIGS Firmware Reference* the companion volume to the *Apple IIGS Hardware Reference* is Apple's definitive guide for assembly-language programmers and hardware developers working with the Apple IIGS. In a single volume, it provides an extensive description of the internal operations of the machine and presents the latest information about the firmware facilities that the IIGS provides.

The manual begins with an overview of Apple IIGS firmware. Then, in detail, it tells how to use the firmware to access the system's monitor, mini-assembler, disassembler, keyboard, mouse, video display, serial ports, and disk drives.

Detailed appendixes contain summary tables and information about the firmware, and tell how a user can include firmware calls within programs, thereby allowing the user to really have control over the machine. The *Apple IIGS Firmware Reference* provides the most authoritative and comprehensive information available on this amazingly versatile computer.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010
TLX 171-576

030-3121-A
Printed in U.S.A.

Addison-Wesley Publishing Company, Inc.

ISBN 0-201-17744-7