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.

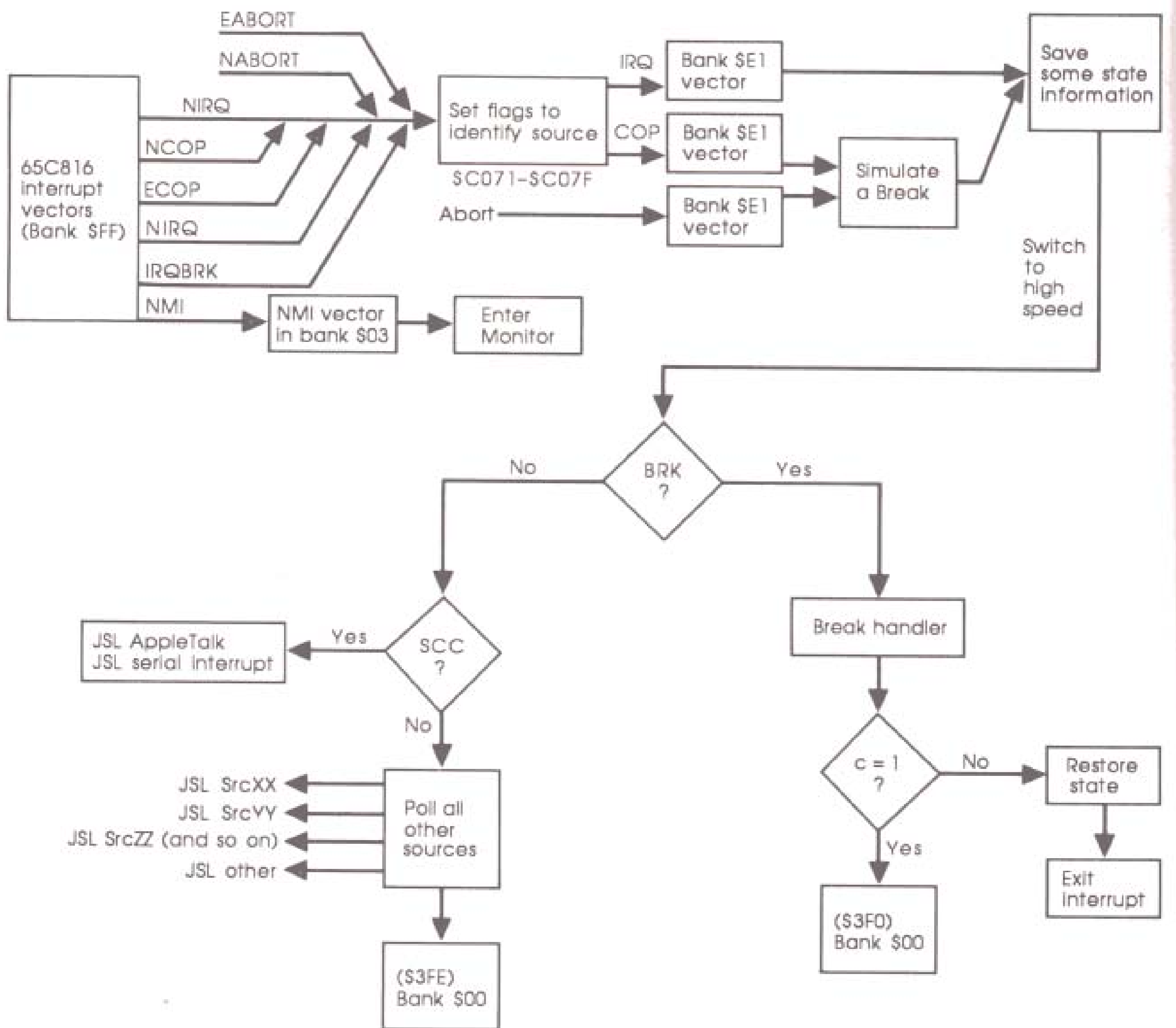


Figure 8-1
Built-in interrupt handler

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.

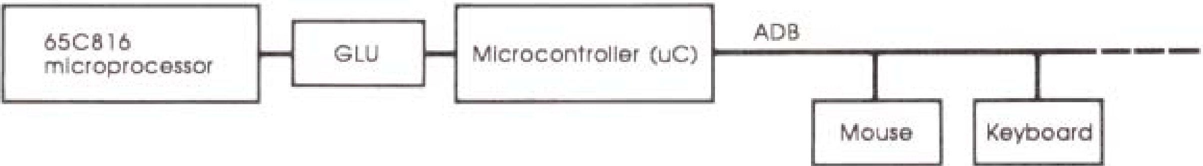


Figure 9-1
Apple DeskTop Bus components

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)

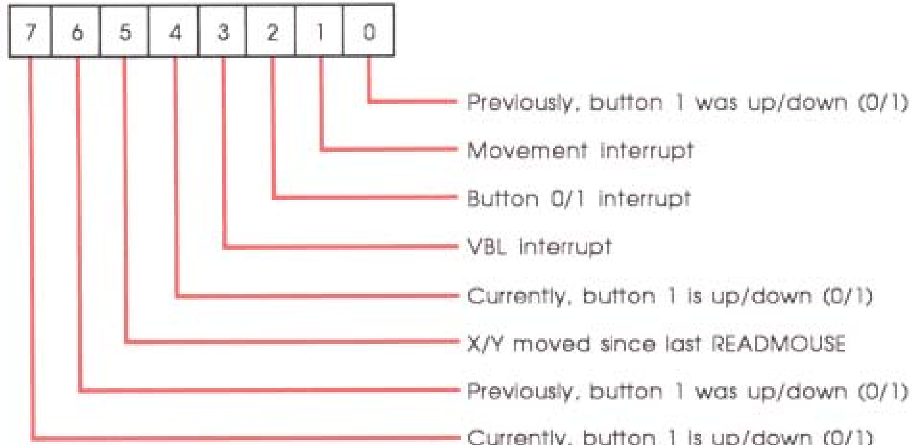


Figure 10-1
Button interrupt status byte, $77C

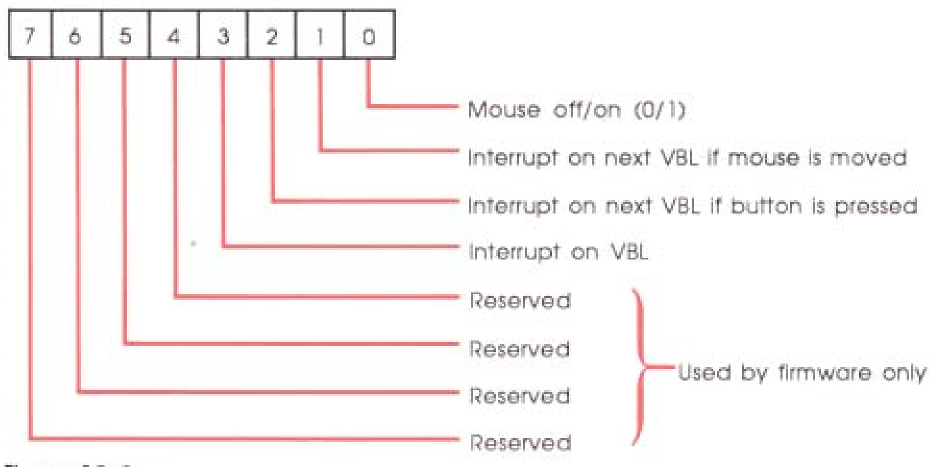


Figure 10-2
Mode byte, $7FC

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

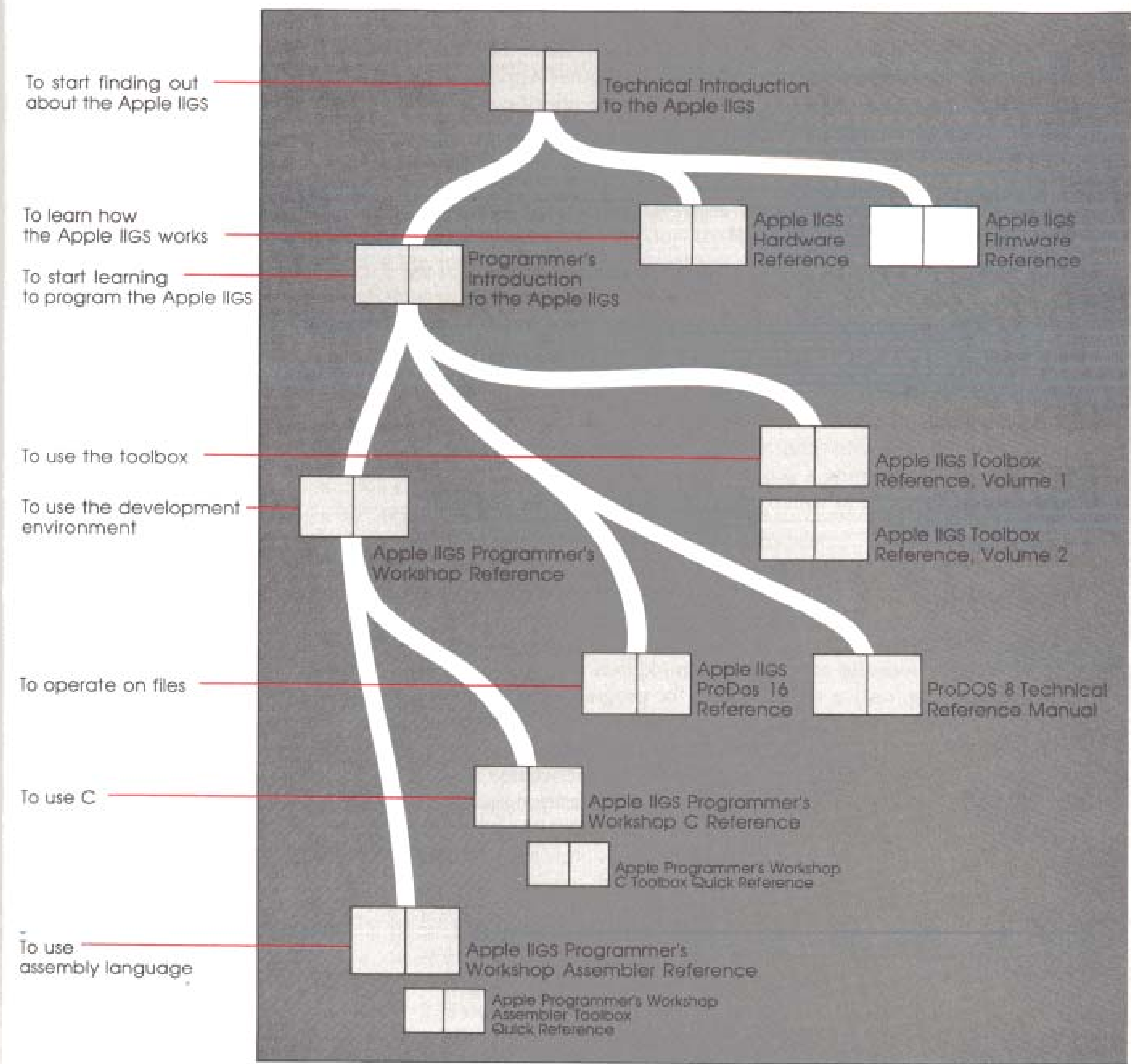
The Apple IIGS personal computer has many advanced features, making it more complex than earlier models of the Apple II. To describe it fully, Apple has produced a suite of technical manuals. Depending on the way you intend to use the Apple IIGS, you may need to refer to a select few of the manuals,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

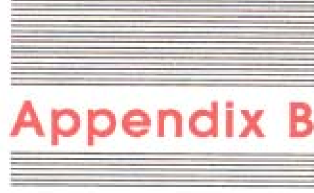
ProDOS 8, previously just called *ProDOS*, is the standard operating system for most Apple II computers with 8-bit CPUs (Apple IIc, IIe, and 64K II Plus). It also runs on the Apple IIGS. As a developer of Apple IIGS programs, you need the *ProDOS 8 Technical Reference Manual* only if you are developing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

$FD0C **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

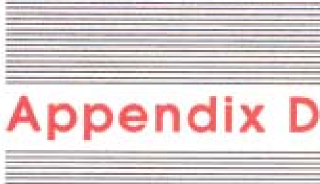
Output Unchanged = DBR/K/D/e
Scrambled = A/B/P/X/Y

$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



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

Address	Name	Description
$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--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 = (open apple) key active
C025:        148 *    Bit 6 = (closed apple) 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   |ln 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 ( 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

Address	Symbol	Address	Symbol	Address	Symbol	Address	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	SOUNDADRl	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

Name	Mode	Bytes	Opcode number
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

Option	Choices
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
Month	1–12
Day	1–31
Year	1904–2044
Format	√ MM/DD/YY DD/MM/YY YY/MM/DD

Option	Choices
Hour	1–12 or 0–23 (depends on Format selected)
Minute	0–59
Second	0–59
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.

Address	Bank $E0	Bank $E1
$FFFF–$E000	$E0 main language card $20 pages (8K reserved)	$E1 aux language card $20 pages (8K reserved)
$E000–$D000	Bank $00 $10 pages (4K reserved) / Bank $01 $10 pages (4K reserved)	Bank $00 $10 pages (4K reserved) / Bank $01 $10 pages (4K reserved)
$D000–$C000	I/O (always active)	I/O (always active)
$C000–$A000	$60 pages (24K free space)	$20 pages (8K free space)
$A000–$6000 (via $8000, $7000)		Super Hi-Res ($6000–$9FFF) **Graphics**
$6000–$4000 (via $5000)	Double Hi-Res page two ($4000–$5FFF) **Graphics**	Double Hi-Res page two ($4000–$5FFF) **Graphics**
$4000–$2000 (via $3000)	Double Hi-Res page one ($2000–$3FFF) **Graphics**	Double Hi-Res page one ($2000–$3FFF) **Graphics**
$2000–$0C00	$14 pages (5K reserved)	$14 pages (5K reserved)
$0C00–$0800	Text Page 2	Text Page 2
$0800–$0400	Text Page 1	Text Page 1
$0400–$0000	$4 pages (1K reserved)	$4 pages (1K reserved)

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.**

: 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

A

ABORT 179
Abort command 188
ABORTMGRV 265
accumulator 35
accumulator mode 62
ADB microcontroller. *See* Apple DeskTop Bus microcontroller
addition 32-bit
ADVANCE 240
AMPERV 259
apostrophe (') 40, 64
Apple DeskTop Bus connector 8
Apple DeskTop Bus input devices 10
Apple DeskTop Bus microcontroller 6, 183, 185-196
  commands 188-195
  status byte 196
Apple 3.5 disk drive 117, 133, 135
  SmartPort calls 138-141
APPLEII 237
Apple IIc 11
Apple IIe Plus 222
Apple IIGS
  boot/scan sequence 17
  detached keyboard 10
  80-column display 71
  firmware 2-6
  40-column display 71
  interrupts 16
  I/O expansion slots 11
  I/O ports 11
  memory addresses 21
  memory space 9
  microprocessor 8-9
  Monitor. *See* system Monitor
  program operation levels 4
  sound system 10
  startup 112
  Super Hi-Res display 9-10
  technical manuals 216-221
  Toolbox 2, 218, 310
Apple IIGS Disk II
  firmware 5
  I/O port characteristics 111
  SmartPort interactions 158
  support 109-112
Applesoft BASIC 2, 43, 74, 87, 112, 178
Apple Super Serial Card (SSC) 82
AppleTalk 3, 8, 15, 17, 82, 98, 173
  interrupts 180
A register 18, 35
  changing 60
  system interrupt handler 181
arrow keys 72
ASCII 25, 26, 29, 51, 67, 86, 123, 152
  filters 31
  flip 30, 64
  input mode 30
  literal 30, 64
assembly language
  mouse routines 202, 211-213
  Pascal protocol 93-94
at sign (@) 226
AUXMOVE 260

B

Back Arrow key 75
background printing 97-98
backslash (\) 75
Backspace key 70, 75
BADBLOCK 156
BADCMD 156
BADCTL 156
BADCTLPARM 156
BADPCNT 156
BADUNIT 156
bank $00 12, 15
  firmware entry points 224-257
  page Fx vectors 262-263
  page 3 routines 260-261
  page 3 vectors 259
  running a program in 49, 65
bank/address 21, 22, 26, 29, 32, 64
bank $E0 308-310
bank $E1 308-310
  vectors 264-265
BASCALC 239
BASIC 48, 51, 74, 75, 82, 83, 86, 87, 90, 112
  command 43
  interface 93
  mouse programs 206-208
  mouse routines 203
BASICIN 70-73
BASICINPUT 209
BASICOUT 70, 73, 76-78, 80
BASICOUTPUT 209
Battery RAM 299, 306
baud rate 88
BD command 96, 97, 183
BELL 253
BELL1 239
BELL1.2 239
BELL2 240
BELLVECTOR 270
BINITENTRY 209
boot-failure screen 17
boot/scan sequence 17
BREAK 233
Break (BRK) 36, 183
BREAKVECTOR 270
B register 18, 35
BRK 179

BRKV 259
BS 241
Buffering Enable 83
BUSERR 156
bus residents 157
button 1 status 204-205

C

Call statement 20
caret (^) 53, 55
carriage return 59, 75, 83
CLAMPMOUSE 209, 213
Clear Modes command 189
CLEARMOUSE 209, 212
CLEOLZ 79
clock 306
clock chip interrupts 180
Close call 5, 131-132
CLREOL 79, 243
CLREOLZ 243
CLREOP 79, 242
CLRSCR 79, 226
CLRTOP 79, 227
cold start 65, 112, 178, 234
colon (:) 28, 29, 40, 51, 52, 64
color graphics 10
command characters 87
  communications mode 87
  printer mode 87
  terminal mode 91-92
command packets, SmartPort 159, 166-167
command strings 87
communications mode 83
  command character 87
  commands 91-92
Continue BASIC command 43
Control-^ 64
Control-[77
Control-\ 77
Control-] 77
Control-_ 77
Control-^ 77
Control-A 87
Control-B 43, 65
Control-C 43, 65
Control call 129-130
control characters 73, 76-78
  suppressing 90
Control-E 60, 77
Control-F 77
Control-G 77
Control-H 77
Control-I 87
Control-J 77
Control-K 64, 77
Control-L 77
Control-M 77
Control-N 77
Control-O 77
Control-P 40, 64
Control Panel 3, 40, 75, 82, 83, 86, 90, 93, 97, 110, 112, 117, 130, 299-307
Control-Q 77
Control-R 66, 77
Control-Reset 43, 46, 112
Control-S 77
Control-T 64
Control-U 77
Control-V 77
Control-W 77, 87
Control-X 58, 75, 77, 247
Control-Y 47, 65, 77
COP 36, 179
COPMGRV 265
copy-protection engineer (CPE) tools 144-145
COPYRIGHT 209
COUT 70, 71, 75, 76, 79, 249
COUT1 64, 70, 74, 76-80, 249
COUT subroutine 39
COUTZ 249
CPE (copy-protection engineer) tools 144-145
CR 242
C register 18, 35
CROUT 79, 248
CROUT1 247
C3COUT1 64, 70, 76-78
CTRLYVECTOR 274
CUPDATE 269
cursor 71
  changing 41, 64
  control 72
   keys 10

D

data bank register 13, 16, 35, 92
  changing 61
  system interrupt handler 181
data buffer pointer 126-127
data byte encoding table 164
data carrier detect (DCD) 84
data format 88
data set ready (DSR) 84-85, 95
data terminal ready (DTR) 84-85, 95
date
  changing 64
  displaying 40, 63
DBR register 11, 13, 35
DCB (device control block) 123, 130
DCD (data carrier detect) 84
debugging 48
DECBUSYFLG 270
decimal numbers, converting 41, 65
Delete key 75
delta 199
Desk Manager 180
device control block (DCB) 123, 130
device mapping 117-119
DEVSPEC 156
DIAGMOUSE 209
diagnostic routines 3
DIG 256
Digital Oscillator Chip (DOC) 10
direct page 12, 15
direct-page register 13
direct register, system interrupt handler 181
Disable Device SRQ command 195
disassembler 55-56
  opcodes 293-298
Disk II firmware 5
DISKSW 156
DISPATCH1 264
DISPATCH2 264
dispatch address 115
display 302
division, 32-bit 42
DOC (Digital Oscillator Chip) 10
dollar sign ($) 54

DOS 70, 110
DOS 3.3 43
Download 143
D register 11, 35
  changing 60
DSR (data set ready) 84-85, 95
DTR (data terminal ready) 84-85, 95
DuoDisk 110

E

EABORT 177, 263
EBRKIRQ 263
echo 91
ECOP 177, 263
ED command 91
EE command 91
e flag 37
Eject 138, 142
emulation mode 9, 14, 37-38, 56, 120
  accumulator 18
  changing 62
  code 15
  stack 13
EMULSTACK 13
Enable Device SRQ command 194
enable line formatting 89
ENMI 263
Ensoniq chip interrupts 180
environment 8, 36
  firmware routines 11-16
  resetting 66
  restoring 14
  system interrupt handler 181
equal sign (=) 37
ERESET 263
error codes, SmartPort 156
error status register 95
Esc A 73
Esc @ 73
escape codes 72, 73
Escape key 72
escape mode 71, 72
Esc B 73
Esc C 73
Esc-Control-D 73
Esc-Control-E 73
Esc-Control-Q 73
Esc D 73
Esc E 73
Esc 8 73
Esc F 73
Esc 4 73
Esc I 73
Esc J 73
Esc K 73
Esc M 73
Event Manager 75, 183
Examine instruction 37
exclamation point (!) 52
Execute 142

F

FD command 90
FD10 245
Fill Memory command 59
filter mask, changing 63
firmware. *See also specific type*
  entry points 224-257
  ID bytes 222-223
  I/O routines 11-16, 79
flag-modification commands 38
flags 8, 12, 14, 16, 35-38
  examining and changing 36-38
  restoring 66
flashing text 78
flip ASCII 30, 64
Flush command 6, 180
Flush Device Buffer command 195
FlushInQueue 102
Flush Keyboard Buffer command 188
FlushOutQueue 102
Format 5, 128, 139, 147
free space 308, 310

G

GBASCALC 227
GetDTR 105
GET816LEN 230
GetInBuffer 101
GetIntInfo 96, 105, 184
GETLN 21, 71, 74-75, 79, 246
GETLN0 247
GETLN1 247
GETLNZ 246
GetModeBits 95, 100
GETNUM 256
GetOutBuffer 97, 98, 101
GetPortStat 104
GetSCC 104
Get Version Number command 192
GLU chip 183, 186, 199
GO 252
Go command 36, 49
graphics display modes 10
graphics tablets 10

H

handshaking 84-85
  protocol 89
HEADR 244
hexadecimal 21, 25, 26, 32, 53, 115, 116
  math 42
  numbers, converting 41, 65
HLINE 79, 226
HOME 79, 242
HOMEMOUSE 209, 213
hook table 145

I

IDROUTINE 250
immediate mode 56-57
INCBUSYFLG 270
index mode, changing 62
INIT 236
Init call 130
INITMOUSE 203, 209
INPORT 251
input buffer 46, 75, 91
input links, redirecting 64
input routines 71-75
InQStatus 96, 103
INSDS1.2 229
INSDS2 229
INSTDSP 230
Integer BASIC 43, 74

Integrated Woz Machine (IWM) chip 5, 110-111
intelligent devices 5
interrupt 15, 16, 95, 96-97, 171
  priorities 177-180
  processing 181-182
  vectors 177
interrupt handler 16
  built-in 172-174
  firmware 6, 169-184
Interrupt Request (IRQ) line 171
INTMGRV 264
Inverse command 39, 63
inverse text 78
inverse video 39, 71
IOERROR 156
I/O links 70
I/O port 5 114
IORTS 254
IRQ 180
IRQ.APTALK 266
IRQ.DSKACC 268
IRQ.EXT 269
IRQ.FLUSH 269
IRQ.KBD 268
IRQ.MICRO 269
IRQ.MOUSE 267
IRQ.1SEC 269
IRQ.OTHER 269
IRQ.QTR 268
IRQ.RESPONSE 268
IRQ.SCAN 267
IRQ.SERIAL 266
IRQ.SOUND 267
IRQ.SRQ 268
IRQ.VBL 267
IRQLOC 259
IRQVECT 177
IWM (Integrated Woz Machine) chip 5, 110-111

J

JMP instruction 47, 50, 65, 66, 145
joystick 10
JSL. *See* jump to subroutine long
JSR. *See* jump to subroutine
jump to subroutine (JSR) 12, 14, 47, 49, 49, 50, 114
jump to subroutine long (JSL) 12, 14, 50, 98, 152

K

KBDWAIT 238
keyboard 10, 40, 43, 71, 72
  input buffering 75
  interrupts 180
  language codes 190
Keyboard command 40
KEYIN 70, 71-72, 79, 245
K register 35

L

language card 16
  area 310
  bank 35, 56, 63
language options 305
last-opened location 25, 26
less-than character (<) 31, 34
LF 242
line feed 83
  automatic 90
  masking 91
line length 89
"LIST" 250
Listen 6
List instruction 53, 55, 66
literal ASCII 30, 64
local-area network. *See* AppleTalk

M

machine-language programs 48-50
machine registers 12
machine state 36
  changing 61
mailbox registers 186
mark table 144-145
Masking Enable 83
Mega II chip 308
memory 9
  changing 28-31, 64
  comparing data 33
  moving data 31-32
  searching for bytes 34
memory dump 27
memory locations
  changing 28-30
  displaying 58
  examining 26-27
  text window 80
Memory Manager 9, 15, 308, 310
memory range
  display 27
  filling 34
  terminating 58
m flag 37
microprocessor. *See specific type*
mini-assembler 51-55, 74
  instruction formats 54-55
  opcodes 293-298
modem communications 84
modem port 301
MON 255
Monitor. *See* system Monitor
Monitor command 49
Monitor firmware 4
MONZ 255
MONZ2 255
MONZ4 256
mouse
  interrupts 180, 183
  position clamps 201
  position data 199-201
mouse firmware 6, 197-213
  calls 209
  using 202-205
mouse programs, BASIC 206-208
MOVE 250
Move command 31-32, 45, 59
M register 36
MSGPOINTER 275
MSLOT 266
multiplication, 32-bit 42
music 10

N

NABORT 177, 262
native mode 9, 14, 56
  accumulator 18
  stack pointer 13, 14, 15
NBREAK 177, 262
NCOP 177, 262

next-changeable location 25, 26
NIRQ 177, 262
NMI 177, 178, 259
NNMI 177, 262
NODRIVE 156
NOINT 156
NONFATAL 156
Normal command 39, 44, 63
normal video 39
NOWRITE 156
numeric keypad 10
NXTA1 244
NXTA4 244
NXTCHAR 257
NXTCOL 227

O

OFFLINE 156
OLDBRK 233
OLDIRQ 233
OLDRST 255
opcodes 56-57, 293-298
Open call 5, 131
options 304-305
OPTMOUSE 209
OUTPORT 252
output links, redirecting 64
output routines 76-78
OutQStatus 96, 103

P

palettes 10
parity 89
Pascal 48, 82, 86, 97, 110, 210
Pascal 1.1 93
Pattern Search command 34, 59
PBR register 11, 35
PCADJ 232
period (.) 26, 27
picture element. *See* pixel
PInit 209, 210
pixel 10
PLOT 79, 225
PLOT1 225
plus sign (+) 71, 72
Poll Device command 195
POSMOUSE 209, 211, 213
PRA1 248
PRBL2 79, 231
PRBLNK 231
PRBYTE 79, 248
PRead 209, 210
PREAD 235
PREAD4 235
P register 35
PRERR 253
PRHEX 79, 248
Printer command 40
printer mode 83
  command character 87
  commands 88-90
printer port 300
PRNTAX 79, 230
PRNTX 231
PRNTYX 230
processor status
  changing 61
  register 37
  system interrupt handler 181
ProDOS 43, 70, 110, 114, 115, 130
ProDOS 8 117, 220
ProDOS 16 117, 220
program bank register 17, 35
  system interrupt handler 181
program counter 51
program operation levels 4
program register, changing 61
prompt 74
PROMPT 247
prompt character
  (*) 20, 26, 74
  (]) 74
  (!) 52, 74
  (>) 74
  (?) 74
PRO16MLI 274
pseudoregisters 8, 16
PStatus 209, 210
PWREDUP 259
PWrite 209, 210
PWRUP 234

Q

Q register 36
Quagmire register 16, 36
Quagmire state, changing 62
quarter-second timer interrupts 180
question mark (?) 74
quit 306
Quit Monitor command 43, 65
quotation mark (") 34, 52

R

RAM disk 17, 110, 114, 117, 234, 303
random-number generator 72
R command 90
RdAddr 146
RDCHAR 246
RDKEY 70, 71, 79, 244
RDKEY1 245
READ 253
Read Address Field 139
Read Available Character Sets command 193
Read Available Keyboard Layouts command 193
ReadBlock call 5, 126
Read call 132-133
Read and Clear Error Byte command 192
Read Configuration Bytes command 192
ReadData 146
Read Microcontroller Memory command 191
Read Modes Byte command 191
READMOUSE 183, 203, 209, 212
read-only memory 20
Receive Bytes command 194
Recharge routine 97, 98
REGDSP 235
register addresses, mouse 200
register-display command 22
register-modification commands 38
registers 8, 12-18, 35-38
  examining 60
  examining and changing 36-38
  restoring 66
RESERVED 156
RESET 177, 178, 234
Reset ADB command 194
ResetHook 140

Reset Keyboard Microcontroller command 188
ResetMark 141
Reset the System command 193
RESTORE 254
Resume command 50, 179
return from subroutine (RTS) 49, 65
return from subroutine long (RTL) 14
Retype key 75
ROM (read-only memory) 20
ROM disk 17, 110, 114, 117, 234
  driver 152-155
  passing parameters 152-153
  ROM for 154-155
RTBL 235
RTL (return from subroutine long) 14
RTS (return from subroutine) 49, 65

S

SAVE 254
scan-line interrupts 180
Scrap Manager 180
screen holes 203
SCRN 79, 228
SCROLL 243
SCSI (Small Computer System Interface) 115
Seek 139, 147
Send ADB Keycode command 193
Send command 97
SendQueue 97, 98, 103
SendReset 6
serial-port firmware 5, 81-108
  background printing 97-98
  buffering 95-96
  compatibility 82
  error handling 95
  extended interface 99
  handshaking 84-85
  interrupt notification 96-97
  operating commands 86-92
  operating modes 83
  programming 92-94
serial-port interrupts 180, 183-184
SERVEMOUSE 202, 209, 212
SetAddress 143
SETCOL 79, 225, 226, 228
Set Configuration Bytes command 190
SetDTR 105
SETGR 236
SetHook 138-139
SetInBuffer 95, 102
SetInterleave 141
SetIntInfo 96, 106, 184
SETINV 251
SETKBD 251
SetMark 140-141
SetModeBits 95, 97, 100-101
Set Modes command 189
SETMOUSE 209, 211
SETNORM 251
SetOutBuffer 95, 97, 102
SETPWRC 237
SetSCC 105
SetSides 141
SETTXT 236
SETVBLCNTS 209
SETVID 252
SETWND 236
SETWND2 237
shadowing 308, 310
Shadow register 16
6805 AppleMouse microprocessor card 213
6502 microprocessor 8
65C816 assembly language 54
65C816 microprocessor 8-9
  Apple Desktop Bus microcontroller 186
  emulation mode 14
  execution speeds 9
  indexed instructions 17
  modes 9
slash (/) 22, 40
SLOOP 234
slots 304
Small Computer System Interface (SCSI) 115
SmartPort 110
  assignment of unit numbers 117-119, 157-158
  call parameters 116
  control flow 159-165
  Disk II interactions 158
  dispatch address 115
  error codes 156
  extended commands 137
  issuing a call 120-121
  locating 114-115
  read protocol 161
  standard commands 136
  write protocol 162
SmartPort bus 133, 157-165
  packet contents 164
  packet format 163
SmartPort calls 121-137
  device-specific 138
  specific to Apple 3.5 disk drive 138-141
  specific to UniDisk 3.5 142-143
SmartPort firmware 5, 17, 113-165
SOFTEV 259
soft switches 277-290
sound 303
Speed register 16
S register 11, 35
SRQ 180
SSC (Apple Super Serial Card) 82
stack 15
stack pointer 13-15, 35
  changing 61
STARTTIME 209
Status calls 121-125
status code error 122
status register 56
Step command 50, 66
STEPVECTOR 271
STORADV 240
Store command 44
subtraction, 32-bit 42
Super Hi-Res display 8, 9-10
symbol table 291, 292
Sync command 191
SYSDMGRV 265
system interrupts 175-180

system Monitor
  command syntax 21
  command types 21-24
  creating commands 47
  80-column mode 25-26
  filling memory 45
  firmware 4, 19-67
  40-column mode 25
  invoking 20
  memory commands 25-34
  miscellaneous commands 39-43
  multiple commands 44
  repeating commands 46

T

tabbing 92
TABV 237
Talk 6
terminal mode 83
  command character 91-92
TEXT2COPY 232
text display, changing 63
text window 80
time
  changing 64
  displaying 40, 63
TIMEDATA 209
TOBRAMSETUP 273
TOCTRL.PANEL 273
toolbox routines 43
tool error number 67
Tool Locator 43, 55, 67
TOPRINTMSG8 273
TOPRINTMSG16 274
TOREADBR 272
TOREADTIME 273
TOSUB 247
TOTEXTPG2DA 274
TOWRITEBR 272
TOWRITETIME 272
Trace command 50, 66
TRACEVECTOR 271
Transmit num Bytes command 194
Transmit Two Bytes command 195

U

UDISPATCH1 264
UDISPATCH2 264
underscore (_) 41, 67, 83
UniDiskStat 143
UniDisk 3.5 110, 117, 133, 135
  internal functions 144-145
  internal routines 146-149
  memory allocation 150-151
  SmartPort calls 142-143
UP 241
User command 47
user vector 65
USRADR 259

V

vectors 70, 149, 258-275
Verify 33, 45, 59, 140, 148
VERSION 238
vertical blanking signal 180, 183
video firmware 5, 69-80
VIDOUT 240
VIDWAIT 238
VLINE 79, 226
VTAB 241
VTABZ 79, 241

W

WAIT 243
warm start 65, 112, 178
windows 219
WRITE 253
WriteBlock call 5, 127
Write call 134-135
WriteData 147
Write Data Field 139
Write Microcontroller Memory command 191
Write Track 139-140
WriteTrk 148

X

XBA 18
X command 50
XFER 261
x flag 37
XOFF 85, 89, 95
XON 85, 89, 95
X register 35, 74, 98, 121
  changing 60
  system interrupt handler 181

Y

Y register 35, 98, 121
  changing 60
  system interrupt handler 181

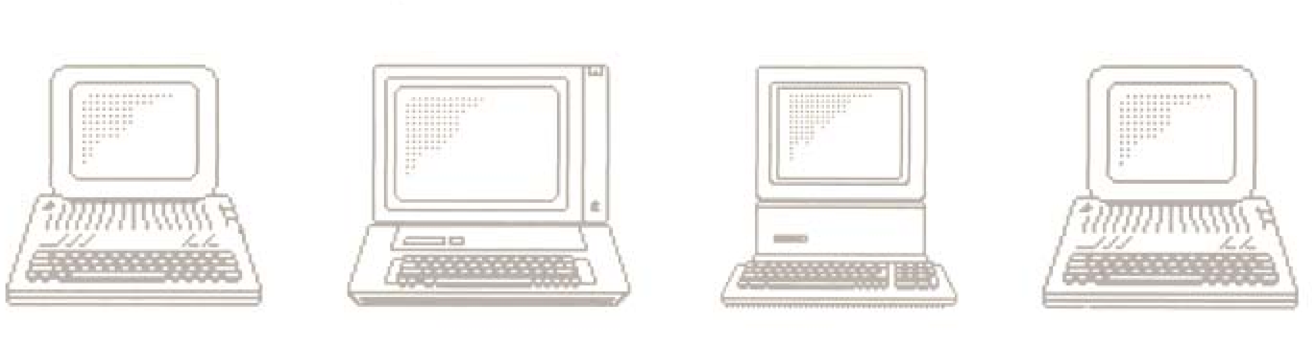
Z

Zap command 34, 87
zero page 12, 15
ZIDBYTE 239
ZIDBYTE2 238
Zilog Serial Communications Controller chip 82
ZMODE 257

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