Apple IIGS Firmware Reference Manual

Beta Draft
10/24/86

Writer: Rob Peck
Apple Technical Publications

This document contains preliminary information. It does not include

- *final editorial corrections*
- *final art work*
- *an index*

It may not include final technical changes

[Additional credit lines as needed]

Simultaneously published in the United States and Canada.

CONTENTS

20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010

[additional credit lines as needed]

Simultaneously published in the United States and Canada.

Chapter 1

Apple IIGS Firmware Overview

This chapter gives a brief overview of the Apple IIGS firmware and how it relates to the rest of the sytem software.

Introduction

The Apple IIGS firmware is composed of various kinds of routines that are stored in the system's read-only memory (ROM). The Apple IIGS firmware routines provide the means to adapt and control the Apple IIGS system.

The following is a list of the Apple IIGS firmware routines that are covered in this manual:

- System Monitor firmware
- Video firmware (I/O routines)
- Serial Port firmware (for character-at-a-time I/O)
- Disk Support firmware (slot 6 support)
- SmartPort firmware (for block device I/O)
- Interrupt Handler
- Apple Desktop Bus microcontroller
- Mouse firmware

A word about other Apple IIGS firmware

The above topics do not comprise the whole body of Apple IIGS firmware. The Apple IIGS ROM contains other firmware, important enough to warrant separate manuals: the Apple IIGS Tools (described in detail in the *Apple IIGS Tools Reference Manual*), Applesoft BASIC (described in the Applesoft BASIC Reference Manual), and AppleTalk (described in the *AppleTalk Manual*).

Apple IIGS Tools

The Apple IIGS Tools provide a means of constructing application programs that conform to the standard user interface. By offering a common set of routines that every application can call to implement the user interface, the tools not only ensure familiarity and consistency for the user, but also help to reduce the application's code size and development time.

AppleTalk

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

The role of firmware in the Apple IIGS system

The firmware is an interface to the system's hardware that controls the display, the mouse, serial I/O, and disk drives. Moreover, firmware programs, such as the Monitor and Control Panel, work directly with the system memory.

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

Levels of program operation

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

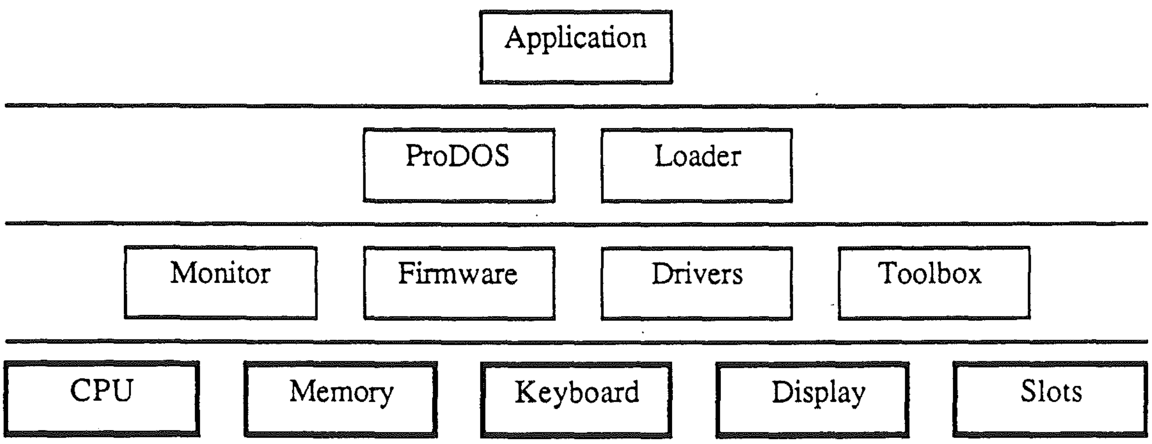


Figure 1-1. Levels of program operation

Apple IIGS firmware overview

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

System Monitor firmware

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

Video firmware

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

Serial Port firmware

The Apple IIGS Serial Port firmware facilitates serial communication with external devices, such as printers and modems. The serial firmware provides support for such things as optional hardware and/or software handshaking, and background printing. There are two serial ports, either of which can be configured as a printer or a modem port.

Disk Support firmware

The Apple IIGS Disk II firmware is a disk-support subsystem. It uses a built-in Integrated Woz Machine (IWM) chip and accommodates Disk II (Duodisk or Unidisk) drives. Slot 6 is the standard Disk II support slot.

SmartPort firmware

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

Interrupt Handler

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

Apple Desktop Bus (ADB) microcontroller

The ADB Microcontroller is used to receive information from peripheral units attached to the Apple Desktop Bus (ADB). The ADB microcontroller *polls* the internal keyboard, sensing key-up and key-down events as well as control keys and optionally buffers them for later access by the 65816. In addition, the ADB uC acts as host for the ADB peripherals, such as the detachable keyboard and mouse. The ADB Microcontroller has its own built-in set of instructions, including Talk, Listen, SendReset, and Flush.

Mouse firmware

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

Diagnostic routines

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

Chapter 4

Video Firmware

This chapter describes the routines and command sequences that you use to produce and control the video output of text to the Apple IIGS video screen.

Introduction

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

Using the video firmware I/O routines you can

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

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

Standard I/O links

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

> **Marginal Gloss:** DOS and ProDOS maintain their own links to the standard I/O subroutines.

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

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

Standard input routines

The Apple IIGS firmware includes two different subroutines for reading from the keyboard. One subroutine is named RDKEY, which stands for *read key*. RDKEY calls the character input subroutine KEYIN (or BASICIN when the 80-column firmware is active) and accepts one character at a time from the keyboard.

The other subroutine is named GETLN, which stands for *get line*. By making repeated calls to RDKEY, GETLN accepts a sequence of characters terminated with a carriage return. GETLN also provides on-screen editing features.

RDKEY input subroutine

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

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

KEYIN/BASICIN input subroutines

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

If the 80-column firmware is inactive, KEYIN displays a cursor by storing a checkerboard block in the cursor location, then storing the original character, then the checkerboard again. If the 80-column firmware is active, BASICIN displays a steady inverse space (rectangle) as a cursor. In an additional operating mode, escape mode, the cursor displayed is an inverse video plus sign (+). This indicates that escape mode is active.

> **Marginal Gloss:** See the section titled "Cursor control" later in this chapter for more information about the escape mode.

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

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

Escape codes

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

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

Cursor control

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

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

Table 4-1. Escape codes

Cursor control	**Function**
ESC A	Moves the cursor right one space; exits from escape mode
ESC B	Moves the cursor left one space; exits from escape mode
ESC C	Moves the cursor down one line; exits from escape mode
ESC D	Moves the cursor up one line; exits from escape mode
Cursor control/ Entering escape mode	**Function**
ESC I (or ESC up arrow)	Moves the cursor up one line and remains in escape mode
ESC J (or ESC left arrow)	Moves the cursor left one space and remains in escape mode
ESC K (or ESC right arrow)	Moves the cursor right one space and remains in escape mode
ESC M (or ESC down arrow)	Moves the cursor down one line and remains in escape mode
Screen/line clearing	**Function**
ESC @	Clears the window and moves the cursor to its home position (upper-left corner of screen); exits from escape mode
ESC E	Clears to the end of the line; exits from escape mode
ESC F	Clears to the bottom of the window; exits from escape mode

Screen format control	Function
ESC 4	Switches from 80-column display to 40-column display if 80-column firmware is active; sets links to BASICIN and BASICOUT; restores normal window size; exits from escape mode
ESC 8	Switches from 40-column display to 80-column display by enabling the 80-column firmware; sets links to BASICIN and BASICOUT; restores normal window size; exits from escape mode
ESC-CONTROL-D	Disables control characters; only carriage returns, line feeds, bells, and backspaces have effects when printing is performed
ESC-CONTROL-E	Reactivates control characters
ESC-CONTROL-Q	If 80-column firmware is active, deactivates the 80-column firmware; sets links to KEYIN and COUT1; restores normal window size; exits from escape mode

GETLN input subroutine

Programs often need strings of characters as input. While it is possible to call RDKEY repeatedly to get several characters from the keyboard, there is a more powerful subroutine you can use to get an edited line of characters. This routine is named *GETLN*, which stands for *get line*; GETLN starts at location $FD6A. Using repeated calls to RDKEY, GETLN accepts characters from the standard input subroutine—usually KEYIN—and puts them into the input buffer located in the memory page from $200 to $2FF.

Marginal Gloss: GETLN also provides the user with onscreen editing and control features described in the next section, "Editing with GETLN".

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

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

Table 4-2. Prompt characters

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

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

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

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

Editing with GETLN

The subroutine GETLN provides the standard onscreen editing features used with BASIC interpreters and the Monitor. Any program that uses GETLN for reading the keyboard has these features.

> **Marginal Gloss:** For an introduction to editing with GETLN, refer to the Applesoft Tutorial.

Cancel line

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

Backspace

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

Retype

The function of the Retype key (-->) is complementary to the function of the Backspace key. When you press Retype, GETLN picks up the character at the display position just as if it had been typed on the keyboard. You can use this procedure to pick up characters that you have just deleted by backspacing across them. You can use the backspace and retype functions with the cursor motion functions to edit data on the display.

> **Marginal Gloss:** For more information about cursor motion, see the section "Cursor control" earlier in this chapter.

Keyboard input buffering

In versions of the Apple II prior to the Apple IIGS, if a keystroke happened while your program was processing the previous keystroke, it was possible to lose characters that the user was typing into your program. The Apple IIGS allows you to enable keyboard input buffering to prevent the loss of keystrokes.

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

Standard output routines

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

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

COUT/BASICOUT subroutines

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

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

Control characters with COUT1 and C3COUT1

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

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

Control character	Action taken by COUT1
CONTROL-G	Produces user-defined tone (Control Panel menu)
CONTROL-H	Backspace
CONTROL-J	Line feed
CONTROL-M	Return
CONTROL-^ {char}	First character output after CONTROL-^ becomes the new cursor. If the DELETE key is the first character, the default prompt is restored.

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

Control character	Action taken by C3COUT1
CONTROL-E	Turns cursor off
CONTROL-F	Turns cursor on
CONTROL-G	Produces user-defined tone (Control Panel menu)
CONTROL-H	Backspace
CONTROL-J	Line feed
CONTROL-K	Clears from cursor position to the end of the screen

CONTROL-L	Form feed
CONTROL-M	Carriage return
CONTROL-N	Changes to normal display format
CONTROL-O	Changes to inverse display format
CONTROL-Q	Sets 40-column display
CONTROL-R	Sets 80-column display
CONTROL-S	Stops listing characters until another key is pressed
CONTROL-U	Deactivates enhanced Video firmware
CONTROL-V	Scrolls the display down one line, leaving the cursor in the current position
CONTROL-W	Scrolls the display up one line, leaving the cursor in the current position
CONTROL-X	Disables MouseText character display and uses inverse uppercase
CONTROL-Y	Home
CONTROL-Z	Clears the line on which the cursor resides
CONTROL-[	Enables MouseText character display by mapping inverse uppercase characters to MouseText characters
CONTROL-\	Moves cursor position one space to the right; from edge of window, moves to left end of next line
CONTROL-]	Clears from cursor position to the right end of the line
CONTROL-_	Moves cursor up one line with no scroll
CONTROL-^	Goes to XY; using the next two characters minus 32 as one-byte X and Y values, moves the cursor to CH=X, CV=Y (Pascal)
CONTROL-^ {char}	First character output after CONTROL-^ becomes the new cursor. If the DELETE key is the first character, the default prompt is restored. *Note:* This only works when using BASIC links, not Pascal output links.

Inverse and flashing text

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

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

Table 4-5. Text format control values

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

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

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

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

Other firmware I/O routines

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

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

Table 4-6. A partial list of other Monitor firmware I/O routines

Location	Name	Description
$FC9C	CLREOL	Clears to end of line from current cursor position
$FC9E	CLEOLZ	Clear to end of line using contents of Y register as cursor position
$FC42	CLREOP	Clears to bottom of window
$F832	CLRSCR	Clears the low-resolution screen
$F836	CLRTOP	Clears the top 40 lines of the low-resolution screen
$FDED	COUT	Calls the output routine whose address is stored in CSW, normally COUT1
$FDF0	COUT1	Displays a character on the screen
$FD8E	CROUT	Generates a carriage return
$FD8B	CROUT1	Clears to end of line and then generates a carriage return
$FD6A	GETLN	Displays the prompt character; accepts a string of characters by means of RDKEY
$F819	HLINE	Draws a horizontal line of blocks
$FC58	HOME	Clears the window and puts the cursor in the upper left corner of the window
$FD1B	KEYIN	With 80-column firmware inactive, displays checkerboard cursor; accepts characters from keyboard
$F800	PLOT	Plots a single low-resolution block on the screen
$F94A	PRBL2	Sends 1 to 256 blank spaces to the output device
$FDDA	PRBYTE	Prints a hexadecimal byte
$FDE3	PRHEX	Prints 4 bits as a hexadecimal number

$F941	PRNTAX	Prints the contents of A and X in hexadecimal format
$FD0C	RDKEY	Displays blinking cursor; goes to standard input routine, normally KEYIN or BASICIN
$F871	SCRN	Reads color of a low-resolution block
$F864	SETCOL	Sets the color for plotting in low-resolution block
$FC24	VTABZ	Sets the cursor vertical position
$F828	VLINE	Draws a vertical line of low-resolution blocks

The text window

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

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

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

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

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

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

After you have changed the text window boundaries, no changes occur to the screen appearance until you send the next character to the screen.

Chapter 6

Disk II Support

This chapter describes the Apple IIGS Disk II Support Firmware. Several different types of disks can be attached to the Apple IIGS, some of which contain built-in intelligence. This chapter describes the methods by which the Disk II product can be connected to the Apple IIGS.

Introduction

The Apple IIGS Disk Support system, with its built-in Integrated Woz Machine (IWM) chip, accommodates Disk II (DuoDisk and UniDisk drives) and Sony 3.5-inch drives with built-in intelligence (UniDisk 3.5) or without built-in intelligence (Apple 3.5 drives).

Port 6 is the standard Disk II support slot. Disk II boot routines are built into ROM. Disk II routines in DOS, ProDOS, and Pascal operate the same as they do in an Apple II prior to the Apple IIGS.

Port 5 (internal slot 5) controls the intelligent Sony and Apple 3.5 drives as well as the RAM disk. You can attach up to two Disk IIs, two Apple 3.5 drives, and two or more intelligent Sony 3.5-inch drives, depending on IWM output specifications. The disks must be attached as shown in Figure 6-1.

Two Apple 3.5 drives are shown in Figure 6-1. This is the maximum number supported. There may be more than one UniDisk 3.5 where this drive is shown in the figure.

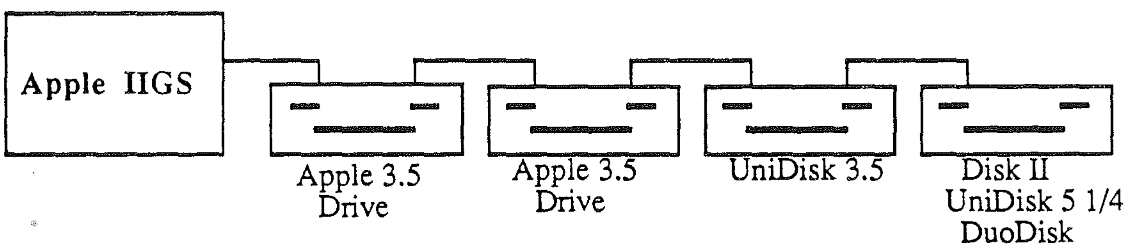


Figure 6-1. Maximum disk drive configuration

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

The port 5 disk interface for UniDisk 3.5 is called *SmartPort.* It consists of an expanded version of the SmartPort software used in the 32K Apple II ROM. SmartPort supports two Apple 3.5 drives, the RAM disk, and UniDisk 3.5, up to a total of 127 combined devices. The SmartPort software is covered in detail in Chapter 7.

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

Table 6-1. Disk I/O port characteristics

Port Number:	I/O port 6 drive 1 I/O port 6 drive 2
Commands:	IN#6 or PR#6 from BASIC, or CALL -151 (to get to the Monitor from BASIC), then 6 Control-P
Initial Characteristics:	All resets except Control-Reset with a valid reset vector pass control to slot 6 drive 1 if this drive is set as the boot device (set through the Control Panel)
Hardware Location:	$C0E0-$C0EF, reserved for Disk II usage
Monitor Firmware Routines:	None
I/O Firmware Entry Points:	$C600 (port 6 boot address) $C65E (read first track, first sector and begin execution of the code found there)
Use Of Screen Holes:	Port 6 main and auxiliary memory screen holes are reserved

Startup

The Apple IIGS has two ways to start up -- a cold start and a warm start. A cold start clears the machine's memory and tries to load an operating system from disk. A warm start stops the current program that is running and leaves the machine in Applesoft with memory and programs intact.

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

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

Assuming you have set the startup device (from the Control Panel) to slot 6, the cold-start routine first sets a number of soft-switches (see Appendix E) and then passes control to the program entry point at $C600. This code turns on the Disk II, Unit 1 device motor, recalibrates the head to track 0, then reads sector 0 from that track. The sector contents are loaded into memory starting at address $0800; then program control passes to $0801. The program loaded depends on the operating system or application program on the disk.

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

A warm start begins when you press Control-Reset, if a valid reset vector exists. Normally, a warm start leaves you in BASIC with memory unchanged. If a program has changed the reset vector the system won't do a warm start; instead, a program may do any number of things. Usually a program either does a cold start or it beeps or it does nothing, leaving you in the currently executing program.

Chapter 7
SmartPort Firmware

Introduction

The SmartPort firmware is an extension to the ProDOS block device driver resident in internal slot 5. It consists of a set of assembly-language routines that supports a series of block or character devices connected to the external disk port on the Apple IIGS. The SmartPort converts calls to a format which is transmitted over the disk port to control intelligent devices, such as the UniDisk 3.5. SmartPort also provides an interface to several non intellegent devices through the use of device specific drivers. Non intelligent devices that are supported on the Apple IIGS through SmartPort include the AppleDisk 3.5, RAM Disk and ROM Disk.

Using the SmartPort

To use the SmartPort interface, a program issues calls in a manner similar to that used for ProDOS Machine Language Interface calls. The topmost level of one of these calls is a JSR to the SmartPort entry point followed by a SmartPort command byte and a pointer to a table which contains the parameters necessary for the call.

Locating SmartPort

You can determine if the SmartPort Interface exists by examining the ProDOS Block Device signature bytes shown below:

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

In addition, you must also verify the existence of the SmartPort signature byte shown below:

$Cn07 = $00

In the above addresses, n = the slot number for which the signature bytes are being examined. All peripheral cards or ports with these signature byte values support both ProDOS block device calls and SmartPort calls. You can examine the SmartPort ID Type byte to obtain more information about any special support that may be built into the SmartPort driver. The SmartPort ID Type byte located at $CnFB has been encoded to indicate the type of devices that can be supported by the SmartPort driver. Note that a driver that supports the Extended SmartPort calls must also support Standard SmartPort calls.

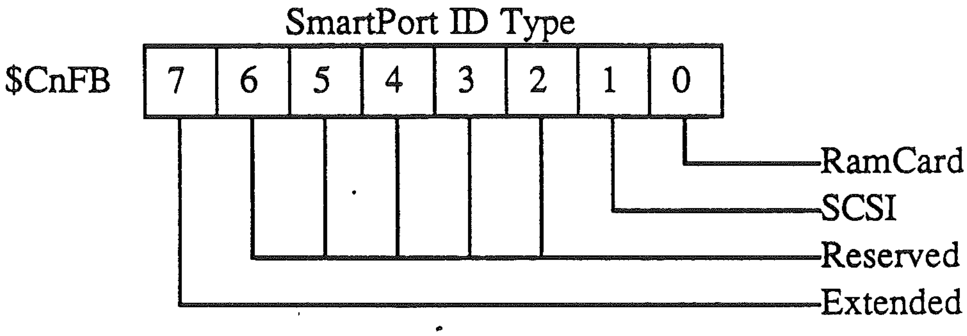


Fig.7-1. SmartPort ID type byte

Locating the dispatch address

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

For example, if you find the signature bytes for the SmartPort interface in slot 5, and $C5FF contains a hexadecimal value of $0A, the ProDOS entry point will be $C50A, and the SmartPort entry point is three larger than $C50A, or $C50D.

SmartPort call parameters

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

Command name:	The name used to identify the SmartPort call
Command number:	A byte value contiguous in memory with the JSR to the SmartPort entry point. A hexadecimal number that specifies the type of SmartPort call. Bit 6 will be cleared to 0 for standard calls or set to 1 for extended calls.
Parameter List Pointer:	A pointer contiguous in memory with the command number that points to the parameter list.
Parameter count:	A hexadecimal byte value that specifies the number of parameters contained in the parameter list.

Unit number:	A hexadecimal byte value that specifies the unit number of the device that the SmartPort call is to direct I/O to or from.
Buffer Address:	A pointer to memory that will be used in the I/O transaction. For standard SmartPort calls this will be a word wide pointer referencing memory in bank zero. For extended calls, the pointer will be a longword referencing memory in any bank.
Block number:	A number specifying the block address used in an I/O transaction with a block device. For standard SmartPort calls this parameter is 24 bits wide. For extended calls this parameter is 32 bits wide.
Byte count:	This parameter is used to specify the number of bytes to be transferred between memory and the device. This parameter is 16 bits wide.
Address pointer:	This parameter is used to specify an address within the device.

SmartPort assignment of unit numbers

The Unit number is part of every parameter list. The unit number specifies which device connected to the SmartPort will respond to the command you are giving. Calls which allow you to reference the SmartPort itself use a unit number of zero. Only the status, init, and control calls may be made to unit zero. The Apple IIGS assigns unit numbers to devices in ascending order starting with a unit number of $01. Devices are assigned unit numbers starting with the RamDisk, RomDisk, AppleDisk 3.5 and finally intelligent devices such as the UniDisk3.5.

Dynamic allocation of device unit numbers

The Apple IIGS implementation of SmartPort interacts with the control panel selection of boot devices. For any given port, a boot can only occur from the first device logically connected to that port. SmartPort support is provided to allow booting from any of three types of devices:

Ram Disk
Rom Disk
Disk type device (AppleDisk 3.5 or UniDisk 3.5)

Depending on the devices that are connected to the SmartPort, the selected boot device may not be the first logical device in the chain. In order to boot from the selected device, the selected device must be moved logically to the first unit in the device chain. This means that all devices that were previously ahead of the selected boot device must now be moved logically so that they are now located behind the selected boot device.

The initialization call handles assignments of unit numbers in a two stage process. The first stage assigns unit numbers as described above. The second stage remaps the units so that the selected boot device is always the first logical device in the chain. If 'scan' is selected as the boot option in the control panel, SmartPort will place the first physical disk device as the first logical device in the device chain.

Remapping of devices has some interresting implications when running with ProDOS 1.1.1. Current implementations of ProDOS only support two devices per port or slot. If more than two devices are logically connected to the device chain, devices beyond the second device can not be accessed with ProDOS 1.1.1. The interim version of ProDOS for Apple IIGS that will be available before ProDOS'16 is ProDOS 1.2. ProDOS 1.2 will support up to four devices on SmartPort. ProDOS 1.2 will map the to two devices beyond the second device in the device chain so that the additional devices will appear as if they are connected to slot 2. Due to the affects of the logical remapping that places the boot device as the first device in the chain, the relationship of devices and slots with ProDOS 1.2 varies with the boot configuration as set by the control panel.

Interaction between the control panel and the logical assignment of unit numbers to devices on the SmartPort device chain will also be visable with ProDOS'16, however all the devices will appear in slot 5. No remapping of units to slot 2 will be neccessary with ProDOS'16 since ProDOS'16 will support more than two devices per port or slot.

Several illustrations follow, showing remapping of devices based on the selected boot device vs. the device configuration. Only a few of the possible derivations of the device mapping are shown.

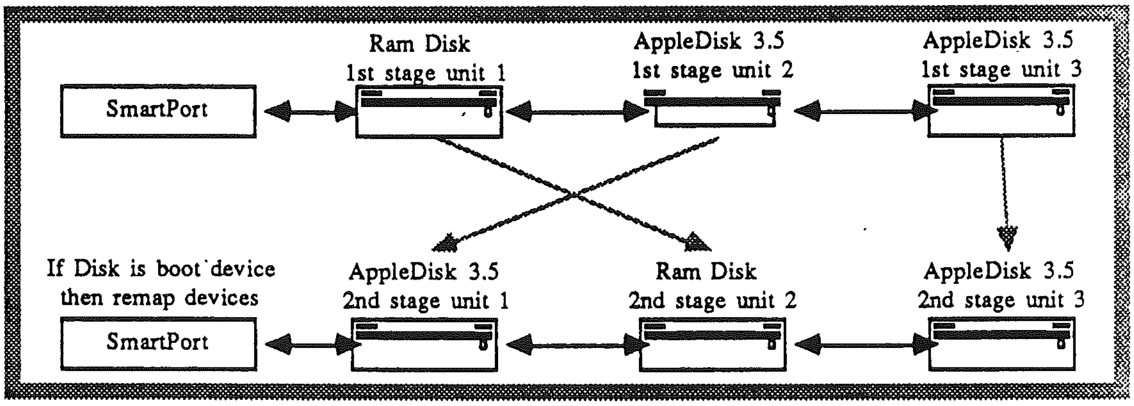


Figure 7-2. Device mapping - Derivation 1.

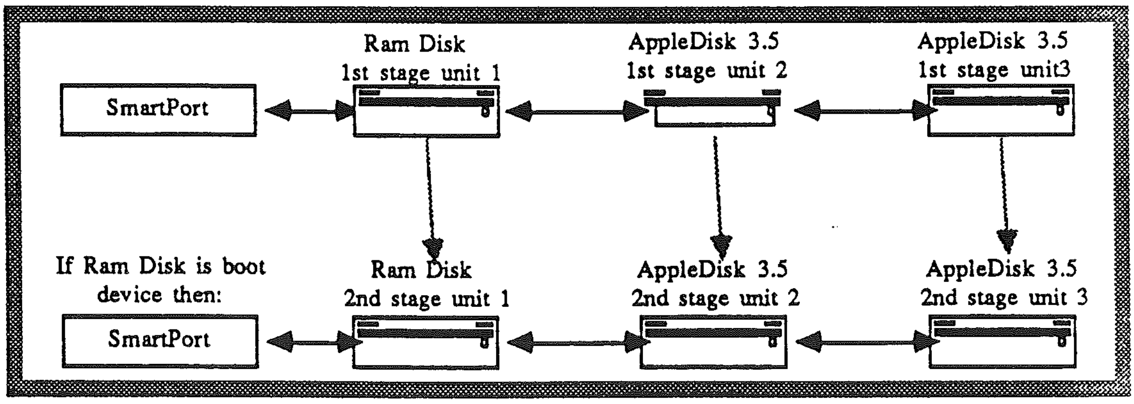


Figure 7-3. Device mapping - Derivation 2.

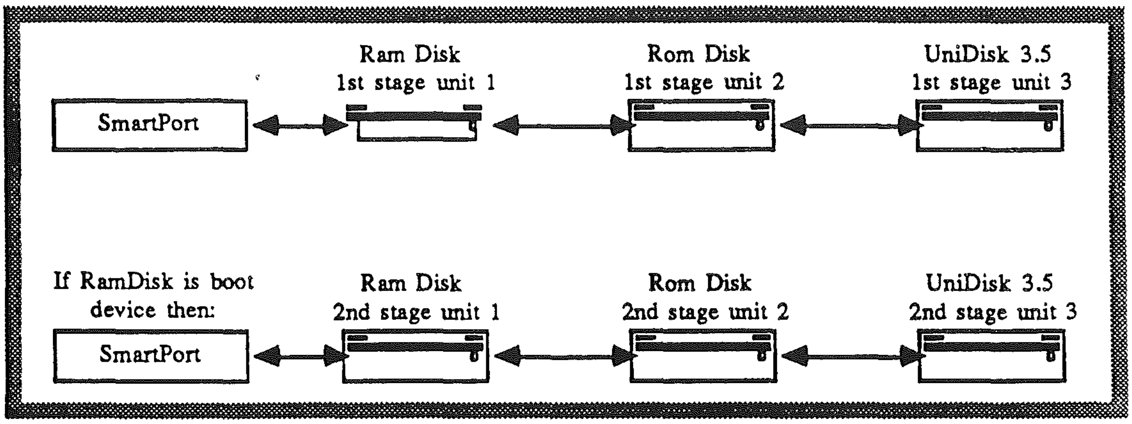


Figure 7-4. Device mapping - Derivation 3.

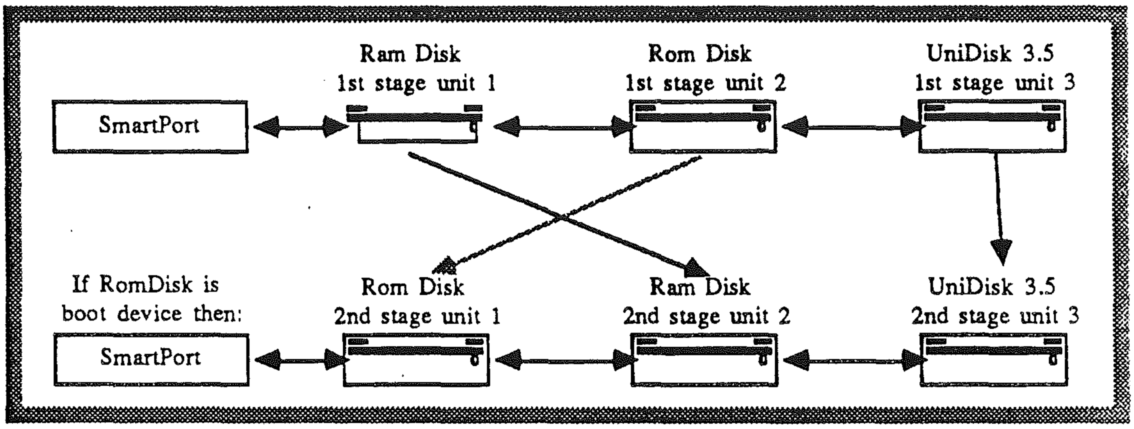


Figure 7-5. Device mapping - Derivation 4.

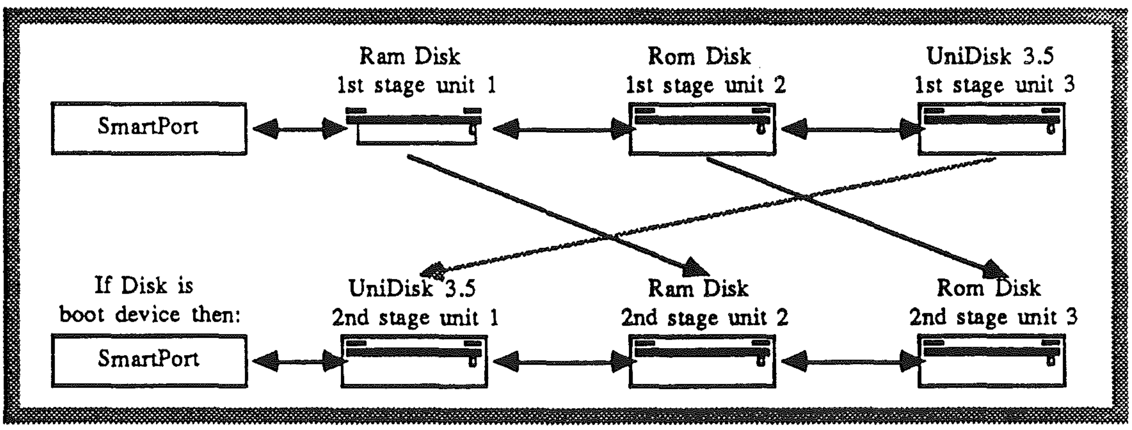


Figure 7-6. Device mapping - Derivation 5.

Issuing a call to SmartPort

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

When making Standard SmartPort calls, the pointer following the SmartPort command byte is a word wide pointer to a parameter list in bank zero. When making Extended SmartPort calls, the pointer is a longword pointer to a parameter list in any memory bank.

There are several constraints on the use of the SmartPort.

- the stack usage is 30-35 bytes. Programs should allow 35 bytes of stack space for each call.

- the SmartPort cannot generally be used to put anything into absolute Zero Page locations. Absolute Zero Page is defined as Direct Page when the Direct Register is set to $0000.

- SmartPort can only be called from Apple II emulation mode. This means that the emulation flag in the 65C816 processor status byte must be set to a 1, and the Direct Page Register and Data Bank Register must both be set to zero. Native mode programs wishing to call SmartPort must switch to emulation mode prior to making the SmartPort call. You must assure that your complete operating environment is carefully preserved before making the call and restored after making the call. You can find additional details about the environment in Chapter 2, Notes for Programmers.

This is an example of a standard SmartPort call:

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

This is an example of a extended SmartPort call:

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

Upon completion of the call, execution returns to the RTS address plus three for a standard call, or the RTS address plus five for an extended call (the BCS statement in the examples). If the call was successful, the C flag is cleared, and the A register is set to 0. If the call was unsuccessful, the C flag is set and the A register contains the error code. The complete register status upon completion is summarized below.

REGISTER STATUS ON RETURN FROM SMARTPORT						
	65816 Status byte N V 1 B D I Z C	Acc	Xreg	Yreg	PC	SP
Successful Nonextended Call	X X 1 X 0 U X 0	0	n	n	JSR+3	U
Successful Extended Call	X X 1 X 0 U X 0	0	n	n	JSR+5	U
Unsuccessful Nonextended Call	X X 1 X 0 U X 1	Error	X	X	JSR+3	U
Unsuccessful Extended Call	X X 1 X 0 U X 1	Error	X	X	JSR+5	U
(Note: X = undefined, U = unchanged, n = undefined for tranfers to the device or number of bytes transferred when the transfer was from the device to the host)						

Figure 7-7. Register status on return from SmartPort

SMARTPORT STATUS CALL

	Standard call	Extended call
CMDNUM	$00	$40
CMDLIST	parameter count	parameter count
	unit number	unit number
	status list pointer (low byte)	status list pointer (low byte, low word)
	status list pointer (high byte)	status list pointer (high byte, low word)
	status code	status list pointer (low byte, high word)
		status list pointer (high byte, high word)
		status code

This call returns the status information about a particular device or about the SmartPort itself. This chapter lists status calls that return general information. Device specific status calls can be implemented by a device for diagnostic or other information. Device specific calls would have to be implemented with a status code of $04 or greater.

On return from a status call, the X and Y registers contain a count of the number of bytes tranferred to the host. X contains the low byte of the count, while Y contains the high byte value of the count.

Required parameters

Parameter Count: byte value = $03

unit number: 1 byte value in the range : $00, $01 to $7E

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

status list pointer:	Standard	Extended
	Word pointer (bank $00)	Longword pointer

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

status code: 1 byte value in range of $00 - $FF

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

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

Although devices must respond to the status requests listed above, the device may not be able to support the request. In this case, the device should return an Invalid Status Code error ($21).

Statcode = $00

The device status consists of four bytes. The first is the general status byte:

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

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

Statcode = $01

The device control block or DCB is device dependent. The DCB is typically used to control various operating characteristics in a device. The DCB is set with the corresponding control call. The first byte will be the number of bytes in the control block. A value of $00 returned in this byte should be interpreted as a DCB length of 256, while a value of $01 would be a DCB length of 1 byte. The length of the DCB will always be in the range of 1 to 256 bytes excluding the count byte.

Statcode = $02

There are currently no character devices implemented for use on the SmartPort, and therefore the Newline status is presently undefined.

Statcode = $03

This call returns the device information block or DIB. It contains information identifying the device, its type, and various other attributes. The returned status list has the following form:

STATLIST:	Standard call	Extended call
	Device Status byte	Device Status byte
	Block Size (low byte)	Block Size (low byte, low word)
	Block Size (mid byte)	Block Size (high byte, low word)
	Block Size (high byte)	Block Size (low byte, high word)
	ID String length	Block Size (high byte, high word)
	ID String (16 bytes)	ID String length
	Device Type byte	ID String (16 bytes)
	Device Subtype byte	Device Type byte
	Version word	Device Subtype byte
		Version word

The Device Status is a one byte field which is the same as the general status byte returned in the device status call (statcode = $00). The Block Size field is the same as the Block Size field retuned in the device status call. The ID String consists of a single byte prefix indicating the number of ASCII characters in the ID string. This is followed by a 16 byte field containing an ASCII string identifying the device. The most significant bit of each ASCII character will be set to zero.

If the ASCII string consists of less than sixteen characters, ASCII spaces are used to fill the unused portion of the string buffer. The Device Type and Device Subtype fields are single byte fields. Several bits encoded within the DIB subtype byte have been defined to indicate whether a device supports the extended SmartPort interface, disk switched errors or removable media. A breakdown of the DIB subtype byte is shown below:

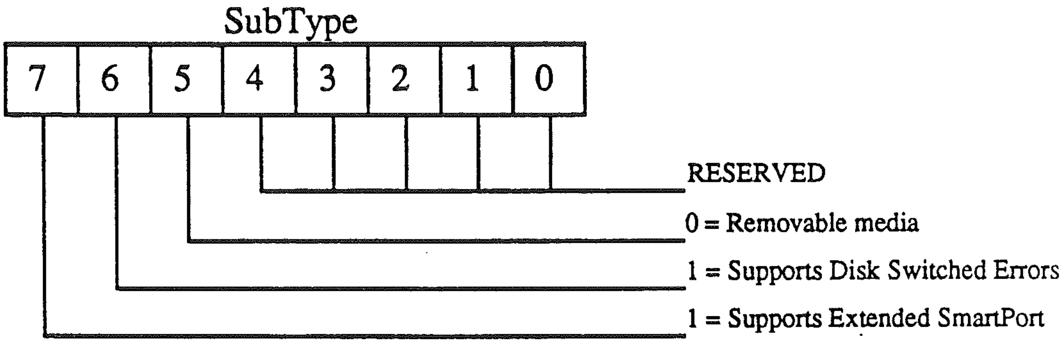


Figure 7-8. SmartPort device subtype byte

Several device types and subtypes have been assigned to existing SmartPort devices. These types and subtypes are shown below:

TYPE	SUBTYPE	DEVICE
$00	$00	Apple II Memory Expansion Card
$01	$00	UniDisk 3.5
$01	$C0	AppleDisk 3.5
$03	$E0	Apple II SCSI with non-removable media

Undefined SmartPort devices may implement the following types and subtypes:

TYPE	SUBTYPE	DEVICE
$02	$20	Hard Disk
$02	$00	Removable Hard Disk
$02	$40	Removable Hard Disk supporting disk switched errors
$02	$A0	Hard Disk suporting extended calls
$02	$C0	Removable Hard Disk suporting extended calls & disk switched errors
$02	$A0	Hard Disk suporting extended calls
$03	$C0	SCSI with removable media

The Firmware Version field is a two byte field consisting of a number that indicates the firmware version.

SmartPort driver status

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

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

The number of devices field is a single byte field that indicates to the caller the total number of devices connected to this slot or port. This number will always be in the range of 0 to 127.

Possible errors

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

SMARTPORT READ BLOCK CALL

	Standard call	**Extended call**
CMDNUM	$01	$41
CMDLIST	parameter count	parameter count
	unit number	unit number
	data buffer pointer (low byte)	data buffer pointer (low byte, low word)
	data buffer pointer (high byte)	data buffer pointer (high byte, low word)
	block number (low byte)	data buffer pointer (low byte, high word)
	block number (middle byte)	data buffer pointer (high byte, high word)
	block number (high byte)	block number (low byte, low word)
		block number (high byte, low word)
		block number (low byte, high word)
		block number (high byte, high word)

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

Required parameters

<u>parameter count:</u> byte value = $03

<u>unit number</u>: 1 byte value in the range: $01 to $7E

<u>data buffer pointer:</u>	<u>Standard Call</u>	<u>Extended Call</u>
	word pointer (bank $00)	LongWord pointer

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

<u>block number:</u>	<u>Standard Call</u>	<u>Extended Call</u>
	3 byte number	4 byte number

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

Possible errors

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

SMARTPORT WRITE BLOCK CALL

	Standard call	**Extended call**
CMDNUM	$02	$42
CMDLIST	parameter count	parameter count
	unit number	unit number
	data buffer pointer (low byte)	data buffer pointer (low byte, low word)
	data buffer pointer (high byte)	data buffer pointer (high byte, low word)
	block number (low byte)	data buffer pointer (low byte, high word)
	block number (middle byte)	data buffer pointer (high byte, high word)
	block number (high byte)	block number (low byte, low word)
		block number (high byte, low word)
		block number (low byte, high word)
		block number (high byte, high word)

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

Required parameters

parameter count: byte value = $03

unit number: 1 byte value in the range: $01 to $7E

data buffer pointer:	Standard Call	Extended Call
	word pointer (bank $00)	LongWord pointer

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

block number:	Standard Call	Extended Call
	3 byte number	4 byte number

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

Possible errors

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

SMARTPORT FORMAT CALL

	Standard call	Extended call
CMDNUM	$03	$43
CMDLIST	parameter count	parameter count
	unit number	unit number

This call formats a block device. It should be noted that the format done by this call is NOT linked to any operating system: it simply prepares all blocks on the medium for reading and writing. Operating system specific catalog information such as bit maps and catalogs are not laid down by this call.

Implementation of the format call

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

Required parameters

parameter count:	byte value = $01
unit number:	byte value in the range: $01 to $7E

Possible errors

$06	BUSERR	Communications error
$27	IOERROR	I/O Error
$28	NODRIVE	No Device Connected
$2B	NOWRITE	Disk Write Protected
$2F	OFFLINE	Device off line or no disk in drive

SMARTPORT CONTROL CALL

	Standard call	**Extended call**
CMDNUM	$04	$44
CMDLIST	parameter count	parameter count
	unit number	unit number
	control list pointer (low byte)	control list pointer (low byte, low word)
	control list pointer (high byte)	control list pointer (high byte, low word)
	control code	control list pointer (low byte, high word)
		control list pointer (high byte, high word)
		control code

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

Required parameters

parameter count: byte value = $03

unit number: byte value in the range: $00 to $7E

control list:	Standard Call	Extended Call
	word pointer (bank $00)	Longword pointer

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

control code: byte value
Range: $00-$FF

This is the number of the control request being made. This number and function is device specific, with the exception that all devices must reserve the following codes for specific functions.

Code	**Control function**
$00	Reset the device.
$01	Set device control block
$02	Set newline status (character devices only)
$03	Service device interrupt

Code = $00

Performs a soft reset of the device. Generally returns 'housekeeping' values to some reset value.

Code = $01

This control call is used to set the device control block. Devices generally use the bytes in this block to control global aspects of the device's operating environment. Since the length is device dependent, the recommended way to set the DCB is to first read in the DCB (with the STATUS call), alter the bits of interest, and then write out the same string with this call. The first byte is the length of the DCB (excluding the byte itself). A value of $00 in the length byte corresponds with a DCB size of 256 bytes, while a count value of $01 corresponds with a DCB size of 1 byte. A count value of $FF corresponds with a DCB size of 255 bytes.

Possible errors

$06	BUSERR	Communications error
$21	BADCTL	Invalid control code
$22	BADCTLPARM	Invalid parameter list
$30-$3F	UNDEFINED	Device specific error

SMARTPORT INIT CALL

	Standard call	**Extended call**
CMDNUM	$05	$45
CMDLIST	parameter count	parameter count
	unit number	unit number

This call provides the application with a way of resetting the SmartPort.

Required parameters

<u>parameter count:</u> byte value = $01

<u>unit number:</u> byte value = $00

The SmartPort will go through it's initialization sequence, hard resetting all devices and sending each their device numbers. This call may not be made to a specific unit, rather it must be made to the SmartPort as a whole. This call should not be executed by an application. It is possible that making this call in conjunction with control panel changes may relocate devices contrary to the ProDOS device list.

Possible errors

$06	BUSERR	Communications error
$28	NODRIVE	No Device Connected

SMARTPORT OPEN CALL

	Standard call	**Extended call**
CMDNUM	$05	$45
CMDLIST	parameter count	parameter count
	unit number	unit number

This call is used to prepare a character device for reading or writing.

Note that block devices do not accept this call, and will return a invalid command error ($01).

Required parameters

<u>parameter count:</u> byte value = $01

<u>unit number:</u> byte value in the range: $01 to $7E

Possible errors

$01	BADCMD	Invalid command
$06	BUSERR	Communications error
$28	NODRIVE	No Device Connected

SMARTPORT CLOSE CALL

	Standard call	**Extended call**
CMDNUM	$07	$47
CMDLIST	parameter count unit number	parameter count unit number

This call is used to tell an extended character device that a sequence of reads or writes is over. In the case of a printer, this call could have the effect of flushing the print buffer.

Note that block devices do not accept this call, and will return a invalid command error ($01).

Required parameters

<u>parameter count:</u>	byte value = $01
<u>unit number:</u>	byte value in the range: $01 to $7E

Possible errors

$01	BADCMD	Invalid command
$06	BUSERR	Communications error
$28	NODRIVE	No Device Connected

SMARTPORT READ CALL

	Standard call	Extended call
CMDNUM	$08	$48
CMDLIST	parameter count	parameter count
	unit number	unit number
	data buffer pointer (low byte)	data buffer pointer (low byte, low word)
	data buffer pointer (high byte)	data buffer pointer (high byte, low word)
	byte count (low byte)	data buffer pointer (low byte, high word)
	byte count (high byte)	data buffer pointer (high byte, high word)
	address pointer (low byte)	byte count (low byte)
	address pointer (mid byte)	byte count (high byte)
	address pointer (high byte)	address pointer (low byte, low word)
		address pointer (high byte, low word)
		address pointer (low byte, high word)
		address pointer (high byte, high word)

This call reads the number of bytes specified by the byte count into memory. The starting address of memory that the data is read into is specified by data buffer pointer. The address pointer references an address within the device that the bytes are to be read from. The meaning of the address parameter depends on the device involved. Although this call is generally intended for use by character devices, a block device might use this call to read a block of a non standard size (greater than 512 bytes per block). In this case, the address pointer may be interpreted as a block address.

Required parameters

<u>parameter count:</u> byte value = $04

<u>unit number</u>: 1 byte value in the range: $01 to $7E

<u>data buffer:</u>

Standard Call	Extended Call
word pointer (bank $00)	longword pointer

For standard calls, this is the two byte pointer a buffer that the data is to be read into. For extended calls, the pointer is a longword specifying a buffer in any memory bank. The buffer must be large enough to accomodate the number of bytes requested.

<u>byte count</u>: 2 byte number

This specifies the number of bytes which are to be transferred. All of the current implementations of the SmartPort utilizing SmartPort Bus have a limitation of 767 bytes. Other peripheral cards supporting the SmartPort interface may not have this limitation.

<u>address:</u>

Standard Call	Extended Call
3 byte address	4 byte address

The address is a device specific parameter usually specifying a source address within the device. An example of how this call might be implemented with an extended block device, is to use the address as a block address for accessing a non standard block. This is done with the AppleDisk 3.5 and UniDisk3.5 to read 524 byte Macintosh blocks from 3.5 inch media.

Possible errors

$06	BUSERR	Communications error
$27	IOERROR	I/O Error
$28	NODRIVE	No Device Connected
$2B	NOWRITE	DISK WRITE PROTECTED
$2F	BADBLOCK	Invalid block number
$2F	OFFLINE	Device off line or no disk in drive

SMARTPORT WRITE CALL

	Standard call	Extended call
CMDNUM	$09	$49
CMDLIST	parameter count	parameter count
	unit number	unit number
	data buffer pointer (low byte)	data buffer pointer (low byte, low word)
	data buffer pointer (high byte)	data buffer pointer (high byte, low word)
	byte count (low byte)	data buffer pointer (low byte, high word)
	byte count (high byte)	data buffer pointer (high byte, high word)
	address pointer (low byte)	byte count (low byte)
	address pointer (mid byte)	byte count (high byte)
	address pointer (high byte)	address pointer (low byte, low word)
		address pointer (high byte, low word)
		address pointer (low byte, high word)
		address pointer (high byte, high word)

This call writes the number of bytes specified by the byte count to the device specified by the unit number. The starting address of memory that the data is read from is specified by data buffer pointer. The address pointer references an address within the device where the bytes are to be written. The meaning of the address parameter depends on the device involved. The meaning of the address parameter depends on the device involved. Although this call is generally intended for use by character devices, a block device might use this call to write a block of a non standard size (greater than 512 bytes per block). In this case, the address field would be interpreted as a block address.

Required parameters

parameter count:	byte value = $04	
unit number:	1 byte value in the range: $01 to $7E	
data buffer:	Standard Call	Extended Call
	word pointer (bank $00)	longword pointer

For standard calls, this is the two byte pointer a buffer that the data is to be read into. For extended calls, the pointer is a longword specifying a buffer in any memory bank. The buffer must be large enough to accomodate the number of bytes requested.

byte count:	2 byte number

This specifies the number of bytes which are to be transferred. All of the current implementations of the SmartPort utilizing SmartPort Bus have a limitation of 767 bytes. Other peripheral cards supporting the SmartPort interface may not have this limitation.

address:	Standard Call	Extended Call
	3 byte value	4 byte value

The address is a device specific parameter usually specifying a destination address within the device. An example of how this call might be implemented with a block device, is to use the address as a block address for accessing a non standard block. This is done with the AppleDisk 3.5 and UniDisk3.5 to write 524 byte Macintosh blocks to 3.5 inch media.

Possible errors

$06	BUSERR	Communications error
$27	IOERROR	I/O Error
$28	NODRIVE	No Device Connected
$2B	NOWRITE	DISK WRITE PROTECTED
$2F	BADBLOCK	Invalid block number
$2F	OFFLINE	Device off line or no disk in drive

The following tables summarize the command numbers and parameter lists for standard and extended SmartPort calls.

Summary of Standard Commands and Parameter Lists						
Command	Status	ReadBlock	WriteBlock	Format	Control	Init
CMDNUM	$40	$41	$42	$43	$44	$45
CMDLIST Byte 0:	$03	$03	$03	$01	$03	$01
1:	Unit #	Unit #	Unit #	Unit #	Unit #	Unit #
2:	StatList Ptr	Buffer Ptr	Buffer Ptr		CtrlList Ptr	
3:	StatList Ptr	Buffer Ptr	Buffer Ptr		CtrlList Ptr	
4:	StatusCode	Block Addr	Block Addr		Ctrl Code	
5:		Block Addr	Block Addr			
6:		Block Addr	Block Addr			
7:						
8:						

Figure 7-9. Summary Of standard commands and parameter lists

Summary of Standard Commands and Parameter Lists						
Command	Open	Close	Read	Write		
CMDNUM	$46	$47	$48	$49		
CMDLIST Byte 0:	$01	$01	$04	$04		
1:	Unit #	Unit #	Unit #	Unit #		
2:			Buffer Pt	Buffer Pt		
3:			Buffer Pt	Buffer Pt		
4:			Byte Coun	Byte Coun		
5:			Byte Coun	Byte Coun		
6:			*	*		
7:			*	*		
8:			*	*		

This parameter is device specific

Figure 7-10. Summary of standard commands and parameter lists

1) The read byte count and the Control call list contents in some SmartPort implementations may not be larger than 767 bytes.

2) Upon return from the Read call, the byte count bytes will contain the number of bytes actually read from the device.

Summary of Extended Commands and Parameter Lists						
Command	Status	ReadBlock	WriteBlock	Format	Control	Init
CMDNUM	$40	$41	$42	$43	$44	$45
CMDLIST Byte 0:	$03	$03	$03	$01	$03	$01
1:	Unit #	Unit #	Unit #	Unit #	Unit #	Unit #
2:	StatList Ptr	Buffer Ptr	Buffer Ptr		CtrlList Ptr	
3:	StatList Ptr	Buffer Ptr	Buffer Ptr		CtrlList Ptr	
4:	StatList Ptr	Buffer Ptr	Buffer Ptr		CtrlList Ptr	
5:	StatList Ptr	Buffer Ptr	Buffer Ptr		CtrlList Ptr	
6:	StatusCode	Block Addr	Block Addr		Ctrl Code	
7:		Block Addr	Block Addr			
8:		Block Addr	Block Addr			
9:		Block Addr	Block Addr			
10:						
11:						

Figure 7-11. Summary of extended commands and parameter lists (part 1)

Summary of Extended Commands andaPameter Lists						
Command	Open	Close	Read	Write		
CMDNUM	$46	$47	$48	$49		
CMDLIST Byte 0:	$01	$01	$04	$04		
1:	Unit #	Unit #	Unit #	Unit #		
2:			Buffer Ptr	Buffer Ptr		
3:			Buffer Ptr	Buffer Ptr		
4:			Buffer Ptr	Buffer Ptr		
5:			Buffer Ptr	Buffer Ptr		
6:			Byte Count	Byte Count		
7:			Byte Count	Byte Count		
8:			*	*		
9:			*	*		
10:			*	*		
11:			*	*		

* This parameter is device spfici

Figure 7-12. Summary of extended commands and parameter lists (part 2)

Notes:

1) The read byte count and the Control call list contents in some SmartPort implementations may not be larger than 767 bytes.

2) Upon return from the Read call, the byte count bytes will contain the number of bytes read from the device.

Device Specific SmartPort Calls

In addition to the common command set of SmartPort calls already listed, a device may implement it's own device specific calls. Usually these calls will be implemented as a subset of the SmartPort Status or Control calls rather than a new command.

SmartPort calls unique to the AppleDisk 3.5

Seven AppleDisk 3.5 device specific calls have been added as an extension to the Control call. These device specific control calls may only be used with the AppleDisk 3.5. To determine if a device is an AppleDisk 3.5 the type and subtype bytes returned from a DIB statatus call may be examined. If the type byte is returned with a value of $01 and the subtype byte is returned with a value of $C0, then the device is an AppleDisk 3.5. Since the device specific calls to the AppleDisk 3.5 are implemented as control calls, only the control code and control list for these calls will be defined here. Refer to the section on the SmartPort Control Calls for information on the command byte and parameter list.

Eject	Control Code:		$04
	Control List:	Count Low Byte	$00
		Count High Byte	$00

This call is used to eject the media from the 3.5 inch drive.

SetHook	Control Code:		$05
	Control List:	Count Low Byte	$04
		Count High Byte	$00
		Hook Reference number	$xx
		Address Low	$xx
		Address High	$xx
		Address Bank	$xx

This call is used to redirect routines internal to the AppleDisk 3.5 driver. The routine to be redirected is referenced by the Hook Reference Number. The address that the routine is to be redirected to is specified by the 3 byte address field in the control list.

Valid Hook Reference Numbers and their associated routines are shown in the table below:

Hook Reference	Routine
$01	Read Address Field
$02	Read Data Field
$03	Write Data Field
$04	Seek
$05	Format Disk
$06	Write Track
$07	Verify Track

The routine READ ADDRESS FIELD reads bytes from the disk until it finds the address marks and a sector number specified as input parameters to the routine. The READ BLOCK routine will read a 524 byte Macintosh block or 512 byte Apple II block from the disk.

The WRITE DATA FIELD routine will write a 524 byte block of data to the disk. For Apple II blocks, the first 12 bytes will be written as zero.

The SEEK routine will position the read/write head over the appropriate cylinder on the disk.

The FORMAT routine writes the address marks, data marks, zeroed data blocks, checksum and end of block marks.

ADDRESS MARKS			ADDRESS FIELD							GAP	DATA MARKS				DATA FIELD		EOB MARKS		
D5	AA	96	TRACK	SEC	SIDES	FMT	CHKSUM	SLIP	SLIP	5 - 10 BYTES SYNC	D5	AA	AD	SECTOR #	342 DATA BYTES	CHKSUM	DE	AA	FF

Figure 7-13. Disk sector format

The WRITE TRACK routine is called by the formatter to write out one track of empty blocks. The number of blocks written is dependent on the track that the read/write head is positioned over.

The VERIFY routine is called by the formatter to verify that the data written by the WRITE TRACK routine was written correctly.

<u>ResetHook</u>	Control Code:		$06
	Control List:	Count Low Byte	$01
		Count High Byte	$00
		Hook Reference number	

This call is used to restore the default address for the hook specified in the control list.

<u>SetMark</u>	Control Code:		$07
	Control List:	Count Low Byte	$xx
		Count High Byte	$00
		Start Byte	$xx
		Data.......	

This call is used to change individual bytes in the mark tables. The count field specifies how many bytes in the mark table are to be written plus 1. The start byte references an offset into the mark table that the new bytes are to be written to. Bounds checking will be performed to make sure the byte count will not overflow the internal mark table. The default values for the MARK table is shown in detail below:

Byte $FF	: Byte 0 Sector Number
Byte $AD	: Byte 1 Data Marks
Byte $AA	: Byte 2
Byte $D5	: Byte 3
Byte $FF	: Byte 4
Byte $FC	: Byte 5 Sync Bytes
Byte $F3	: Byte 6
Byte $CF	: Byte 7
Byte $3F	: Byte 8
Byte $FF	: Byte 9
Byte $FF	: Byte 10 Bit Slip Marks
Byte $AA	: Byte 11
Byte $DE	: Byte 12
Byte $FF	: Byte 13
Byte $FF	: Byte 14 Inter-Header Gap

Byte $FF	: Byte 15
Byte $FF	: Byte 16
Byte $FF	: Byte 17
Byte $96	: Byte 18 Address Marks
Byte $AA	: Byte 19
Byte $D5	: Byte 20
Byte $FF	: Byte 21

ResetMark	Control Code:		$08
	Control List:	Count Low Byte	$xx
		Count High Byte	$00
		Start Byte	$xx

This call is used to restore individual bytes in the mark tables to the default values. The count field defines how many bytes in the mark table are to be restored plus 1. The start field defines where in the mark table the bytes are to be restored.

SetSides	Control Code:		$09
	Control Code:	Count Low Byte	$01
		Count High Byte	$00
		Number of Sides	$nn

This call is used to set the number of sides of the media to be formatted by the format call. This allow for support of either single sided or double sided media. When the most significant bit of the number of sides field is set to '1', then double sided media will be formatted. If the most significant bit is cleared to '0', then single sided media will be formatted.

SetInterleave	Control Code:		$0A
	Contrl List:	Count Low Byte	$01
		Count High Byte	$00
		Interleave	$01 to $0C

This call is used to set the sector interleave that will be layed down on the disk by the format call.

SmartPort calls unique to the UniDisk 3.5

Five UniDisk 3.5 device specific calls have been added as extensions to the Control and Status calls. These device specific calls may only be used with the UniDisk 3.5. To determine if a device is an UniDisk 3.5 the type and subtype bytes returned from a DIB statatus call may be examined. If the type byte is returned with a value of $01 and the subtype byte is returned with a value of $00, then the device is a UniDisk 3.5. For calls implemented as extensions to the control call, only the control code and control list will be defined. For calls implemented as extensions to the status call, only the status code and status list will be defined. Refer to the sections on the SmartPort Control and Status Calls for more information on these calls.

Eject	Control Code:		$04
	Control List:	Count Low Byte	$00
		Count High Byte	$00

This call is used to eject the media from the 3.5 inch drive.

Execute

Control Code:		$05
Control List:	Count Low Byte	$06
	Count High Byte	$00
	Accumulator Value	$xx
	X Register Value	$xx
	Y Register Value	$xx
	Processor Status Value	$xx
	Low Program Counter	$xx
	High Program Counter	$xx

This call is used to dispatch the intelligent controller in the UniDisk 3.5 device to execute a 65C02 subroutine. The register setup is passed to the routine to be executed from the control list.

SetAddress

Control Code:		$06
Control List:	Count Low Byte	$02
	Count High Byte	$00
	Low Byte Address	$xx
	High Byte Address	$xx

This call is used to set the address in the UniDisk 3.5 controller's memory space that the DownLoad call will load a 65C02 routine into. Care must be taken that the download address is set only to free space in the UniDisk 3.5 memory map.

DownLoad

Control Code:		$07
Control List:	Count Low Byte	$xx
	Count High Byte	$xx
	Executable 65C02 Routine.....	

This call is used to download an executable 65C02 routine into the memory resident on the UniDisk 3.5 controller. The address that the routine is loaded into is set by the SetAddress call. The count field must be set to the length of the 65C02 routine to be downloaded.

UniDiskStat

Status Code:		$05
Status List:	Byte	$00
	Soft Error	$xx
	Retries	$xx
	Byte	$00
	A Register after Execute	$xx
	X Register after Execute	$xx
	Y Register after Execute	$xx
	P Register after Execute	$xx

This call allows an application to get more detail about an error that may be returned on a read or a write. It also allows an application to have access to the 65C02 register state after dispatching the UniDisk 3.5 controller to execute a 65C02 routine via the EXECUTE call.

Addresses of memory mapped I/O internal to the UniDisk 3.5 controller are shown below:

UniDisk 3.5 Intelligent Controller

RAM Usage Memory Map

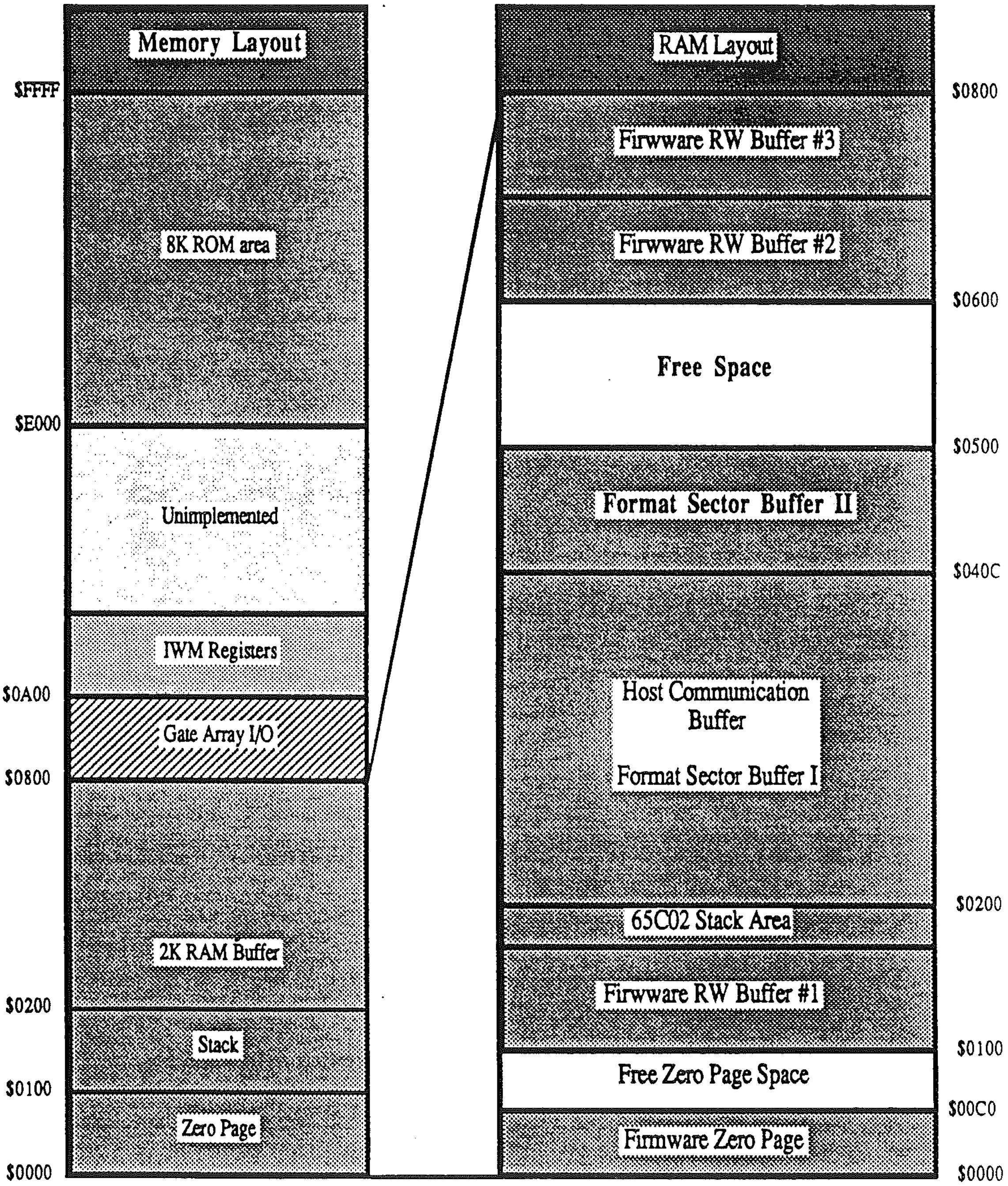


Figure 7-14. Unidisk 3.5 memory map

UniDisk 3.5 Gate Array I/O Locations					
Function	**data4**	**data3**	**data2**	**data1**	**data0**
Read $800	LASTONE	/BUSEN	/WRREQ	/GATENBL	HDSEL
Wrt $800	TRIGGER	ENBUS	PH3EN	IWMDIR	HDSEL
Read $801	SENSE	BLATCH1	BLATCH2	LIRONEN	CA0
Wrt $801	/RSTIWM	/BLATCH CLR1	/BLATCH CLR2	DRIVE1	DRIVE2

Figure 7-15. Unidisk 3.5 gate array I/O locations

UniDisk 3.5 IWM Locations			
Location	Specific Label	IWMDIR=0 (drv)	IWMDIR=1 (host)
$0A00	PHASE0 reset	CA0 reset	/BSY handshake
$0A01	PHASE0 set	CA0 set	/BSY handshake
$0A02	PHASE1 reset	CA1 reset	--
$0A03	PHASE1 set	CA1 set	--
$0A04	PHASE2 reset	CA2 reset	--
$0A05	PHASE2 set	CA2 set	--
$0A06	PHASE3 reset	LSTRB reset	--
$0A07	PHASE3 set	LSTRB set	--
$0A08	MOTOROFF	--	--
$0A09	MOTORON	--	--
$0A0A	ENABLE1	--	--
$0A0B	ENABLE2	--	--
$0A0C	L6 reset	--	--
$0A0D	L6 set	--	--
$0A0E	L7 reset	--	--
$0A0F	L7 set	--	--

Figure 7-16. Unidisk 3.5 IWM locations

ROM Disk Driver

The *ROM Disk* is a plug-in card that houses ROM's that may be organized into blocks that emulate a disk device, or provide space for rom based programs. This may include ROM based extensions to the tool set, desk accessories or applications. Although SmartPort has no built-in ROM Disk, SmartPort will support an external ROM Disk driver.

Installing a RomDisk driver

A RomDisk driver must reside at address $F0/0000. The ROMDISK may only occupy the address space from $F0/0000 through $F7/FFFF. The base address of the driver must contain the ASCII string 'ROMDISK' in upper case with the MSB on. Entry to the RomDisk driver will be through address $F0/0007. The SmartPort firmware will search for a RomDisk driver during the boot process while assigning unit numbers to each of the SmartPort devices. If the ASCII string 'ROMDISK' is found at address $F0/0007, an initialization call will be executed to the ROM Disk driver via the RomDisk entry point. If the RomDisk returns with no error, the RomDisk driver will be installed into the SmartPort device chain. If the RomDisk initialization call returns an error, the RomDisk driver will not be installed in the SmartPort device chain.

Passing parameters to the ROM disk

Call parameters are passed to the ROM Disk from SmartPort through fixed memory locations in absolute zero page. All input to device specific drivers are passed in an extended format. This is done even for standard SmartPort calls so that the call parameters will always be found in fixed locations. This does not mean that a non extended call will be changed to an extended call. Only the organiztion of parameters is affected.

Some parameters do not occupy contiguous memory when presented in an extended format. This occurs because the order of parameters has been prepared so that the parameters can be transmitted over SmartPort Bus to intelligent devices. Absolute zero page locations $40-62 have been saved by SmartPort prior to dispatching to the ROM Disk driver, and will be restored by SmartPort after returning from the driver. This means that these locations are available for use by the ROM Disk driver. Call parameters are passed to the ROM Disk driver as shown below:

Location	Parameters	Call Type
$42	Buffer Address (bits 0-7)	All
$43	Buffer Address (bits 8-15)	All
$44	Buffer Address (bits 16-23)	All
$45	Command	All
$46	Parameter Count	All
$47	Buffer Address (bits 24-31)	All
$48	Extended Block (bits 0-7) Status Code or Control Code Byte Count (bits 0-7)	ReadBlock & Writeblock Status & Control Read & Write
$49	Extended Block (bits 8-15) Byte Count (bits 8-15)	ReadBlock & Writeblock Read & Write
$4A	Extended Block (bits 16-23) Address Pointer (bits 0-7)	ReadBlock & Writeblock Read & Write
$4B	Extended Block (bits 24-31) Address Pointer (bits 8-15)	ReadBlock & Writeblock Read & Write
$4C	Address Pointer (bits 16-23)	Read & Write
$4D	Address Pointer (bits 24-31)	Read & Write

Parameters being returned to the application from the ROM Disk Driver are passed in absolute zero page locations as follows:

Location	Output Parameter Passed
$000050	Error Code
$000051	Low byte of count of bytes tranferred to host
$000052	High byte of count of bytes tranferred to host

All I/O information being passed between the application making the SmartPort call and the ROM Disk driver will be passed through the buffer specified in the parameter list.

ROM organization

The ROM must contain the ROM Disk signature string as well as a ROM Disk driver. If portions of the ROM are to be organized as blocks then a map of the ROM address space might look like the figure shown below.

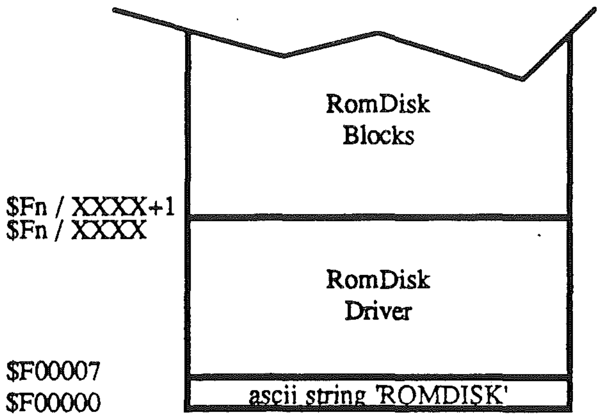


Figure 7-17. The ROMDISK

It is possible to use the expansion ROM for both a RomDisk and ROM based extensions to the tool set by partitioning the ROM into three areas (Driver, Blocks and Tools) as shown below:

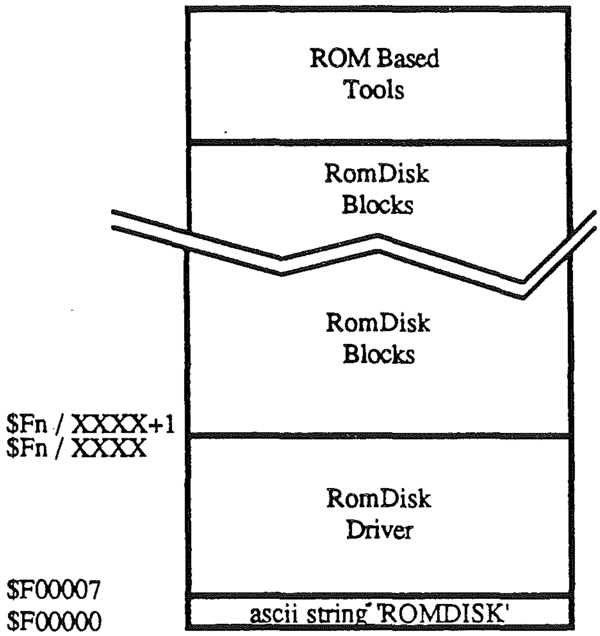


Figure 7-18. Partitioning the ROM

The initialization call made to the RomDisk driver should make a call to the ToolLocator to install any ROM based tool set extensions. This then allows the tool locator to dispatch to the ROM based tool set extensions directly rather than down loading the tool set to RAM as would be neccessary if the ROM disk only emulated disk I/O.

A block diagram of a RomDisk that occupies 128k of ROM (including the driver itself) is shown below:

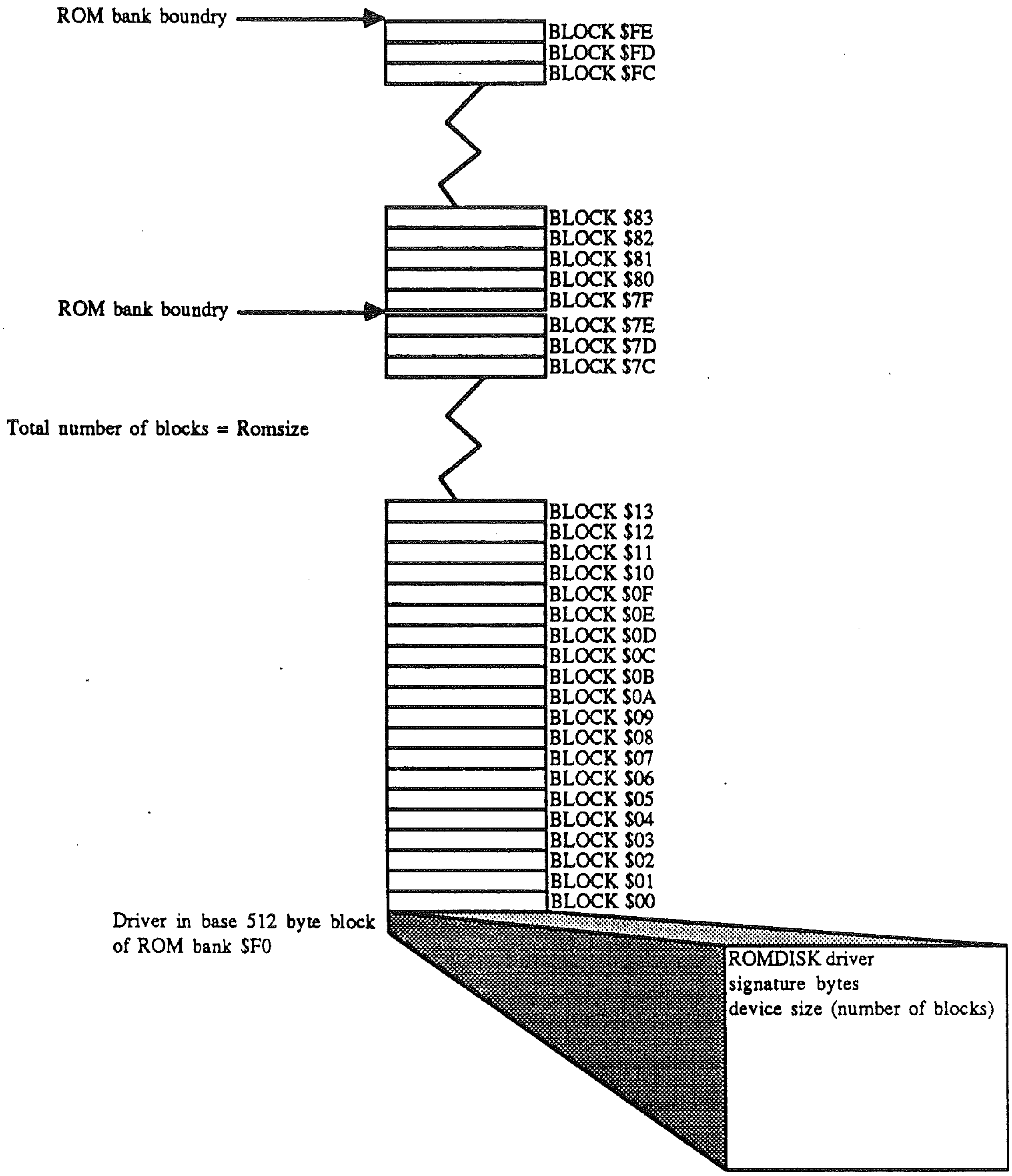


Figure 7-19. Block diagram of a 128k ROM disk

Note that no ROM space has been reserved for toolset expansion in this example.

SUMMARY OF SMARTPORT ERROR CODES

Acc value	Error type	Description
$00	No error	
$01	BADCMD	A nonexistent command was issued.
$04	BADPCNT	Bad call parameter count. This error will occur only if the call parameter list was no properly constructed.
$06	BUSERR	A communications error with the IWM occured.
$11	BADUNIT	An invalid unit number was givien.
$1F	NOINT	Interrupt devices not supported.
$21	BADCTL	The control or status code is not supported by the device.
$22	BADCTLPARM	The control list contains invalid information.
$27	IOERROR	The device encountered an I/O error.
$28	NODRIVE	The device is not connected. This can occur if the device is not connected but its controller is.
$2B	NOWRITE	The device is write protected.
$2D	BADBLOCK	The block number is not present on the device.
$2F	OFFLINE	Device off line or no disk in drive.
$30-$3F	DEVSPEC	These are device specific error codes
$40-$4F	RESERVED	
$50-$5F	NONFATAL	A device specific 'soft' error. The operation completed successfully, but some exception condition was detected.

THE SMARTPORT BUS

The SmartPort Bus is a daisy chain configuration of intelligent devices, sometimes called "bus residents", which are connected to the disk port of the host CPU. A Disk][type device may be physically connected to the end of the SmartPort device chain on the Apple IIGS and its operation occurs transparently to host software. The Disk][device remains dormant when a SmartPort bus resident is addressed. The number of bus residents is limited by supply power and IWM drive considerations, since the software supports up to 127 residents. Power requirements usually limit the maximum number of bus residents to four.

The drive selection is done through software. The command string contains a byte specifying the device to be accessed. These device ID bytes are assigned at bus reset by the SmartPort in a manner to be described below.

There are two functions which are strictly hardware invoked, bus reset, and bus enable. Both of these conditions are envoked by asserting combinations of phase lines on the disk port which never occur under normal Disk][operation (Both functions involve asserting opposing phases - this is pointless to do on a Disk][.) This allows a Disk][type device and other bus residents to effectively stay out of each other's way.

Function	PH3	PH2	PH1	PH0
Enable	1	X	1	X
Reset	0	1	0	1

The PH0 don't care ('X') is necessary since when the bus is enabled, PH0 is used as a REQ handshake line cycled on a packet basis. ACK is sensed from the device through the IWM write protect sense status.

How SmartPort assigns unit numbers

The assignment of unit numbers is initiated by the executing a call to the slot 5 boot entry point, and always begins with a bus reset. The reset flips a latch on all bus residents which causes the daisy-chained phase 3 line to become low. This causes all daisy-chained devices to be incapable of receiving the bus enable signal, which involves phase 3 high.

The host then sends the ID definition command. Whenever a device receives this command (with ENABLE), it takes the unit number embedded in the command string and assigns that number as its own unit number. Thereafter it will not respond to any command string with a unit number other than that given it in the ID definition command.

Upon completing the ID definition command, the bus resident re-enables the phase 3 line, allowing the next resident to recieve its ID definition command. This process continues as long as their are bus residents. The last bus resident in the device chain returns an exception indicating that is the last bus resident.

Although disk][devices are connected to the disk port, they are not bus residents and will not respond to the ID definition command. A resident determines that he is the last intelligent device in the chain through the sensing of a signal, normally unused in Disk][operation, which is grounded by all intelligent devices. If no bus resident or a Disk][type device is daisy-chained to the port, this line is read as high.

SmartPort interaction with the Disk][

The disk port built into the Apple IIGS will support a daisy chained 5 1/4" disk (UniDisk5.25, Disk// or DuoDisk). This is done by sharing the same disk port hardware between two different slot rom areas. Slot 5 ROM area contains the SmartPort interface and ProDOS block device driver while slot 6 ROM area contains the Disk][interface. The Disk][device is enabled by the disk port signal /ENABLE2. The SmartPort must activate this line to communicate with intelligent bus residents. If this line were not intercepted before being passed on to daisy chained devices, any attempt to talk to devices on the bus would result in spurious operation of the Disk][at the end of the chain.

For the Disk][to remain aloof to SmartPort activity, each resident must gate the /ENABLE2 line so that whenever any SmartPort bus resident is enabled (PHASE1 and PHASE3), any Disk][at the end of the chain will be disabled.

In other words, the /ENABLE2 line is only passed onto daisy chained devices when either PHASE1 or PHASE3 are low:

BUS ENABLE (PH1 & PH3)	/ENABLE2 (daisy)
PHASE1=0 or PHASE3=0	/ENABLE2
PHASE1=1 and PHASE3=1	deasserted (high)

Other considerations

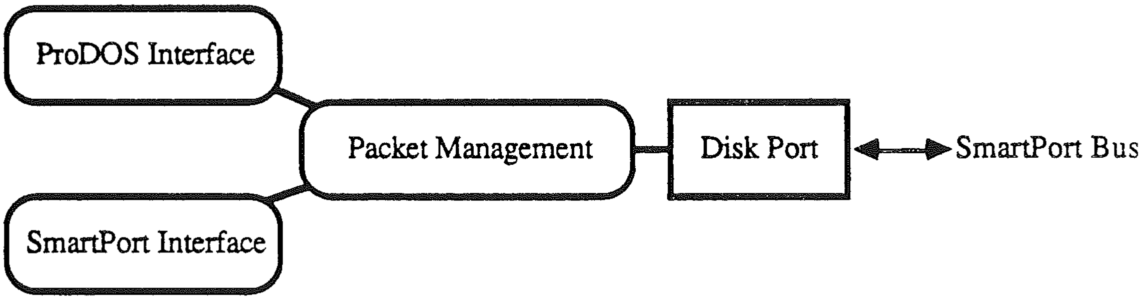
All intelligent residents try to process every command packet that goes over the bus, responding only if it recognizes its own ID, Type and SubType encoded in the packet. It is the Device Type and command that will be used by the device to arbitrate between extended and standard packets. Thus one resident can tell when some other resident is being accessed or if the packet type (extended or standard) is compatable with the device. It is therefore possible for a device controller to bring itself to some low power consumption mode when it is not being accessed constantly.

Extended and standard command packets

There is no difference in the number of bytes passed over the SmartPort bus in a standard command packet versus an extended command packet. Standard SmartPort commands may have a parameter list consisting of nine bytes maximum. Extended SmartPort commands may have a parameter list consisting of eleven bytes maximum. The command packet was designed for a maximum of nine bytes of information. The first two bytes always contains the SmartPort command number and parameter count. The remaining seven bytes consists of the seven bytes of the parameter list starting with the third byte for standard commands or the fifth byte for extended commands. Seven bytes from the parameter list are always copied into the command packet even though the parameter list for the current command may consists of less than seven bytes.

SmartPort Bus description

The general flow of control of the SmartPort might be represented in this manner:



SmartPort Anatomy

Figure 7-20. SmartPort anatomy

Whenever a call is made to the SmartPort device driver that utilizes SmartPort Bus, the command table sent to the device driver is converted into what is known as a 'command packet' before being sent to the device. Then the results of the call are sent back from the device in a 'packet'. All data sent across the bus is placed in these packets. Each byte of the packet is a seven bit quantity (bit 7 is always set), a limitation imposed by the IWM. All data sent is converted from eight bit quantities to seven bit before transmission.

The information of the packet can be broken down into the following categories:

- General Overhead
- Source and Destination IDs
- Contents Type and AuxType
- Contents Status
- Contents

The IDs are seven bit quantities and are assigned sequentially according to the device's position in the chain. The host is always ID=0. Since every byte in the packet has the MSB set, the host is $80, the first device in the chain is $81, etc.

The contents type consists of a type and auxtype byte. There are three currently defined contents types. Type = $80 is a command packet, $81 is a status packet, and $82 is a data packet. Bit 6 in the command byte and the AuxType byte defines the packet as either extended or non extended. AuxType = $80 is a non extended packet, $C0 is an extended packet. Command = $8X is a non extended packet, $CX is an extended packet.

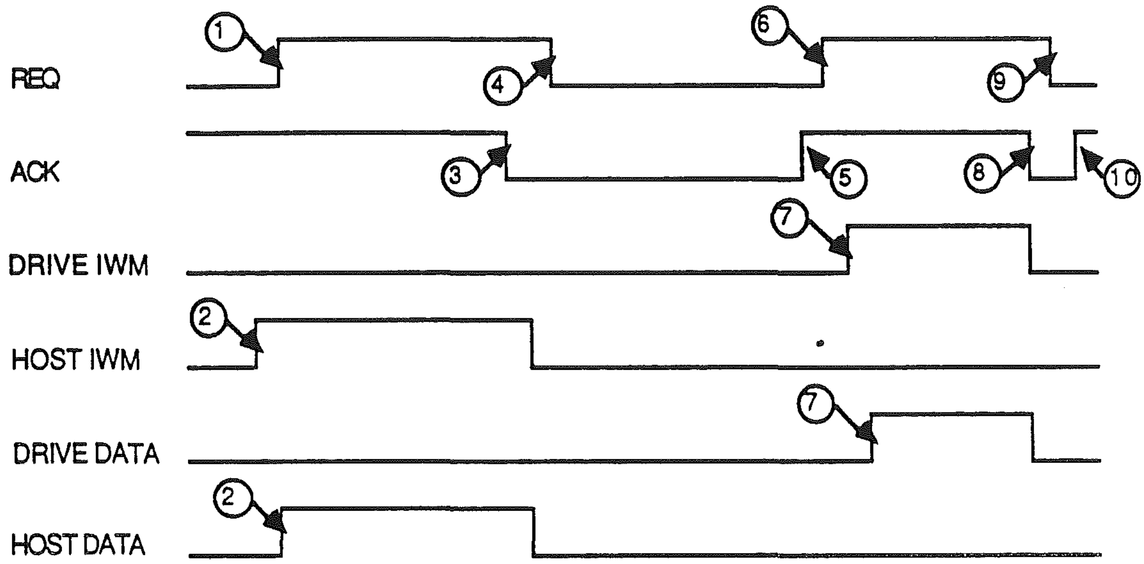
The contents byte is used on status and data packets. It contains the error code for read/write operations. This is the byte that the SmartPort will return as an error code for the call.

The contents itself consists of bytes of seven bits (hi bit set) of encoded data. Preceding the bytes themselves are two length bytes. If the number of content bytes is BYTECOUNT, then the first byte is defined as BYTECOUNT DIV 7, and the second byte is defined as BYTECOUNT MOD 7. In other words, the first byte specifies the number of groups of seven bytes of content, and the second is the remainder. Note that the second byte will never have a value larger than 6. Both these bytes have their MSB set.

The general overhead bytes are packet begin and end marks, sync bytes (6) to ensure correct synchronization of the IWMs, and a checksum. The checksum is computed by exclusive ORing all the content data bytes (8 bits) and the IDs, type bytes, status and length bytes. The checksum is eight bits sent as sixteen (see SmartPort Bus Packet Format diagram).

SmartPort Bus Communications Protocol

EXECUTING A READ FROM THE DEVICE

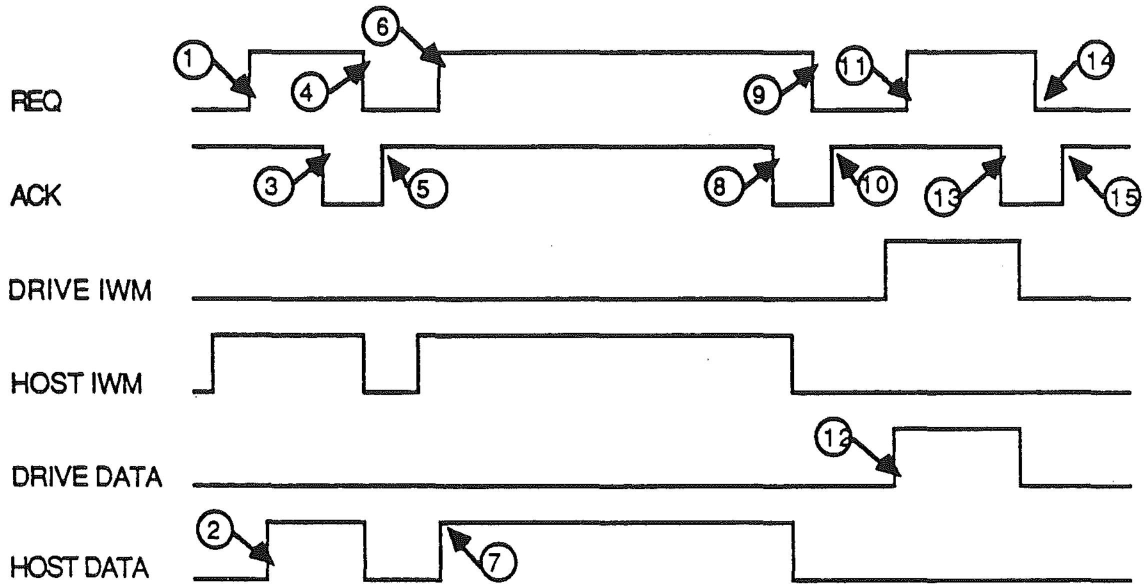


1. Host asserts REQ when ACK is negated and command packet is
2. Host enables IWM and sends packet to
3. Device deasserts ACK signalling the HOST that the packet was
4. Host responds by deasserting REQ.
5. Device asserts ACK when ready to send response packet to
6. Host asserts REQ when ready to receive response packet from
7. Device enables IWM and sends response packet to
8. Device deasserts ACK at end of
9. Host deasserts REQ when packet
10. Device asserts ACK to indicate ready to receive command

Figure 7-21. SmartPort Bus communications -Read protocol

SmartPort Bus Communications Protocol

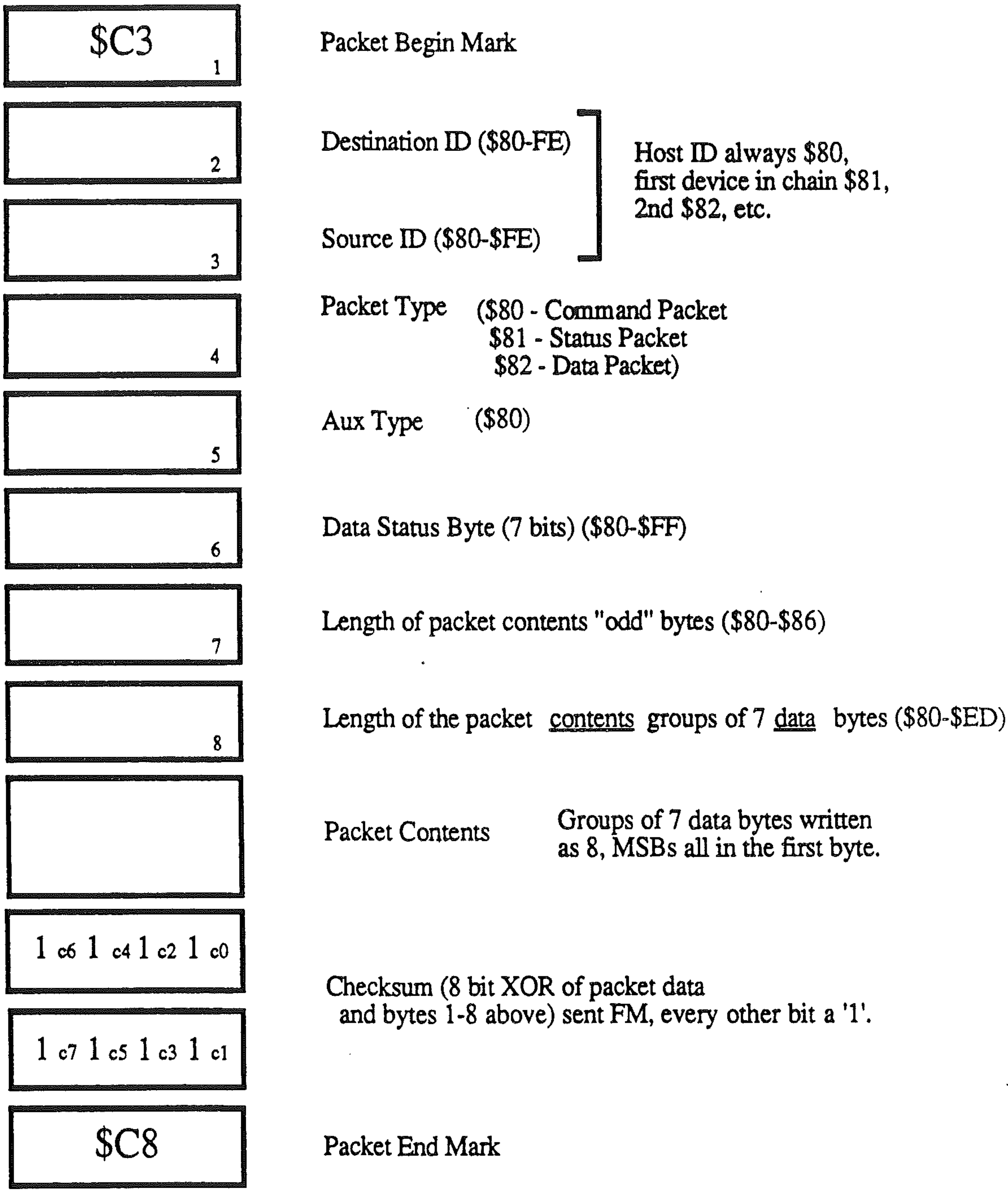
EXECUTING A WRITE TO THE DEVICE



1. Host asserts REQ when ACK is negated and command packet is
2. Commad packet is sent.
3. Device asserts ACK signalling that it got the
4. Host negates ACK finishing the command
5. When REQ is negated and device is ready to receive write data, device
6. When ACK is negated and host is ready to send, host asserts
7. Host sends write data.
8. Device asserts ACK signalling it received the
9. Host negates REQ allowing device to write data to
10. Device negates ACK and writes data to
11. Host responds to negated ACK by asserting REQ signalling ready for
12. Device responds to REQ by sending status to the
13. Device asserts ACK signalling status
14. Host acknowledges reciept of status by negating
15. Device negates ACK when ready for next command

Figure 7-22. SmartPort Bus communications -Write protocol

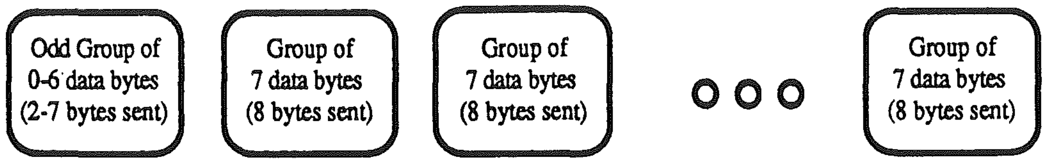
SmartPort Bus Packet Format



All bytes have the MSB set per IWMeqts.

Figure 7-23. SmartPort Bus packet format

SmartPort Bus Packet Contents



(Packet Sizes range from 0 to 767 data bytes)

- Take a group of 7 data bytes to be encoded,

  d1 $_{\text{bits }7..0}$ d2 $_{\text{bits }7..0}$ d3 $_{\text{bits }7..0}$ d4 $_{\text{bits }7..0}$ d5 $_{\text{bits }7..0}$ d6 $_{\text{bits }7..0}$ d7 $_{\text{bits }7..0}$

  where bit 7 is the most significant bit, then the bytes which are serially sent are as follows:

Top Bits Byte	1	$d1_7$	$d2_7$	$d3_7$	$d4_7$	$d5_7$	$d6_7$	$d7_7$
Byte 1	1	$d1_6$	$d1_5$	$d1_4$	$d1_3$	$d1_2$	$d1_1$	$d1_0$
Byte 2	1	$d2_6$	$d2_5$	$d2_4$	$d2_3$	$d2_2$	$d2_1$	$d2_0$
Byte 3	1	$d3_6$	$d3_5$	$d3_4$	$d3_3$	$d3_2$	$d3_1$	$d3_0$
Byte 4	1	$d4_6$	$d4_5$	$d4_4$	$d4_3$	$d4_2$	$d4_1$	$d4_0$
Byte 5	1	$d5_6$	$d5_5$	$d5_4$	$d5_3$	$d5_2$	$d5_1$	$d5_0$
Byte 6	1	$d6_6$	$d6_5$	$d6_4$	$d6_3$	$d6_2$	$d6_1$	$d6_0$
Byte 7	1	$d7_6$	$d7_5$	$d7_4$	$d7_3$	$d7_2$	$d7_1$	$d7_0$

- To determine the number of bytes in the odd group, is the remainder of the # of data bytes in the packet divided by 7. When encoding the oddbytes, assume that the rest of the bytes making up a group of seven are zero. (See example) Also note that if there are no oddbytes (ie #packet data bytes/7 has zero remainder) the oddbytes group is just omitted. Similarly, if the number of bytes in the packet is less than seven, there will be no 'seven' groups, only an 'oddbyte' group.

- For example, if you are sending a 512 byte packet, the number of groups of seven equals 73, and the remainder is 1. Therefore the first data byte will be sent as an odd group, followed by 73 groups of seven bytes. The groups of seven bytes will be encoded as above, and the odd bytes (byte in this example {d1 bits 7..0}) will be sent like so:

Top Bits Byte	1	$d1_7$	0	0	0	0	0	0
Byte 1	1	$d1_6$	$d1_5$	$d1_4$	$d1_3$	$d1_2$	$d1_1$	$d1_0$

Note that the Top Bits for data bytes 2 through 7 in this example are omitted along with the bytes which would have encoded the low seven bits of the data bytes 2 through 7. The oddbytes group is simply a special case of an instance of a group of seven.

Figure 7-24. SmartPort Bus packet contents

Standard Command Packet Contents

	Status	ReadBlock	WriteBlock	Format	Control	Init
BYTE 1	$00	$01	$02	$03	$04	$05
BYTE 2	Parameter Count	Parameter Count	Parameter Count	Parameter Count	Parameter Count	Parameter Count
BYTE 3	Byte 3 of Parameter List	Byte 3 of Parameter List	Byte 3 of Parameter List	-	Byte 3 of Parameter List	-
BYTE 4	Byte 4 of Parameter List	Byte 4 of Parameter List	Byte 4 of Parameter List	-	Byte 4 of Parameter List	-
BYTE 5	-	Byte 5 of Parameter List	Byte 5 of Parameter List	-	-	-
BYTE 6	-	Byte 6 of Parameter List	Byte 6 of Parameter List	-	-	-
BYTE 7	-	Byte 7 of Parameter List	Byte 7 of Parameter List	-	-	-
BYTE 8	-	-	-	-	-	-
BYTE 9	-	-	-	-	-	-

Note: Bytes with '-' have indeterminate values...
the device should ignore these.

Figure 7-25. Standard command packet contents (part 1)

Standard Command Packet Contents

	Open	Close	Read	Write
BYTE 1	$06	$07	$08	$09
BYTE 2	Parameter Count	Parameter Count	Parameter Count	Parameter Count
BYTE 3	Byte 3 of Parameter List	Byte 3 of Parameter List	Byte 3 of Parameter List	Byte 3 of Parameter List
BYTE 4	Byte 4 of Parameter List	Byte 4 of Parameter List	Byte 4 of Parameter List	Byte 4 of Parameter List
BYTE 5	-	-	Byte 5 of Parameter List	Byte 5 of Parameter List
BYTE 6	-	-	Byte 6 of Parameter List	Byte 6 of Parameter List
BYTE 7	-	-	Byte 7 of Parameter List	Byte 7 of Parameter List
BYTE 8	-	-	Byte 8 of Parameter List	Byte 8 of Parameter List
BYTE 9	-	-	Byte 9 of Parameter List	Byte 9 of Parameter List

Note: Bytes with '-' have indeterminate values...
the device should ignore these.

Figure 7-26. Standard command packet contents (part 2)

Extended Command Packet Contents

	Status	ReadBlock	WriteBlock	Format	Control	Init
BYTE 1	$40	$41	$42	$43	$44	$45
BYTE 2	Parameter Count	Parameter Count	Parameter Count	Parameter Count	Parameter Count	Parameter Count
BYTE 3	Byte 5 of Parameter List	Byte 5 of Parameter List	Byte 5 of Parameter List	-	Byte 5 of Parameter List	-
BYTE 4	Byte 6 of Parameter List	Byte 6 of Parameter List	Byte 6 of Parameter List	-	Byte 6 of Parameter List	-
BYTE 5	-	Byte 7 of Parameter List	Byte 7 of Parameter List	-	-	-
BYTE 6	-	Byte 8 of Parameter List	Byte 8 of Parameter List	-	-	-
BYTE 7	-	Byte 9 of Parameter List	Byte 9 of Parameter List	-	-	-
BYTE 8	-	-	-	-	-	-
BYTE 9	-	-	-	-	-	-

Note: Bytes with '-' have indeterminate values...
the device should ignore these.

Figure 7-27. Extended command packet contents (part 1)

Extended Command Packet Contents

	Open	Close	Read	Write
BYTE 1	$46	$47	$48	$49
BYTE 2	Parameter Count	Parameter Count	Parameter Count	Parameter Count
BYTE 3	Byte 5 of Parameter List	Byte 5 of Parameter List	Byte 5 of Parameter List	Byte 5 of Parameter List
BYTE 4	Byte 6 of Parameter List	Byte 6 of Parameter List	Byte 6 of Parameter List	Byte 6 of Parameter List
BYTE 5	-	-	Byte 7 of Parameter List	Byte 7 of Parameter List
BYTE 6	-	-	Byte 8 of Parameter List	Byte 8 of Parameter List
BYTE 7	-	-	Byte 9 of Parameter List	Byte 9 of Parameter List
BYTE 8	-	-	Byte 10 of Parameter List	Byte 10 of Parameter List
BYTE 9	-	-	Byte 11 of Parameter List	Byte 11 of Parameter List

Note: Bytes with '-' have indeterminate values...
the device should ignore these.

Figure 7-28. Extended command packet contents (part 2)

Chapter 10

Mouse Firmware

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

Important: The information in this manual regarding soft switches and hardware registers for the Apple IIGS Mouse are provided for information only. All applications must use the firmware calls only if they wish to be compatible with the Mouse used on previous and future Apple II systems.

Introduction

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 II c mouse interface, each of which depend extensively on firmware to support the mouse. The Apple IIGS Mouse firmware has a true passive mode like the AppleMouse, but differs from the II c Mouse, which requires interrupts to do anything.

Certain devices, to operate properly, must be the sole source of interrupts within a system in that 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 II c if someone is moving the mouse rapidly while an application program is running.

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

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

Unlike the IIc Mouse, which interrupts whenever the mouse is moved, the Apple IIGS Mouse interrupts in sync with the vertical blanking signal. This automatically limits the total number of Mouse interrupts to 60 per second, cutting down on the overhead the mouse 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 happens.

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

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 inches of travel.

Marginal Gloss: 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 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 the last read

Note: At power up or at reset, the GLU chip enters a noninterrupt state while also turning the Mouse interrupt off.

The ADB microcontroller automatically processes Mouse data. The microcontroller periodically polls the Mouse to check for activity. If the mouse is moved, or the button is pushed, two 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 taken place, the two 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 only writes Mouse data 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.

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)

The next section describes the register addresses used by the firmware to control or to communicate with the mouse.

Register addresses (used by 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. The paragraphs that follow this table outline the conditions under which these registers are used by the firmware.

Table 10-2. Register addresses used for the Apple IIGS Mouse

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/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, then writing the modified value back into the register.

If mouse interrupts are enabled, the firmware determines if 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, to allow accurate mouse readings to be obtained. Failure to perform the steps in this order would necessitate some corrective action since the data would 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. The corrective actions that the firmware writer must take is outlined below.

- 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.

  The firmware writer must use caution when using indexed instructions. The false read andwrite 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 are contaminated and must be considered useless. If that 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) ticks.
- 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 only allowed to move 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*.

Marginal Gloss: Clamps are data values that specify a maximum or minimum value for some other variable. For this instance, the Mouse clamps specify the minimum and maximum positions of the cursor onscreen.

The Mouse clamps reside in RAM locations reserved for the firmware. You should only access these locations by 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). 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. "n" is the slot number of the Mouse.

The following assembly code example correctly sets up the entry point for the mouse firmware. To use the code, you must decide which mouse firmware command you wish to perform, and duplicate the code below for each of the routines you use. For example, to utilize SERVEMOUSE from assembly code, you would substitute the line labeled 'SETMENTRY LDA SETMOUSE' with a line that reads 'SERVEMENTRY LDA SERVEMOUSE' where SERVEMOUSE is equated to $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

The Mouse and BASIC have the following interface. To turn the Mouse on, 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 columns.

To accept outputs from BASIC, the firmware changes the output links at $36 and $37 to point to $C407 and performs an INITMOUSE (described above).

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 (to input data), 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 into 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 Y directions, and position data in the screen holes are set to 0.

During a BASIC INPUT statement, the firmware reads the position changes (deltas) from the ADB Mouse, 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, into the input buffer followed by a carriage return and returns to BASIC.

> *Note:* The name '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, but which are unused by the display circuitry. Thus these so-called screen-holes can be utilized 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 that reads '?EXTRA IGNORED' would be displayed.

If you want to read button 1 status, you can use BASIC's 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 = 5 ASCII characters giving the decimal value of X

y1...y5 = 5 ASCII characters giving the decimal value of Y

sb = Minus (-) if key on keyboard was pressed during input statement and plus (+) if no key was pressed during input statement

B0 = ASCII space character

b5 = 1 if button 0 is pressed now and was pressed in last INPUT statement

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

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

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

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. A POKE 49168,0 resets the strobe.

The Mouse is resident in 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, 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 lists the Mouse's screen-hole use when Apple IIGS firmware is used.

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)

Figures 10-1 and 10-2 show how the bits of the Button Interrupt Status Byte and the Mode Byte are assigned.

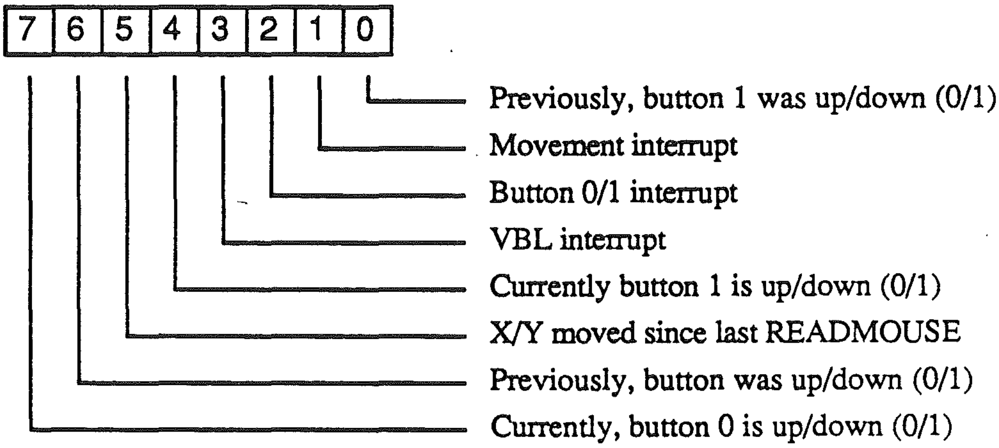


Figure 10-1. Button interrupt status byte ($77C)

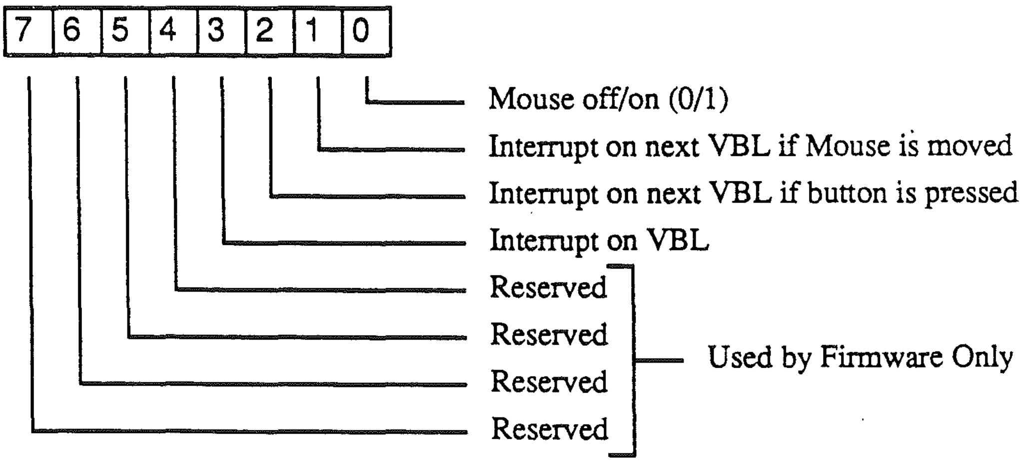


Figure 10-2. Mode byte ($7FC)

Summary of firmware calls

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

Table 10-4. Firmware calls

Location	Routine	Definition
$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

The following are routines implemented on Apple IIGS, Apple II, and the AppleMouse card:

Location	Routine	Definition
$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 defaults, positions to 0,0

The next six entry points are provided for compatibility with the AppleMouse card and do nothing in the Apple IIGS:

Location	Routine	Definition
$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

In addition to the routines listed above, the following locations are also signficant:

Location	Routine	Definition
$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

The above sections described how the mouse is accessed from BASIC. The next section talks about entry points from Pascal.

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 carry set to 1. The following is a list of the 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 it's called 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

The above sections described how the mouse is accessed from BASIC and PASCAL. The next section talks about entry points from assembly language.

Assembly language calls

This section lists the Assembly Language firmware calls. When you use the Mouse from assembly language, there are several items that you must keep in mind. These items are specified in the form of notes that precede the table of routines.

Note:

- 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 applications program. The only exception is during the function POSMOUSE when new Mouse coordinates are written, by the applications program, directly into the screen holes. No other Mouse screen hole can be changed by an applications 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.

Now, 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 (Example $C412) to which you add the value of the main entry point (Example $C400) to obtain the actual address to which the processor must jump to execute that routine.

SETMOUSE ($C412)

Function:	Sets Mouse operation mode
Input:	A = mode ($00 to $0F, only valid modes) X = Cn for standard interface (Apple IIGS Mouse not affected) Y = n0 for standard interface (Apple IIGS Mouse not affected)
Output:	A = mode if illegal mode entered, else 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 ($413)

Function:	Tests for interrupt from Mouse and resets Mouse's interrupt line
Input:	A, X, Y = not affected
Output:	X, Y, V, N, Z = scrambled C = 0 if it was a Mouse interrupt C = 1 if it was not a Mouse interrupt 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 affected) Y = n0 for standard interface (Apple IIGS Mouse not affected)
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 ($415)

Function:	Resets buttons, movement, and interrupt statusto 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 affected) Y = n0 for standard interface (Apple IIGS Mouse not affected)
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 affected) Y = n0 for standard interface (Apple IIGS Mouse not affected)
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 defaults are 0 to 1023 ($0000-$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 affected) Y = n0 for standard interface (Apple IIGS Mouse not affected)
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 performs an automatic HOMEMOUSE operation after a CLAMPMOUSE. The execution of a HOMEMOUSE is required because the delta information is being fed to the firmware instead of ±1's as in the case of the Apple II and the 6805 AppleMouse microprocessor cards. The delta information from 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 affected) Y = n0 for standard interface (Apple IIGS Mouse not affected)
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 defaults and sets clamping window to default of 0000-1023 ($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 affected)
Y = n0 for Standard interface (Apple IIGS Mouse not affected)

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. Interrupt status bits are valid only after a SERVEMOUSE. Interrupt status bits are reset after a 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.

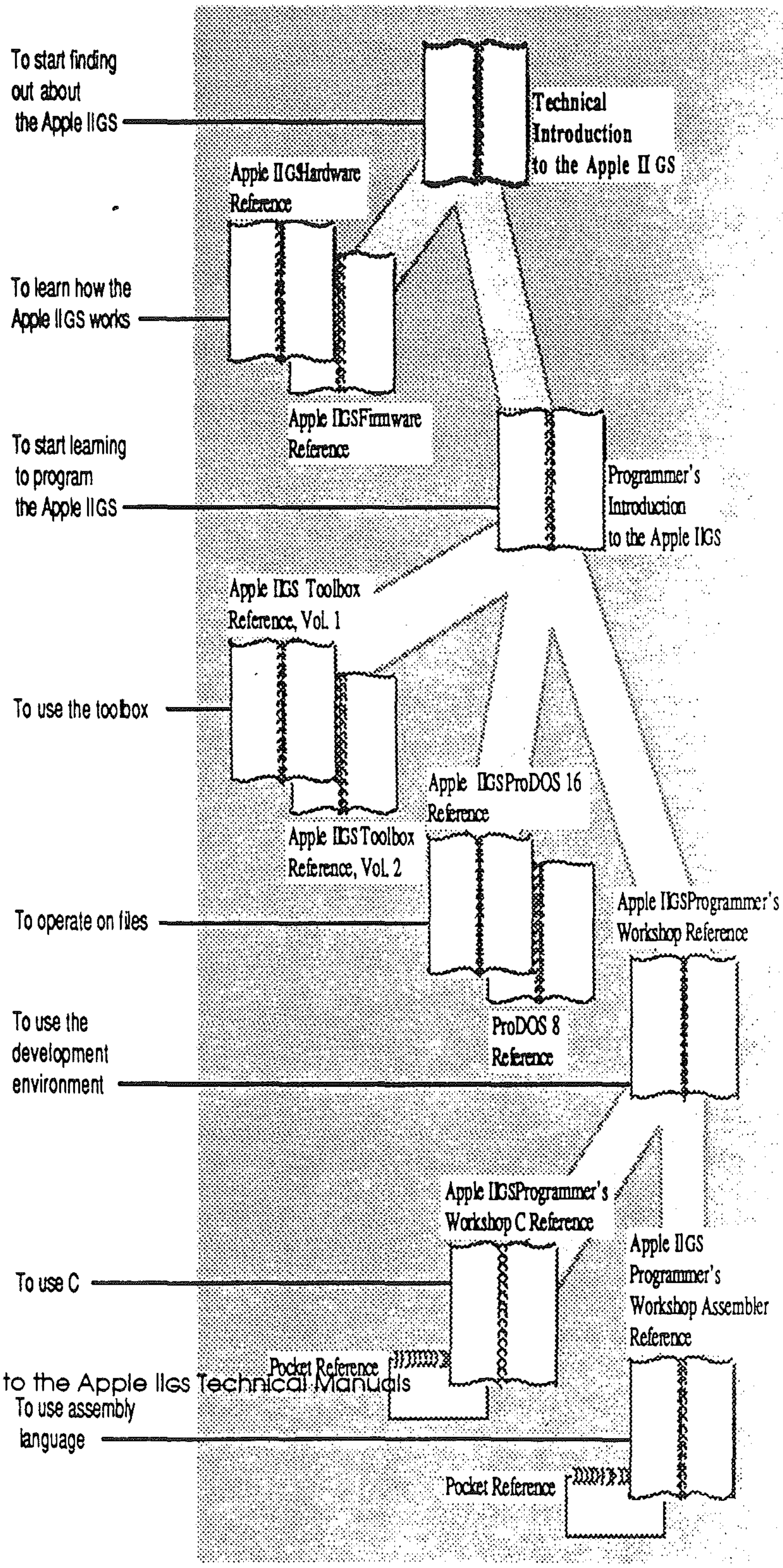
Table A-1
The Apple IIGS technical manuals

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 sepcifications
Apple IIGS Toolbox Reference: Volume 2	More toolbox specifications
Apple IIGS Programmer's Workshop Reference	The development environment

*Apple IIGS Workshop Assembler Reference**	Using the APW assembler
*Apple IIGS Workshop C Reference**	Using C on the Apple IIGS
ProDOS 8 Reference	ProDOS for Apple II programs
Apple IIGS ProDOS 16 Reference	ProDOS and Loader for Apple IIGS
Human Interface Guidelines	Guidelines for the desktop interface
Apple Numerics Manual	Numerics for all Apple computers

*There is a Pocket Reference for each of these.

Figure A-1
Roadmap to the technical manuals



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* 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 programs that use the Apple IIGS user interface (with windows, menus, and the mouse), 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 for the Apple IIGS. 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.

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* includes information about interrupt routines and low-level I/O subroutines for the serial ports, the disk port, and for the DeskTop Bus interface, which controls the keyboard and the mouse. The *Firmware Reference* also describes the Monitor, a low-level programming and debugging aid for assembly-language programs.

The toolbox 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. It also tells how to write and install your own tool set. The *Apple IIGS Toolbox Reference,* Volume 2, contains information about the rest of the tools.

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 manual

The development environment on the Apple IIGS is the Apple IIGS Programmer's Workshop (APW). APW is a set of programs that enable developers to create and debug application programs on the Apple IIGS. The *Apple IIGS Programmer's Workshop Reference* includes information about the parts of the workshop that all developers will use, regardless which programming language they use: the shell, the editor, the linker, the debugger, and the utilities. The manual also tells how to write other programs, such as custom utilities and compilers, to run under the APW Shell.

The APW reference manual describes the way you use the workshop to create an application and includes a sample program to show how this is done.

Programming-language manuals

Apple is currently providing 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 write a program in that language. The manuals for the languages Apple provides are the *Apple IIGS Workshop Assembler Reference* and the *Apple IIGS Workshop C Reference.*

Operating-system 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 Reference* and *Apple IIGS ProDOS 16 Reference*. ProDOS 16 uses the full power of the Apple IIGS and is not compatible with earlier Apple II's. The ProDOS 16 manual 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.*

ProDOS 8, previously just called *ProDOS*, is compatible with the models of Apple II that use 8-bit CPUs. As a developer of Apple IIGS programs, you need to use ProDOS 8 only if you are developing programs to run on 8-bit Apple II's as well as on the Apple IIGS.

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* and *Apple Numerics Manual.* If you develop programs for any Apple computer, you should know about those manuals.

The *Human Interface Guidlines* manual describes Apple's standards for the desktop interface of programs that run on Apple computers. If you are writing an application for the Apple IIGS, you should be 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 floating-point arithmetic. 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 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. A description of the version of the SANE routines for the 65C816 is available through the Apple Programmer's and Developer's Association, administered by the A.P.P.L.E. cooperative in Renton, Washington.

v *Note:* The address of the Apple Programmer's and Developer's Association is 290 SW 43rd Street, Renton, WA 98055, and the telephone number is (206) 251-6548.

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 below. 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)
II	$38	$60	$2F
II+	$EA	$EA	$EA
IIe	$06	$EA	$C1
IIe+	$06	$E0	$00
Apple IIGS	$06	$E0	$00
IIc	$06	$00	$FF
IIc+	$06	$00	$00

To distinguish the Apple IIGS from a IIe, since 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 all Apple II's prior to the Apple IIGS
BCS  ITSAPPLE2E        ;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 Front Desk Bus	
	2	1, if system has SCC	
	1	1, if system has external slots	
	0	1, if system has internal ports	
B	15 - 0	Reserved	
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.

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 in any way, shape, or form. Use of the entry points in this document will assure 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 IIc machines.

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

- 'A' represents the lower eight bits of the accumulator
- 'B' represents the upper eight bits of the accumulator
- 'X' and 'Y' represent eight bit index registers
- 'DBR' represents the data bank register
- 'K' represents the program bank register
- 'P' represents the processor status register
- 'e' is the emulation mode bit

Warning: For ALL of the routines that are contained in this appendix, the following environment variables must be set with the values shown here

- 'e' bit must be set to 1
- 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

Here are the descriptions of the firmware routines that are supported as entry points now and for future models of the Apple II family, starting with the Apple IIGS.

Addr	Name	Description
$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-resolutions graphics display to black. If CLRSCR is called while the video display is in text mode, it fills the screen with inverse at signs (@) 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 (mixed mode clear of 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/low nibbles

$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/low nibbles

$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 is 0.

$F88C INSDS1.2 Do 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. The BRK opcode for compatibility reasons returns a length of $00 not $01 as one would expect it to.

Input: 'A' =Opcode to determine length of
'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-1 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 to determine length of '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 as hex PRNTYX prints the contents of the 'Y' and 'X' registers as a four-digit hexadecimal value. 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 as hex PRNTYX prints the contents of the 'A' and 'X' registers as a four-digit hexadecimal value. 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 as hex 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 three 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 two bytes. 2=add three bytes. 3=add four 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 are not changed
```

$F962 TEXT2COPY Enable/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 which, 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 will 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

Does a jump to the interrupt handling routines which handle emulation mode break and irqs. All registers are restored after the application RTI's at the end of its installed interrupt routines. Location $45 is not destroyed as in the][,][+ and original //e Apples.

Input: 'A' =?
'X' =?
'Y' =?

Output: Unchanged= 'A'/'X'/'Y'/'DBR'/'P'/'B'/'K'/'D'/'e'
Scrambled= nothing

$FA4C	BREAK	Old 6502 break handler

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

Input: 'A' =Assumes 'A' was stored at addr $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 out 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: Drops user into monitor after displaying information.

$FA62	RESET	Hardware reset handler RESET sets up all necessary warmstart parameters for Apple IIGS. It is called by the 65C816 reset vector stored in ROM in locations $FFFC/$FFFD. If normal warmstart then exits through user vectors at $03F2/$03F3. If coldstart then exits by attempting to startup 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, the direct 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 coldstart routine PWRUP does a partial reset of the system then attempts to startup 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, the direct 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 and AppleTalk have not been selected via the control panel as the startup device.) pointed to by LOC0 and LOC1 (addresses $00/$01). If a startup device can be found it JMPs to that cards ROM space. If no startup device can be found then the message 'Check Startup Device' appears on the screen. If RAM Disk or ROM Disk has been selected then the firmware JMPs to the Smart Port code which handles those startup devices. If slot 7 was selected then and AppleTalk is enabled in port 7 then 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.

$FAD7REGDSP	Display contents of registers

REGDSP displays all 65816 register contents stored by the firmware, displays various Apple IIGS memory state information including shadowing and also displays system speed.
Displayed values includes
'A'/'X'/'Y'/'K'/'DBR'/'S'/'D'/
'M'/'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 (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	Timeout paddle then read the hand controller PREAD4 verifies the paddle (hand controller) has timed out 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 screen 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 as INIT does. 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 then sets top of window to line 20 for 4 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) 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 does a screen clear 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, it falls through to KBDWAIT. If not, control is sent on to VIDOUT where the character is printed and 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, then control is sent on 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 which indicates this is a //e or later system. This byte is the same in the //c, the enhanced //c, the //e, the enhanced //e and 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 which indicates this is a enhanced //e or 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 which indicates this is an enhanced //e or 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 forty 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 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 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 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 advance 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 it 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, linefeed, forward and reverse space, etc., are vectored to 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	Back-space 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 the screen column 0 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 back to the left edge of the window and then goes to LF to get 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. Maintains cursor postion.
		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, then returns to the program that called it. The amount of 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 a 100% 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) then increments A1L/A1H

NXTA1 does a 16 bit compare 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 in Apple IIGS. It does nothing except an RTS back to the calling routine.

Input: 'A' =?
'X' =?
'Y' =?

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

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

RDKEY is the character input subroutine. It places the old Apple][inverse character flashing cursor on the display at the current cursor position and jumps to the subroutine FD10.

Input: 'A' =?
'X' =?
'Y' =?

Output: Unchanged= 'X'/'DBR'/'K'/'D'/'e'
Scrambled= 'Y'/'B'/'P'
Special= 'A'=key pressed (inputted 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 (inputted 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 (inputted character)

$FD1B	KEYIN	Read the keyboard

KEYIN is the 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's 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 under cursor
        'X'   =?
        'Y'   =?

Output: Unchanged=   'X'/'DBR'/'K'/'D'/'e'
        Scrambled=   'Y'/'B'/'P'
        Special=     'A'=key pressed (inputted
                     character)
```

$FD35	RDCHAR	Get an input character and process ESCape codes

RDKEY is the character input subroutine which also interprets the standard Apple ESCape sequences. It also 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 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 (inputted
                     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, 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 the 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 a 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 a 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 nybble of the accumulator as a single hexadecimal digit to the current output device.
		Input: 'A' =Lower nybble 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's 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 return ($8D), linefeed ($8C), backspace/left arrow ($88), right arrow ($95), bell ($87), and 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 a Apple IIGS or later system. If 'c' (carry) returns clear 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 offset into the source/destination buffers to start with.

Input: 'A' =?
'X' =?
'Y' =Offset into source/destination buffers to start with (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 in 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 the 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 the 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 the 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 the 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]['G'o entry point

GO begins execution of the code pointer 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 in Apple IIGS. It does nothing except an RTS back to the calling routine.

Input: 'A' =?
'X' =?
'Y' =?

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

$FEFD READ

Read a data from a cassette tape (OBSOLETE)

READ is an obsolete entry point in Apple IIGS. It does nothing except 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' to output device

PRERR sends the or '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 6502 register information in locations $45-$49 and clear 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 it 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 entry point to the monitor Set up video and keyboard as output and input devices. Set hex mode, do not beep and enter monitor at MONZ2. Does not return to caller. All monitor 65816 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 Clear decimal mode, beep bell and enter 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) 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 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 accum`ulator into A2L/A2H (addresses $3E/$3F). Exits into NXTCHR.

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

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 a non-hex digit is encountered. Exits into NXTCHR.

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

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 lower case ASCII values that appear in the input buffer (addresses $2xx).

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

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 RTSs 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)

Zero out MONMODE (address $31).

Input: 'A' =?
'X' =?
'Y' =?

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

Appendix D
Vectors

This appendix contains a list of 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 utilized to assure that there will be a common point of interface 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 with the new release, thereby maintaining system integrity.

Bank 00 page 3 vectors

$03F0-$03F1	BRKV	User BRK vector
	Address of the subroutine that handles BRK interrrupts. Normally points to OLDBRK (address $FA59) in the monitor ROM.	
$03F2-$03F3	SOFTEV	User soft entry vector for RESET
	Address of the subroutine that handles warm start (RESET pressed). Normally points to BASIC or the operating system.	
$03F4	PWREDUP	EOR of high byte of SOFTEV address
	PWREDUP=SOFTEV+1 EORed with the constant $A5. If PWREDUP does NOT equal SOFTEV+1 EORed with the constant $A5 the system does a cold start. If PWREDUP equals SOFTEV+1 EORed with the constant $A5 the system does a warm start.	
$03F5-$03F6-$3F7	AMPERV	Applesoft's '&' JMP vector
	Address of the subroutine that handles Applesofts '&' (ampersand) commands. Normally points to IORTS (address $FA58) in the monitor. $03F5 contains a JMP ($4C) opcode.	

$03F8-$03F9-$3FA	USRADR	User's Control-Y and Applesoft's USR function JMP vector

Address of the subroutine that handles user's Control-Y and Applesoft's USR function commands. Normally points to MON (address $FF65) in the monitor or to BASIC.SYSTEM's warm start address if PRODOS8 is loaded in. $03F8 contains a JMP ($4C) opcode.

$03FB-$03FC-$3FD	NMI	User NMI vector

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

$03FE-$03FF	IRQLOC	User IRQ vector

Address of the subroutine that operating systems or applications can change to gain access to IRQ interrrupts. Normally points to MON (address $FF65) in the monitor ROM or to the 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 //e and //c to move data blocks between main and auxiliary memory. For compatiblity 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 //e and //c to transfer control between main and auxiliary memory. For compatiblity reasons, Apple IIGS also supports this entry point if the 80 column firmware is enabled via the Control Panel. XFER assumes 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 to restore them after regaining control. Failure to do so will cause 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 which changes with each ROM release. Its value is not guaranteed. No program should make use of 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 which changes with each ROM release. Its value is not guaranteed. No program should make use of 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 which changes with each ROM release. Its value is not guaranteed. No program should make use of 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 which changes with each ROM release. Its value is not guaranteed. No program should make use of 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 which changes with each ROM release. Its value is not guaranteed. No program should make use of 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 which changes with each ROM release. Its value is not guaranteed. No program should make use of 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 which changes with each ROM release. Its value is not guaranteed. No program should make use of 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 which changes with each ROM release. Its value is not guaranteed. No program should make use of 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 which changes with each ROM release. Its value is not guaranteed. No program should make use of this value. This vector is pulled from the ROM and used whenever an emulation mode NMI is executed.

$FFFE-$FFFF	EBRKIRQ	Emulation mode BRK/IRQ vector

This is not a callable routine. It is a 16-bit value which changes with each ROM release. Its value is not guaranteed. No program should make use of 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 the 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 the 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 the 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 the 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 handler/manager

Unconditional jump to the main system interrupt handler/manager. If the application patches out this vector it must be able to handle all interrupts in the same fashion as the built in

ROM interrupt handler/manager. If not 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 (co-processor) manager. Currently points to code which causes the monitor to printout 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 which causes the monitor to printout the instruction being executed's 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)

$E1/001C-001F SYSDMGRV Jump to system death manager

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

- Entry is in 16-bit native mode.
- 'c' (carry) =0 if user defined message pointed to on stack. =1 if use default messsage.
- 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 addr
5,S	=High byte of message addr
4,S	=Low byte of message addr
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

The following vectors IRQ.APTALK and IRQ.SERIAL are normally set up to point to internal interrupt handlers or to code which sets carry and RTL's 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 and/or setup 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, the D register, the 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 handler must not re-enable interrupts from within the interrupt handler. AppleTalk and the Desk Accessory Manager are allowable exceptions. These vectors should only be accessed via the miscellaneous tools. Their location in memory is not guaranteed.

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

Unconditional jump to AppleTalks 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 ports 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

The following vectors IRQ.SCAN through IRQ.OTHER are normally set up to point to internal interrupt handlers or to code which sets carry and RTL's back to the interrupt manager. All the routines are called in 8-bit native mode, high speed, 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 re-enable interrupts from within the interrupt handler. AppleTalk and the Desk Accessory Manager are allowable exceptions. These vectors should only be accessed via the miscellaneous tools. 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

↓ FE/98EE

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 interrupt 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 tools telling it to install a heartbeat task which every VBL time 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 the Desk Accessory interrupt handler

Unconditional jump to the Desk Accessory manager interrupt handler. Invoked by the user pressing Control-Open Apple-Escape.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 the keyboard FLUSH interrupt handler

Unconditional jump to the keyboard FLUSH interrupt handler. Invoked by the user pressing Control-Open Apple-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 can only occur if the keyboard micro had a catastrophic failure. If the failure does occur the firmware will try to resync 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 one second interrupt handler

Unconditional jump to the one 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 which 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 which 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 //. 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 is called in 8-bit native mode, high speed, data bank register set to $00, direct register set to $0000. The user must preserve the data bank register, direct register, speed and return in 8 bit native mode with an RTL. The users program 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 to be used 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 is called in 8-bit native mode, high speed, data bank register set to $00, direct register set to $0000. The user must preserve the data bank register, direct register, 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 sets carry the monitor firmware currently will print 'Trace' on the screen and continue where it left off. This vector is set up to be used in the future by the system firmware and in the present by 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 is called in 8 bit native mode, high speed, data bank register set to $00, direct register set to $0000. The user must preserve the data bank register, direct register, 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 sets carry the monitor firmware currently will print 'Step' on the screen and continue where it left off. This vector is set up to be used in the future by the system firmware and in the present by 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 vector.

This vector is reserved for future system expansion and is not available for 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

The 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 which copies the BATTERYRAM buffer in bank E1 to the clock chip's BATTERYRAM with proper checksums. This routine is called the miscellaneous tools and by the Control Panel programs.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 which copies the clock chip's BATTERYRAM to the BATTERYRAM buffer in bank E1, compares the checksums and if they match just returns to the caller. If they 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's BATTERYRAM with proper checksums. This routine is called the miscellaneous tools and by the Control Panel programs.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 which 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 tools only. It returns carry clear if the write was successful and carry set if 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 which 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 tools only. It returns carry clear if the read was successful and carry set if 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 Accessory Manager. It uses most of zero page. It RTLs back to the Desk Accessory 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	Setup system to BATTERYRAM parameters routine

This vector points to a routine which 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).

Note: BATTERYRAM buffer E1 values can be set via the miscellaneous tools 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 which 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 (three byte pointer)). It then uses that address to get the string to display. This routine is used by the built in Control Panel, 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 which 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 (three byte pointer)). It then uses that address to get the string to display. This routine is used by the built in Control Panel, 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 is called in 8-bit native mode, data bank register set to $00, direct register set to $0000. The user must preserve the data bank register, direct register, 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 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 Accessory Manager. It RTLs back to the Desk Accessory 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 three byte vector points to the address pointer table which points to ASCII strings which are 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 only change the contents of a soft switch by using the appropriate tool from the Tool Set. Please refer to the *Apple IIGS Tool Set Manual* for more information.

Address	Name			Explanation
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 alt. 48K RAM
C004:00	26 WRMAINRAM	DFB	0	;write to main 48K RAM
C005:00	27 WRCARDRAM	DFB	0	;write to alt. 48K RAM
C006:00	28 SETSLOTCXROM	DFB	0	;use ROMS 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 alt. 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 key pressed flag
C011:00	39 RDLCBNK2	DFB	0	;Bit 7=1 if LC bank 2 is in
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 alt 48K
C014:00	42 RDRAMWRT	DFB	0	;Bit 7=1 if writing alt 48K
C015:00	43 RDCXROM	DFB	0	;Bit 7=1 if using int 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 char set in use
C01F:00	53 RD80VID	DFB	0	;Bit 7=1 if 80 column hardware in

```
C020:00                  DFB  0        ;Reserved for future system
                                        expansion

C021:        * _7________6_____5_____4_____3_____2_____1_____0__
C021:        *|         |     |     |     |     |     |     |     |
C021:        *|Enable   |     |     |     |     |     |     |     |
C021:        *|color/   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
C021:        *| mono    |     |     |     |     |     |     |     |
C021:        *|_________|_____|_____|_____|_____|_____|_____|_____|
C021:        *       ^^^^^ MONOCOLOR status byte ^^^^^

C021:        *   MONOCOLOR bits defined as follows:
C021:        *   bit 7= 0 enables color -- 1 disables color
C021:        *   bit 6,5,4,3,2,1,0  Must be 0
C021:00        MONOCOLOR    DFB  0     ;Monochrome/color select register

C022:        * _7______6______5______4______3______2______1______0___
C022:        *|                           |                          |
C022:        *|                           |                          |
C022:        *|   Text Color Bits         |  Background Color Bits   |
C022:        *|                           |                          |
C022:        *|___________________________|__________________________|
C022:        *        ^^^^^ TBCOLOR byte ^^^^^

C022:        *   TBCOLOR bits defined as follows:
C022:        *   bit 7,6,5,4 = Text color bits
C022:        *   bits 3,2,1,0 = Background color bits
C022:        *
C022:        *   Color bits =
C022:        *         $0 = Black
C022:        *         $1 = Deep Red
C022:        *         $2 = Dark Blue
C022:        *         $3 = Purple
C022:        *         $4 = Dark Green
C022:        *         $5 = Dark Gray
C022:        *         $6 = Medium Blue
C022:        *         $7 = Light Blue
C022:        *         $8 = Brown
C022:        *         $9 = Orange
C022:        *         $A = Light Gray
C022:        *         $B = Pink
C022:        *         $C = Green
C022:        *         $D = Yellow
C022:        *         $E = Aquamarine
C022:        *         $F = White
C022:00          TBCOLOR    DFB  0     ;Text/background color select register
```

```
C023:      * __7______6______5______4______3______2______1______0__
C023:      * |      |      |      |      |      |      |      |    |
C023:      * | VGC  | 1sec | Scan | Ext  |      | 1sec | Scan | Ext|
C023:      * | int  | int  | int  | int  |  0   | int  | int  | int|
C023:      * |active|active|active|active|      |enable|enable|enab|
C023:      * |______|______|______|______|______|______|______|____|
C023:      *        ^^^^^ VGCINT status byte ^^^^^

C023:      * VGCINT bits defined as follows:
C023:      * bit 7= 1 if interrupt generated by VGC
C023:      * bit 6= 1 if 1 second timer interrupt
C023:      * bit 5= 1 if scan line interrupt
C023:      * bit 4= 1 if external interrupt (Forced low in Apple IIGS)
C023:      * bit 3= Must be 0
C023:      * bit 2= 1 second timer interrupt enable
C023:      * bit 1= scan line interrupt enable
C023:      * bit 0= ext int enable (Can't cause an int in Apple IIGS)
C023:00      VGCINT     DFB  0      ;VGC interrupt register


C024:      * __7______6______5______4______3______2______1______0__
C024:      * |      |      |      |      |      |      |      |    |
C024:      * |Button|      |                                       |
C024:      * |status|Delta |          Delta movement               |
C024:      * | now  | sign |                                       |
C024:      * |______|______|______|______|______|______|______|____|
C024:      *        ^^^^^ MOUSEDATA byte ^^^^^

C024:      * MOUSEDATA bits defined as follows:
C024:      * bit 7= if reading X data = button 1 status
C024:      *        if reading Y data = button 0 status
C024:      * bit 6= sign of delta 0='+' -- 1='-'
C024:      * bit 5,4,3,2,1,0 = delta movement
C024:00      MOUSEDATA DFB  0       ;X or Y mouse data register


C025:      * __7______6______5______4______3______2______1______0__
C025:      * |      |      |Update|      |      |      |      |    |
C025:      * | Open |Closed| mod  |Keypad|Repeat| Caps | Ctrl |Shft|
C025:      * |Apple | Apple|no key| key  |active| lock | key  | key|
C025:      * | key  | key  | press|active|      |active|active|actv|
C025:      * |______|______|______|______|______|______|______|____|
C025:      *        ^^^^^ KEYMODREG status byte ^^^^^

C025:      * KEYMODREG bits defined as follows:
C025:      * bit 7= Open Apple key active
C025:      * bit 6= Closed Apple key active
C025:      * bit 5= Updated modifier latch without keypress
C025:      * bit 4= Keypad key active
C025:      * bit 3= Repeat active
C025:      * bit 2= Caps lock active
```

```
C025:        * bit 1= Control key active
C025:        * bit 0= Shift key active
C025:00        KEYMODREG DFB  0       ;Key modifier register

C026:        * __7_______6_______5_______4_______3_______2_______1_______0_
C026:        *|       |       |       |       |       |       |       |     |
C026:        *|                                                             |
C026:        *|              Data to/from keyboard micro                    |
C026:        *|                                                             |
C026:        *|_______|_______|_______|_______|_______|_______|_______|_____|
C026:        *        ^^^^^ DATAREG byte ^^^^^

C026:        * DATAREG bits defined as follows:
C026:        * bit 7,6,5,4,3,2,1,0 = data to/from keyboard micro
C026:        *
C026:        * Data at interrupt time in this register is defined as:
C026:        * bit 7= Response byte if set, otherwise status byte
C026:        * bit 6= ABORT valid if set and all other bits reset
C026:        * bit 5= Desktop manager key sequence pressed
C026:        * bit 4= Flush buffer key sequence pressed
C026:        * bit 3= SRQ valid if set
C026:        * bit 2,1,0 If all bits clear then no FDB data valid,
C026:        * else the bits indicate the number of valid
C026:        * bytes received minus 1. (2-8 bytes total)
C026:00        DATAREG    DFB  0      ;Data register in KeyGlu chip

C027:        * __7_______6_______5_______4_______3_______2_______1_______0__
C027:        *|       |       |       |       |       |       |       |     |
C027:        *| Mouse | Mouse | Data  | Data  | Key   | Key   | Mouse |Cmd  |
C027:        *|  reg  |  int  |  reg  |  int  | data  | int   |X/Yreg |reg  |
C027:        *| full  |enable | full  |enable | full  |enable | data  |full |
C027:        *|_______|_______|_______|_______|_______|_______|_______|_____|
C027:        *      ^^^^^ KMSTATUS status byte ^^^^^

C027:        * KMSTATUS bits defined as follows:
C027:        * bit 7= 1 if mouse register full
C027:        * bit 6= mouse interrupt disable/enable
C027:        * bit 5= 1 if data register full
C027:        * bit 4= data interrupt enable
C027:        * bit 3= 1 if key data full<---NEVER USE, WON'T WORK
C027:        * bit 2= key data interrupt enable<---NEVER USE, WON'T WORK
C027:        * bit 1= 0 = mouse 'X' register data available
C027:        *        1 = mouse 'Y' register data available
C027:        * bit 0= Command register full
C027:00        KMSTATUS  DFB  0       ;Keyboard/mouse status register
C028:00        ROMBANK   DFB  0       ;ROM bank select toggle
                                      (Not used in Apple IIGS)
```

```
C029:      * __7______6______5______4______3______2______1______0__
C029:      *|      |      |      |      |      |      |      |    |
C029:      *|Enable|Linear| B/W  |      |      |      |      |Enab|
C029:      *| super| video|Color |  0   |  0   |  0   |  0   |bk 1|
C029:      *|hi-res|      |DHires|      |      |      |      |ltch|
C029:      *|______|______|______|______|______|______|______|____|
C029:      *        ^^^^^ NEWVIDEO byte ^^^^^

C029:      *  NEWVIDEO bits defined as follows:
C029:      *  bit 7= 1=disable Apple //e video (enables super hi-res)
C029:      *  bit 6= 1 to linearize for super hi-res
C029:      *  bit 5= 0 for color double hi-res -- 1 for B/W hi-res
C029:      *  bit 4,3,2,1= MUST be pregrammed as 0
C029:      *  bit 0= Enable bank 1 latch to allow long instructions
C029:      *          to access bank 1 directly.  Set by monitor only.
           *          User must NOT change this bit.
C029:00       NEWVIDEO  DFB  0       ;Video/enable read alt mem
                                     with long instructions
C02A:00                 DFB  0       ;Reserved for future system
                                     expansion


C02B:      * __7______6______5______4______3______2______1______0__
C02B:      *|      |      |      |      |      |      |      |    |
C02B:      *| Character Generator| NTSC/| Lang |      |      |    |
C02B:      *|   language select  |  PAL |select|  0   |  0   |  0 |
C02B:      *|      |      |      |      | bit  |      |      |    |
C02B:      *|______|______|______|______|______|______|______|____|
C02B:      *         ^^^^^ LANGSEL byte ^^^^^

C02B:      *  LANGSEL bits defined as follows:
C02B:      *  bit 7,6,5= Character generator language select
C02B:      *        Primary language   Secondary language
C02B:      *       $0 =   USA                 Dvorak
C02B:      *       $1 =   UK                  USA
C02B:      *       $2 =   French              USA
C02B:      *       $3 =   Danish              USA
C02B:      *       $4 =   Spanish             USA
C02B:      *       $5 =   Italian             USA
C02B:      *       $6 =   German              USA
C02B:      *       $7 =   Swedish             USA
C02B:      *  bit 4= 0 if NTSC video mode -- 1 if PAL video mode
C02B:      *  bit 3= LANGUAGE switch bit 0 if primary lang set selected
C02B:      *  bit 2,1,0 ;MUST be programmed as 0's
C02B:00       LANGSEL   DFB  0       ;Language/PAL/NTSC select register
C02C:00       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 *  bit 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 a 0 here to reset 1 second interrupt
```

```
C032:          * bit 5= write a 0 here to clear scan line interrupt
C032:          * bit 4= Must be 0
C032:          * bit 3= Must be 0
C032:          * bit 2= Must be 0
C032:          * bit 1= Must be 0
C032:          * bit 0= Must be 0
C032:00          SCANINT    DFB  0        ;Scan line interrupt register

C033:          * __7_______6_______5_______4_______3_______2_______1_______0__
C033:          *|       |       |       |       |       |       |       |     |
C033:          *|                                                             |
C033:          *|                   Clock data register                       |
C033:          *|                                                             |
C033:          *|_______|_______|_______|_______|_______|_______|_______|_____|
C033:          *         ^^^^^ CLOCKDATA byte ^^^^^

C033:          * CLOCKDATA bits defined as follows:
C033:          * bit 7,6,5,4,3,2,1,0 -- Data passed to/from clock chip
C033:00          CLOCKDATA DFB  0        ;Clock data register

C034:          * __7_______6_______5_______4_______3_______2_______1_______0__
C034:          *|       |       |       |       |       |       |       |     |
C034:          *| Clock| Read/ |Chip   |       |                             |
C034:          *| xfer |Write  |enable|   0   |        Border Color         |
C034:          *|      | chip  |assert|       |                             |
C034:          *|_______|_______|_______|_______|_______|_______|_______|_____|
C034:          *         ^^^^^ CLOCKCTL byte ^^^^^

C034:          * CLOCKCTL bits defined as follows:
C034:          * bit 7= Set =1 to start transfer to clock
C034:          *        Read =0 when transfer to clock is complete
C034:          * bit 6= 0= write to clock chip -- 1= read from clock chip
C034:          * bit 5= Clk chip enable asserted after transfer 0=no/1=yes
C034:          * bit 4= Must be 0
C034:          * bit 3,2,1,0=select border color (see TBCOLOR for values)
C034:00          CLOCKCTL   DFB  0        ;Clock control register

C035:          * __7_______6_______5_______4_______3_______2_______1_______0__
C035:          *|       |       |       |       |       |       |       |     |
C035:          *|       | Stop  |       | Stop  | Stop  | Stop  | Stop  |Stop |
C035:          *|   0   |I/O/LC|   0   |auxh-r|suprhr|hires2|hires1|txpg |
C035:          *|       |shadow|       |shadow|shadow|shadow|shadow|shad |
C035:          *|_______|_______|_______|_______|_______|_______|_______|_____|
C035:          *         ^^^^^ SHADOW byte ^^^^^

C035:          * SHADOW bits defined as follows:
C035:          * bit 7= Must write 0
C035:          * bit 6= 1 to inhibit I/O and language card operation
C035:          * bit 5= Must write 0
C035:          * bit 4= 1 to inhibit shadowing aux hi-res page
C035:          * bit 3= 1 to inhibit shadowing 32k video buffer
```

```
C035:          * bit 2= 1 to inhibit shadowing hires page 2
C035:          * bit 1= 1 to inhibit shadowing hires page 1
C035:          * bit 0= 1 to inhibit shadowing text pages
C035:00          SHADOW     DFB  0        ;Shadow register

C036:          * __7______6______5______4______3______2______1______0__
C036:          * |      |      |      |      |      |      |      |    |
C036:          * | Slow/|      |      |Shadow|Slot 7|Slot 6|Slot 5|Slt4|
C036:          * | fast |  0   |  0   |in all|motor |motor | motor|motr|
C036:          * | speed|      |      | RAM  |detect|detect|detect|dtct|
C036:          * |______|______|______|______|______|______|______|____|
C036:          *        ^^^^^ CYAREG byte ^^^^^

C036:          * CYAREG bits defined as follows:
C036:          * bit 7= 0=slow system speed -- 1=fast system speed
C036:          * bit 6= Must write 0
C036:          * bit 5= Must write 0
C036:          * bit 4= Shadow in all RAM banks (Never to be used!!)
C036:          * bit 3= Slot 7 disk motor on detect (Set by monitor only)
C036:          * bit 2= Slot 6 disk motor on detect (Set by monitor only)
C036:          * bit 1= Slot 5 disk motor on detect (Set by monitor only)
C036:          * bit 0= Slot 4 disk motor on detect (Set by monitor only)
C036:00          CYAREG     DFB  0        ;Speed and motor on detect
C037:00          DMAREG     DFB  0        ;Used during DMA as bank address
C038:00          SCCBREG    DFB  0        ;SCC channel B cmd register
C039:00          SCCAREG    DFB  0        ;SCC channel A cmd register
C03A:00          SCCBDATA   DFB  0        ;SCC channel B data register
C03B:00          SCCADATA   DFB  0        ;SCC channel A data register

C03C:          * __7______6______5______4______3______2______1______0__
C03C:          * |      |      |      |      |      |      |      |    |
C03C:          * | Busy | Auto |Access|      |                         |
C03C:          * | flag | inc  | doc/ |  0   |        Volume DAC       |
C03C:          * |      |adrptr| RAM  |      |                         |
C03C:          * |______|______|______|______|______|______|______|____|
C03C:          *        ^^^^^ SOUNDCTL byte ^^^^^

C03C:          * SOUNDCTL bits defined as follows:
C03C:          * bit 7= 0 if not busy -- 1 if busy
C03C:          * bit 6= 0=disable auto incrementing of address pointer
C03C:          *        1=enable auto incrementing of address pointer
C03C:          * bit 5= 0=access doc -- 1=access RAM
C03C:          * bit 4= Must be 0
C03C:          * bit 3,2,1,0=volume DAC-$0/$F=low/full volume (write only)
C03C:00          SOUNDCTL   DFB  0        ;Sound control register
```

```
C03D:      * __7______6______5______4______3______2______1______0__
C03D:      *|      |      |      |      |      |      |      |      |
C03D:      *|                                                       |
C03D:      *|              Sound data read/written                  |
C03D:      *|                                                       |
C03D:      *|______|______|______|______|______|______|______|______|
C03D:      *      ^^^^^ SOUNDDATA byte ^^^^^

C03D:      * SOUNDDATA bits defined as follows:
C03D:      * bit 7,6,5,4,3,2,1,0 = Data read from/written to sound RAM
C03D:00      SOUNDDATA DFB  0        ;Sound data register


C03E:      * __7______6______5______4______3______2______1______0__
C03E:      *|      |      |      |      |      |      |      |      |
C03E:      *|                                                       |
C03E:      *|         Low byte of sound address pointer             |
C03E:      *|                                                       |
C03E:      *|______|______|______|______|______|______|______|______|
C03E:      *      ^^^^^ SOUNDADRL byte ^^^^^

C03E:      * SOUNDADRL bits defined as follows:
C03E:      * bit 7,6,5,4,3,2,1,0 = Address into sound RAM low byte
C03E:00      SOUNDADRL DFB  0        ;Sound address pointer, low byte


C03F:      * __7______6______5______4______3______2______1______0__
C03F:      *|      |      |      |      |      |      |      |      |
C03F:      *|                                                       |
C03F:      *|        High byte of sound address pointer             |
C03F:      *|                                                       |
C03F:      *|______|______|______|______|______|______|______|______|
C03F:      *      ^^^^^ SOUNDADRH byte ^^^^^

C03F:      * SOUNDADRH bits defined as follows:
C03F:      * bit 7,6,5,4,3,2,1,0 = Address into sound RAM high byte
C03F:00      SOUNDADRH DFB  0        ;Sound address pointer, high byte
C040:00                DFB  0        ;Reserved for future system
                                     expansion
```

Note: The Mega // mouse is not used in Apple IIGS as a mouse but the softswitches and functions are used. Therefore the user may not use the Mega // mouse softswitches.

```
C041:      * __7______6______5______4______3______2______1______0__
C041:      *|      |      |      |      |      |      |      |      |
C041:      *|      |      |      |Enable|Enable|Enable|Enable|Enab|
C041:      *|  0   |  0   |  0   |1/4sec| VBL  |switch| move |mous|
C041:      *|      |      |      | ints | ints | ints | ints |    |
C041:      *|______|______|______|______|______|______|______|____|
```

```
C041:      *      ^^^^^ INTEN byte ^^^^^

C041:      * INTEN bits defined as follows:
C041:      * bit 7= Must be 0
C041:      * bit 6= Must be 0
C041:      * bit 5= Must be 0
C041:      * bit 4= 1 to enable 1/4 second interrupts
C041:      * bit 3= 1 to enable VBL interrupts
C041:      * bit 2= 1 to enable Mega // mouse switch interrupts
C041:      * bit 1= 1 to enable Mega // mouse movement interrupts
C041:      * bit 0= 1 to enable Mega // mouse operation
C041:00      INTEN      DFB  0      ;Interrupt enable register
                                    (firmware use only)
C042:00                 DFB  0      ;Reserved for future system
                                    expansion
C043:00                 DFB  0      ;Reserved for future system
                                    expansion


C044:      * __7______6______5______4______3______2______1______0__
C044:      *|      |      |      |      |      |      |      |     |
C044:      *|                                                      |
C044:      *|            Mega // mouse delta movement byte         |
C044:      *|                                                      |
C044:      *|______|______|______|______|______|______|______|_____|
C044:      *         ^^^^^ MMDELTAX byte ^^^^^

C044:      * MMDELTAX bits defined as follows:
C044:      * bit 7,6,5,4,3,2,1,0 = delta movement in 2's complement
             notation
C044:00    MMDELTAX     DFB  0      ;Mega // mouse delta X register


C045:      * __7______6______5______4______3______2______1______0_
C045:      *|      |      |      |      |      |      |      |     |
C045:      *|                                                      |
C045:      *|            Mega // mouse delta movement byte         |
C045:      *|                                                      |
C045:      *|______|______|______|______|______|______|______|_____|
C045:      *         ^^^^^ MMDELTAY byte ^^^^^

C045:      * MMDELTAY bits defined as follows:
C045:      * bit 7,6,5,4,3,2,1,0 = delta movement in 2's complement
             notation
C045:00      MMDELTAY DFB  0        ;Mega // mouse delta Y register

C046:      * __7______6______5______4______3______2______1______0__
C046:      *|      |      |      |      |      |      |      |    |
C046:      *|Self/ |MMouse|Status|Status|Status|Status|Status|Stat|
C046:      *|burnin| last |  AN3 |1/4sec| VBL  |switch| move |syst|
C046:      *| diags|button|      | int  | int  | int  | int  | IRQ|
C046:      *|______|______|______|______|______|______|______|____|
C046:      *          ^^^^^ 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 diags 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|Stat|
C046:          504 * |  now | last |  AN3 |1/4sec| VBL  |switch| move |syst|
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 1/4 second interrupted
C046:          514 *  bit 3= 1 if VBL interrupted
C046:          515 *  bit 2= 1 if Mega // mouse switch interrupted
C046:          516 *  bit 1= 1 if Mega // 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 // 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    SETAN3     DFB  0     ;Clear annunciator 3
C05F:00    CLRAN3     DFB  0     ;Set annunciator 3
C060:00    BUTN3      DFB  0     ;Read switch 3
C061:00    BUTN0      DFB  0     ;Read switch 0 (Open Apple key)
C062:00    BUTN1      DFB  0     ;Read switch 1 (Closed Apple key)
C063:00    BUTN2      DFB  0     ;Read switch 2
C064:00    PADDL0     DFB  0     ;Read paddle 0
C065:00               DFB  0     ;Read paddle 1
C066:00               DFB  0     ;Read paddle 2
C067:00               DFB  0     ;Read paddle 3

C068:      * __7______6______5______4______3______2______1______0__
C068:      *|      |      |      |      |      |      |      |    |
C068:      *| ALZP | PAGE2| RAMRD|RAMWRT| RDROM|LCBNK2| ROMB |INTCX|
C068:      *|status|status|status|status|status|status|status|stat|
C068:      *|      |      |      |      |      |      |      |    |
C068:      *|______|______|______|______|______|______|______|____|
C068:      *     ^^^^^ STATEREG status byte ^^^^^

C068:      *  STATEREG bits defined as follows:
C068:      *  bit 7= ALZP status
C068:      *  bit 6= PAGE2 status
C068:      *  bit 5= RAMRD status
C068:      *  bit 4= RAMWRT status
C068:      *  bit 3= RDROM status (Read only RAM/ROM (0/1))
C068:      *   IMPORTANT NOTE:
C068:      *       Do two reads to $C083 then change STATEREG
C068:      *       to change LCRAM/ROM banks (0/1) and still
C068:      *       have the language card write enabled.
C068:      *  bit 2= LCBNK2 status 0=LC bank 0 - 1=LC bank 1
C068:      *  bit 1= ROMBANK status
C068:      *  bit 0= INTCXROM status
C068:00    STATEREG   DFB  0     ;State register
C069:00               DFB  0     ;Reserved for future system
                                  expansion
C06A:00               DFB  0     ;Reserved for future system
                                  expansion
C06B:00               DFB  0     ;Reserved for future system
                                  expansion
C06C:00               DFB  0     ;Reserved for future system
                                  expansion
C06D:00    TESTREG    DFB  0     ;Test mode bit register
C06E:00    CLRTM      DFB  0     ;Clear test mode
C06F:00    ENTM       DFB  0     ;Enable test mode
C070:00     PTRIG     DFB  0     ;trigger the paddles
C071:                 DS   15,0  ;ROM interrupt code jump table
C080:00               DFB  0     ;Sel LC RAM bank2 rd, wrt
                                  protect LC RAM
C081:00    ROMIN      DFB  0     ;Enable ROM read, 2 reads
                                  wrt enb LC RAM
C082:00               DFB  0     ;Enable ROM read, wrt
                                  protect LC RAM
```

```
C083:00     593   LCBANK2   DFB  0      ;Sel LC RAM bank2, 2 rds
                                        wrt enb LC RAM
C084:00     595             DFB  0      ;Sel LC RAM bank2 rd, wrt
                                        protect LC RAM
C085:00     596             DFB  0      ;Enable ROM read, 2 reads
                                        wrt enb LC RAM
C086:00     597             DFB  0      ;Enable ROM read, wrt
                                        protect LC RAM
C087:00     598             DFB  0      ;Sel LC RAM bank2, 2 rds
                                        wrt enb LC RAM
C088:00     600             DFB  0      ;Sel LC RAM bank1 rd, wrt
                                        protect LC RAM
C089:00     601             DFB  0      ;Enable ROM read, 2 reads
                                        wrt enb LC RAM
C08A:00     602             DFB  0      ;Enable ROM read, wrt
                                        protect LC RAM
C08B:00     603   LCBANK1   DFB  0      ;Sel LC RAM bank1, 2 rds
                                        wrt enb LC RAM
C08C:00     605             DFB  0      ;Sel LC RAM bank1 rd, wrt
                                        protect LC RAM
C08D:00     606             DFB  0      ;Enable ROM read, 2 reads
                                        wrt enb LC RAM
C08E:00     607             DFB  0      ;Enable ROM read, wrt
                                        protect LC RAM
C08F:00     608             DFB  0      ;Sel LC RAM bank1, 2 rds
                                        wrt enb LC RAM
0000:610    DEND

0000:       612   CLRROM    EQU  $CFFF  ;Switch out $C8 ROMs
```

Table E-1. Symbol table sorted by symbol

C01E ALTCHARSET	C061 BUTN0	C062 BUTN1	C063 BUTN2
C060 BUTN3	C02C CHARROM	C034 CLOCKCTL	C033 CLOCKDATA
C000 CLR80COL	C00C CLR80VID	C00E CLRALTCHAR	C059 CLRAN0
C05B CLRAN1	C05D CLRAN2	C05F CLRAN3	CFFF CLRROM
C06E CLRTM	C047 CLRVBLINT	C048 CLRXYINT	C036 CYAREG
C026 DATAREG	C046 DIAGTYPE	C031 DISKREG	C037 DMAREG
C06F ENTM	C057 HIRES	C02F HORIZCNT	C041 INTEN
C046 INTFLAG	C000 IOADR	C010 KBDSTRB	C000 KBD
C025 KEYMODREG	C027 KMSTATUS	C02B LANGSEL	C08B LCBANK1
C083 LCBANK2	C056 LORES	C052 MIXCLR	C053 MIXSET
C044 MMDELTAX	C045 MMDELTAY	C021 MONOCOLOR	C024 MOUSEDATA
C029 NEWVIDEO	C064 PADDL0	C070 PTRIG	C018 RD80COL
C01F RD80VID	C016 RDALTZP	C017 RDC3ROM	C003 RDCARDRAM
C015 RDCXROM	C01D RDHIRES	C011 RDLCBNK2	C012 RDLCRAM
C002 RDMAINRAM	C01B RDMIX	C01C RDPAGE2	C013 RDRAMRD
C014 RDRAMWRT	C01A RDTEXT	C019 RDVBLBAR	C028 ROMBANK
C081 ROMIN	C032 SCANINT	C03B SCCADATA	C039 SCCAREG
C03A SCCBDATA	C038 SCCBREG	C001 SET80COL	C00D SET80VID
C00F SETALTCHAR	C009 SETALTZP	C058 SETAN0	C05A SETAN1
C05C SETAN2	C05E SETAN3	C00A SETINTC3ROM	C007 SETINTCXROM
C00B SETSLOTC3ROM	C006 SETSLOTCXROM	C008 SETSTDZP	C035 SHADOW
C02D SLTROMSEL	C03F SOUNDADRH	C03E SOUNDADRL	C03C SOUNDCTL
C03D SOUNDDATA	C030 SPKR	C068 STATEREG	C022 TBCOLOR
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 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 hardware configuration program that is ROM resident. It is invoked when the system is powered up if you press the Solid-Apple key. An alternate means of invoking the Control Panel is to do a cold start by holding down Control and Solid-Apple at the same time and Reset. The Desk Accessory Manager can also call the Control Panel and affect the values specified in this appendix.

Control Panel parameters

The following lists the selections and options available for each Control Panel menu. A checkmark (√) is used to indicate the default 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 1,200 1,800 2,400 3,600 4,800 7,200 9,600 √ 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 √ 1,200 1,800 2,400 3,600 4,800 7,200 19,200
Data bits	√ 8 7 6 5
Stop bits	√ 2 1
Parity	Odd Even √ None
DCD handshake	No √ Yes
DSR/DTR handshake	√ Yes No
XON/XOFF handshake	Yes √ No

Display: Selects all video specific options. Choosing type automatically causes color or monochrome selections to appear on the rest of the screen.

Options are as follows:

Line option	Choices
Type	√ Color Mono
Columns	√ 40 80

Hertz	√ 60 50

Color/Monochrome Selections

Text color	*(Color name is displayed)*
	black
	deep red
	dark blue
	purple
	dark green
	dark gray
	medium blue
	light blue
	brown
	orange
	light gray
	pink
	light green
	yellow
	aquamarine
	√ white
Text background	(Color name is displayed)
	black
	deep red
	dark blue
	purple
	dark green
	dark gray
	√ medium blue
	light blue
	brown
	orange
	light gray
	pink
	light green
	yellow
	aquamarine
	white

Border color	(Color name is displayed)
	black deep red dark blue purple dark green dark gray √ medium blue light blue brown orange light gray pink light green yellow aquamarine white
Standard colors	No √ Yes

(Standard colors indicates whether the user's chosen colors match theApple standard values. If the user selects Yes, in addition the current colors are switched to Apple standard colors.)

Sound: Allows system volume and pitch to be altered via an indicator bar. Default is in the middle of each range.

Speed/RAM disk: Allows default system speed of either normal speed, 1 mhz, or fast speed, 2.6/2.8 (RAM/ROM) mhz. Available options are

Option	**Choices**
System speed:	√ Fast Normal

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 is reached. 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 will be 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 commands for the RAM disk driver.

Total RAM in use: xxxxxK will be displayed on the screen. Total RAM in use equals total system RAM minus total free RAM.

Total free RAM disk will be displayed on the screen. You can determine the amount of total free RAM by calling the memory manager.

Slots:

Allows user to select either built-in device or peripheral card for slots 1, 2, 3, 4, 5, 6, and 7. Also allows the user to select start-up slot or to scan slots at start-up time. Options are available 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	√ Smart port Your card
Slot 6	√ Disk port Your card
Slot 7	√ Built-in AppleTalk Your card
Start-up 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 on the layouts and languages that are available comes from the keyboard micro at power-up time. Options are as follows:

Selection	**Choices**
Display language	Chosen from Table G-1
Keyboard layout	Chosen from Table G-1
Repeat speed	Indicator selects the following options: 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	Indicator selects the following options: .25 sec .50 sec √ .75 sec 1.00 sec No repeat
Double-click time	Indicator selects following options (1 tick = 1/60th of a second): xx ticks (slow) xx ticks √ xx ticks xx ticks xx ticks (fast)
Cursor flash rate	Indicator selects following options (1 tick = 1/60th of a second): xx ticks (slow) xx ticks √ xx ticks xx ticks xx 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
0	English (U. S. A.)
1	English (U. K.)
2	French
3	Danish
4	Spanish
5	Italian
6	German
7	Swedish
8	Dvorak
9	French Canadian
A	Flemish
B	Hebrew
C	Japanese
D	Arabic
E	Greek
F	Turkish
10	Finnish
11	Portuguese
12	Tamil
13	Hindi
14	T1
15	T2
16	T3
17	T4
18	T5
19	T6
1A	L1
1B	L2
1C	L3
1D	L4
1E	L5
1F	L6

(The keyboard microprocessor provides the pointer for the appropriate ASCII value listed in Table G-1) .

Clock: Allows the user 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
Hour (depends on Format selected)	1-12 or 0-23
Minute	0-59
Second	0-59
Format	√ AM-PM 24 hour

Quit: Returns you to calling application or, if called from keyboard, performs a start-up function.

Battery-powered RAM

The battery-powered RAM (called 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 selectable from the control panel. Standard setup values are placed into battery RAM during manufacturing. However, the keyboard layout and display language will be determined by the keyboard used.

Items changeable by manufacturing and the Control Panel program can be changed by the user's application program if desired; however, only an 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 the user 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 $E0/$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 banks $E0 or $E1, or enables a special soft switch, named *shadowing*, is turned on. When shadowing is enabled, whenever the Apple IIgs writes into bank $00 (or bank $01), the system automatically synchronizes with the Mega-II and writes the same data into bank $E0 (or bank $E1).

Figure H-1 depicts the layout of the memory in these banks of memory. Some of this memory is dedicated to display areas, some of it is reserved for firmware use; and some of it is declared as free space and is managed by the memory manager.

Figure H-1 shows the location of the various functions of Apple IIgs banks E0 and E1. In the figure, the notation *K* means a decimal value of 1024 bytes, and the notation *page* means hex $100 bytes.

> *Note:* In Figure H-1, the memory segments called *free space* are available **through the memory manager only.**

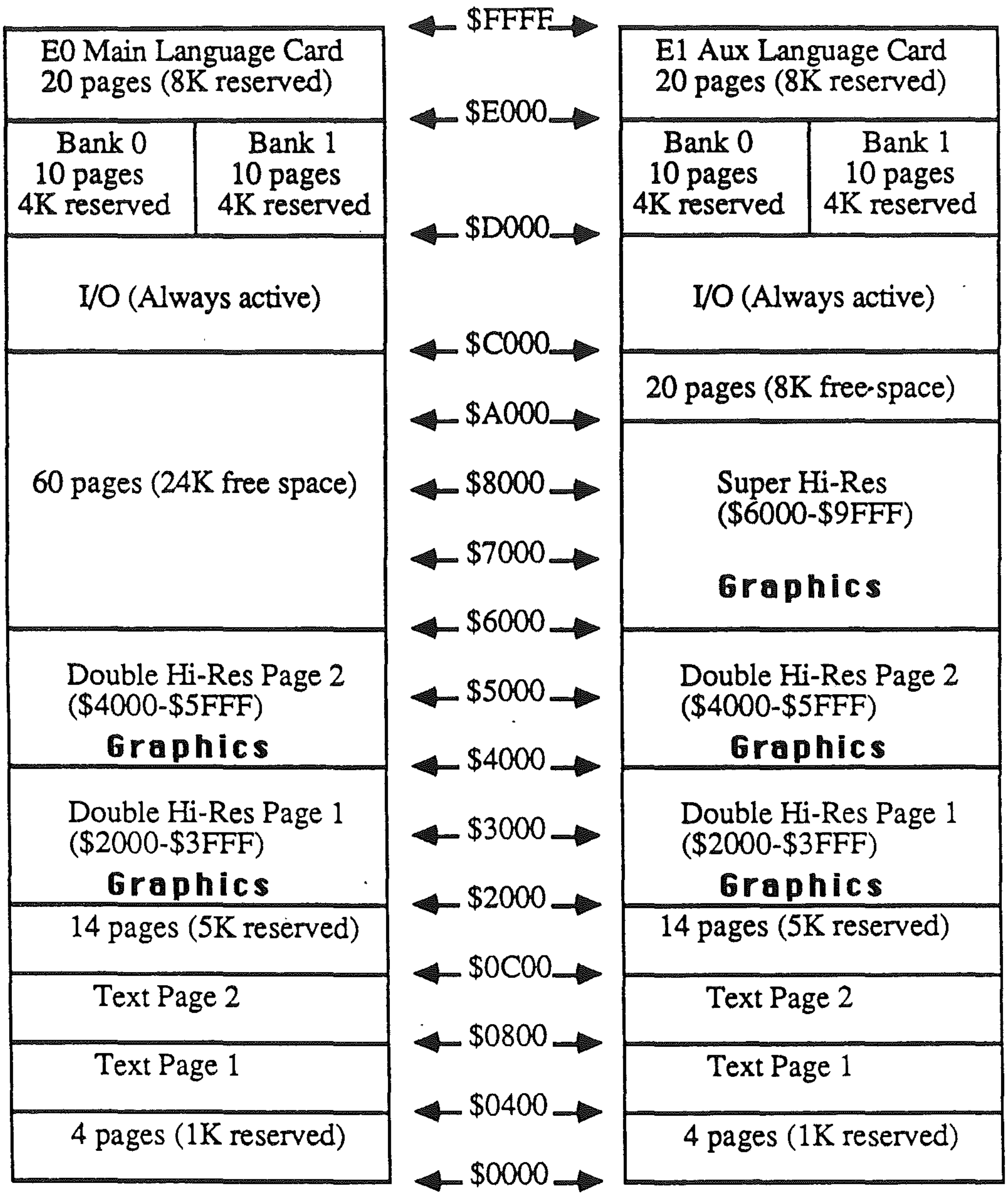


Figure H-1. Memory map of banks E0 and E1

Using banks E0 and E1

You can use graphics memory located in memory banks $E0 and $E1 or the free space via the memory manager; however, you must exercise caution to ensure that you don't use areas that are reserved for machine use.

Free space

Eighty 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 Apple IIGS tools.) If you try to use this space without first calling the Memory manager, you will cause a system crash.

Video buffers not needed for screen display may be used for your applications.

> *Note:* Video buffers are only used by firmware for video displays since 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 langauge 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 in Apple IIGS can be used by applications to display overlay data to 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 bank $00 and bank $01. To display the overlay, turn shadowing off, and write directly into banks $E0 and $E1. This only affects 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 will restore the display to its appearance before the overlay data was written.