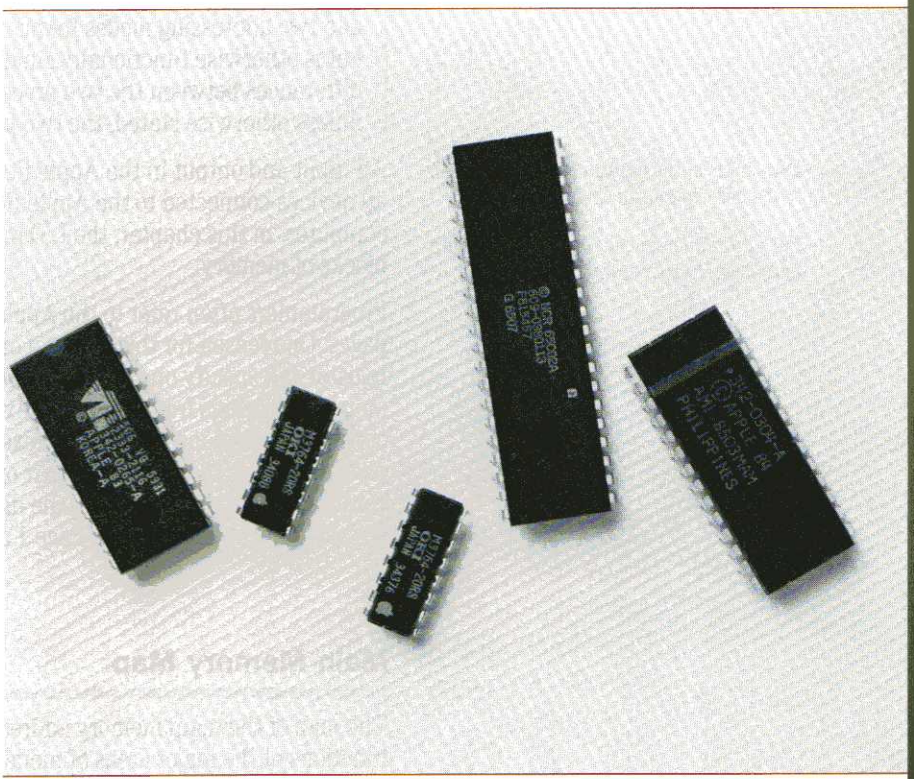
Chapter 4

Memory Organization



The Apple IIe's microprocessor can address 65,536 (64K) memory locations. All of the programmable storage (RAM and ROM) and input and output devices are allocated locations in this 64K address space. Some functions share the same addresses—but not at the same time.

For information about these shared address spaces, see the section "Bank-Switched Memory" in this chapter and the sections "Other Uses of I/O Memory Space" and "Expansion ROM Space" in Chapter 6.

Original IIe

The original version of the Apple IIe, as well as the Apple II Plus and Apple II, use the 6502 microprocessor. The 6502 lacks ten instructions and two addressing modes found on the 65C02 of the enhanced Apple IIe, but is otherwise functionally similar. For more information about the differences between the two processors, see Appendix A. In this manual, unless otherwise stated, the two processors are effectively the same.

For details of the built-in I/O features, refer to the descriptions in Chapters 2 and 3.

For information about I/O operations with peripheral cards, refer to Chapter 6.

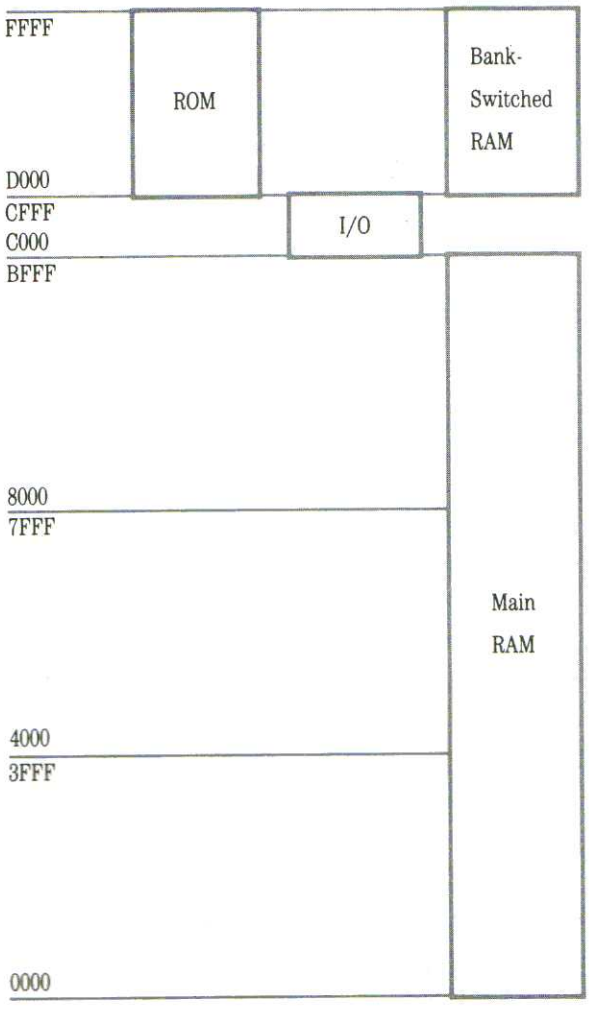
All input and output in the Apple IIe is memory mapped. This means that all devices connected to the Apple IIe appear to be memory locations to the computer. In this chapter, the I/O memory spaces are described simply as blocks of memory.

Programmers often refer to the Apple IIe's memory in 256-byte blocks called pages. One reason for this is that a one-byte address counter or index register can specify one of 256 different locations. Thus, page 0 consists of memory locations from 0 to 255 (hexadecimal $00 to $FF), inclusive. Page 1 consists of locations 256 to 511 (hexadecimal $0100 to $01FF); note that the page number is the high-order part of the hexadecimal address. Don't confuse this kind of page with the display buffers in the Apple IIe, which are sometimes referred to as Page 1 and Page 2.

Main Memory Map

The map of the main memory address space in Figure 4-1 shows the functions of the major areas of memory. For more details on the I/O space from 48K to 52K ($C000 through $CFFF), refer to Chapter 2 and Chapter 6; the bank-switched memory in the memory space from 52K to 64K ($D000 through $FFFF) is described in the section "Bank-Switched Memory" later in this chapter.

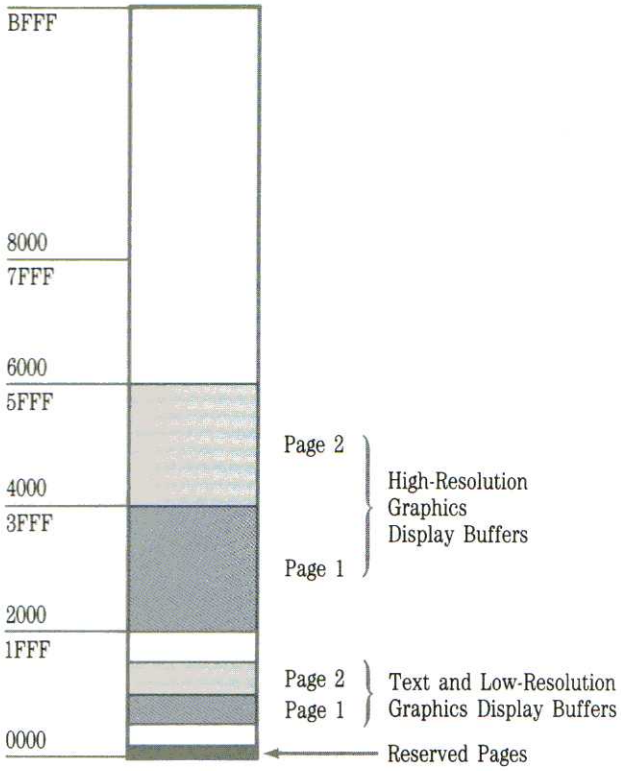
Figure 4-1. System Memory Map



RAM Memory Allocation

As Figure 4-1 shows, the major portion of the Apple IIe's memory space is allocated to programmable storage (RAM). Figure 4-2 shows the areas allocated to RAM. The main RAM memory extends from location 0 to location 49151 (hex $BFFF), and occupies pages 0 through 191 (hexadecimal $BF). There is also RAM storage in the bank-switched space from 53248 to 65535 (hexadecimal $D000 to $FFFF), described in the section "Bank-Switched Memory" later in this chapter, and auxiliary RAM, described in the section "Auxiliary Memory and Firmware" later in this chapter.

Figure 4-2. RAM Allocation Map



Reserved Memory Pages

Most of the Apple IIe's RAM is available for storing your programs and data. However, a few RAM pages are reserved for the use of the Monitor firmware and the BASIC interpreters. The reserved pages are described in the following sections.

Important! The system does not prevent your using these pages, but if you do use them, you must be careful not to disturb the system data they contain, or you will cause the system to malfunction.

Page Zero

Several of the 65C02 microprocessor's addressing modes require the use of addresses in page zero, also called zero page. The Monitor, the BASIC interpreters, DOS 3.3, and ProDOS all make extensive use of page zero.

To use indirect addressing in your assembly-language programs, you must store base addresses in page zero. At the same time, you must avoid interfering with the other programs that use page zero—the Monitor, the BASIC interpreters, and the disk operating systems. One way to avoid conflicts is to use only those page-zero locations not already used by other programs. Tables 4-1 through 4-5 show the locations in page zero used by the Monitor, Applesoft BASIC, Integer BASIC, DOS 3.3, and ProDOS.

As you can see from the tables, page zero is pretty well used up, except for a few bytes here and there. It's hard to find more than one or two bytes that aren't used by either BASIC, ProDOS, the Monitor, or DOS. Rather than trying to squeeze your data into an unused corner, you may prefer a safer alternative: save the contents of part of page zero, use that part, then restore the previous contents before you pass control to another program.

The 65C02 Stack

The 65C02 microprocessor uses page 1 as the stack—the place where subroutine return addresses are stored, in last-in, first-out sequence. Many programs also use the stack for temporary storage of the registers (via push and pull operations). You can do the same, but you should use it sparingly. The stack pointer is eight bits long, so the stack can hold only 256 bytes of information at a time. When you store the 257th byte in the stack, the stack pointer repeats itself, or wraps around, so that the new byte replaces the first byte stored, which is now lost. This writing over old data is called stack overflow, and when it happens, the program continues to run normally until the lost information is needed, whereupon the program terminates catastrophically.

The Input Buffer

The GETLN input routine, which is used by the Monitor and the BASIC interpreters, uses page 2 as its keyboard-input buffer. The size of this buffer sets the maximum size of input strings. (Note: Applesoft uses only the first 237 bytes, although it permits you to type in 256 characters.) If you know that you won't be typing any long input strings, you can store temporary data at the upper end of page 2.

Link-Address Storage

For more information about links, see the section "Changing the Standard I/O Links" in Chapter 6.

The Monitor, ProDOS, and DOS 3.3 all use the upper part of page 3 for link addresses or vectors.

BASIC programs sometimes need short machine-language routines. These routines are usually stored in the lower part of page 3.

The Display Buffers

See Chapter 6 for information on the memory locations that are reserved for peripheral cards.

The primary text and low-resolution-graphics display buffer occupies memory pages 4 through 7 (locations 1024 through 2047, hexadecimal $0400 through $07FF). This entire 1024-byte area is called text Page 1, and it is not usable for program and data storage. There are 64 locations in this area that are not displayed on the screen; these locations are reserved for use by the peripheral cards.

Text Page 2, the alternate text and low-resolution-graphics display buffer, occupies memory pages 8 through 11 (locations 2048 through 3071, hexadecimal $0800 through $0BFF). Most programs do not use Page 2 for displays, so they can use this area for program or data storage.

The primary high-resolution-graphics display buffer, called high-resolution Page 1, occupies memory pages 32 through 63 (locations 8192 through 16383, hexadecimal $2000 through $3FFF). If your program doesn't use high-resolution graphics, this area is usable for programs or data.

High-resolution Page 2 occupies memory pages 64 through 95 (locations 16384 through 24575, hexadecimal $4000 through $5FFF). Most programs use this area for program or data storage.

For more information about the display buffers, see the section "Video Display Pages" in Chapter 2.

The primary double-high-resolution-graphics display buffer, called double-high-resolution Page 1, occupies memory pages 32 through 63 (locations 8192 through 16383, hexadecimal $2000 through $3FFF) in both main and auxiliary memory. If your program doesn't use high-resolution or double-high-resolution graphics, this area of main memory is usable for programs or data.

Table 4-1. Monitor Zero-Page Use

High Nibble of Address	Low Nibble of Address															
	$0	**$1**	**$2**	**$3**	**$4**	**$5**	**$6**	**$7**	**$8**	**$9**	**$A**	**$B**	**$C**	**$D**	**$E**	**$F**
$00																
$10																•*
$20	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
$30	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
$40	•	•	•	•	•	•	•	•	•	•					•	•
$50	•	•	•	•	•	•										
$60																
$70																
$80																
$90																
$A0																
$B0																
$C0																
$D0																
$E0																
$F0																

* Byte used in original Apple IIe ROMs, now free.

Table 4-2. Applesoft Zero-Page Use

High Nibble of Address	Low Nibble of Address															
	$0	**$1**	**$2**	**$3**	**$4**	**$5**	**$6**	**$7**	**$8**	**$9**	**$A**	**$B**	**$C**	**$D**	**$E**	**$F**
$00	•	•	•	•	•	•					•	•	•	•	•	•
$10	•	•	•	•	•	•	•	•	•	•	•	•	•	•		
$20							•	•					•	•		•
$30	•		•	•									•	•	•	•
$40																
$50	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
$60	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
$70	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
$80	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
$90	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
$A0	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
$B0	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
$C0	•	•	•	•	•	•	•	•	•	•	•	•	•	•		
$D0	•	•	•	•	•	•	•		•	•	•	•	•	•	•	•
$E0	•	•	•	•	•	•	•	•	•	•	•					
$F0	•	•	•	•	•	•	•	•	•	•						•

Table 4-3. Integer BASIC Zero-Page Use

High Nibble of Address	Low Nibble of Address															
	$0	**$1**	**$2**	**$3**	**$4**	**$5**	**$6**	**$7**	**$8**	**$9**	**$A**	**$B**	**$C**	**$D**	**$E**	**$F**
$00														•		
$10																
$20																
$30																
$40											•	•	•	•		
$50						•	•	•	•	•	•	•	•	•	•	•
$60	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
$70	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
$80	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
$90	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
$A0	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
$B0	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
$C0	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
$D0	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
$E0																
$F0															•	•

Table 4-4. DOS 3.3 Zero-Page Use

High Nibble of Address	Low Nibble of Address															
	$0	**$1**	**$2**	**$3**	**$4**	**$5**	**$6**	**$7**	**$8**	**$9**	**$A**	**$B**	**$C**	**$D**	**$E**	**$F**
$00																
$10																
$20							•	•			•	•	•	•	•	•
$30						•	•	•	•	•					•	•
$40	•	•	•	•	•	•	•	•	•		•	•	•	•		
$50																
$60								•	•	•	•					•
$70	•															
$80																
$90																
$A0																•
$B0	•															
$C0											•	•	•	•		
$D0									•							
$E0																
$F0																

Table 4-5. ProDOS MLI and Disk-Driver Zero-Page Use

High Nibble of Address	Low Nibble of Address														
	$0	$1	$2	$3	$4	$5	$6	$7	$8	$9	$A	$B	$C	$D	$E	$F
$00	•	•														
$10																
$20																
$30											•	•	•	•	•	•
$40	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	
$50																
$60																
$70																
$80																
$90																
$A0																
$B0																
$C0																
$D0																
$E0																
$F0																

Bank-Switched Memory

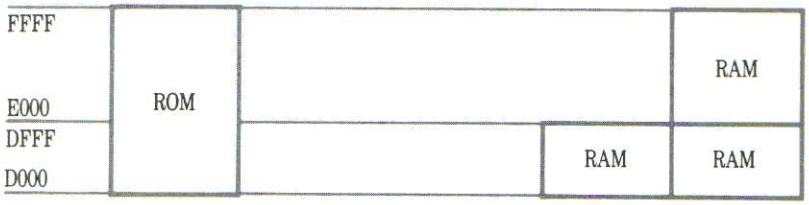
The memory address space from 52K to 64K (hexadecimal $D000 through $FFFF) is doubly allocated: it is used for both ROM and RAM. The 12K bytes of ROM (read-only memory) in this address space contain the Monitor and the Applesoft BASIC interpreter. Alternatively, there are 16K bytes of RAM in this space. The RAM is normally used for storing either the Integer BASIC interpreter or part of the Pascal Operating System (purchased separately).

You may be wondering why this part of memory has such a split personality. Some of the reasons are historical: the Apple IIe is able to run software written for the Apple II and Apple II Plus because it uses this part of memory in the same way they do. It is convenient to have the Applesoft interpreter in ROM, but the Apple IIe, like an Apple II with a language card, is also able to use that address space for other things when Applesoft is not needed.

You may also be wondering how 16K bytes of RAM is mapped into only 12K bytes of address space. The usual answer is that it's done with mirrors, and that isn't a bad analogy: the 4K-byte address space from 52K to 56K (hexadecimal $D000 through $DFFF) is used twice.

Switching different blocks of memory into the same address space is called bank switching. There are actually two examples of bank switching going on here: first, the entire address space from 52K to 64K ($D000 through $FFFF) is switched between ROM and RAM, and second, the address space from 52K to 56K ($D000 to $DFFF) is switched between two different blocks of RAM.

Figure 4-3. Bank-Switched Memory Map



Setting Bank Switches

You switch banks of memory in the same way you switch other functions in the Apple IIe: by using soft switches. Read operations to these soft switches do three things: select either RAM or ROM in this memory space; enable or inhibit writing to the RAM (write-protect); and select the first or second 4K-byte bank of RAM in the address space $D000 to $DFFF.

▲Warning

Do not use these switches without careful planning. Careless switching between RAM and ROM is almost certain to have catastrophic effects on your program.

Table 4-6 shows the addresses of the soft switches for enabling all combinations of reading and writing in this memory space. All of the hexadecimal values of the addresses are of the form $C08x. Notice that several addresses perform the same function: this is because the functions are activated by single address bits. For example, any address of the form $C08x with a 1 in the low-order bit enables the RAM for writing. Similarly, bit 3 of the address selects which 4K block of RAM to use for the address space $D000-$DFFF; if bit 3 is 0, the first bank of RAM is used, and if bit 3 is 1, the second bank is used.

When RAM is not enabled for reading, the ROM in this address space is enabled. Even when RAM is not enabled for reading, it can still be written to if it is write-enabled.

When you turn power on or reset the Apple IIe, it initializes the bank switches for reading the ROM and writing the RAM, using the second bank of RAM. Note that this is different from the reset on the Apple II Plus, which didn't affect the bank-switched memory (the language card). On the Apple IIe, you can't use the reset vector to return control to a program in bank-switched memory, as you could on the Apple II Plus.

> ***Reset With Integer BASIC:*** When you are using Integer BASIC on the Apple IIe, reset works correctly, restarting BASIC with your program intact. This happens because the reset vector transfers control to DOS, and DOS resets the switches for the current version of BASIC.

Table 4-6. Bank Select Switches

Note: R means read the location, W means write anything to the location, R/W means read or write, and R7 means read the location and then check bit 7.

Name	Action	Hex	Function
	R	$C080	Read RAM; no write; use $D000 bank 2.
	RR	$C081	Read ROM; write RAM; use $D000 bank 2.
	R	$C082	Read ROM; no write; use $D000 bank 2.
	RR	$C083	Read and write RAM; use $D000 bank 2.
	R	$C088	Read RAM; no write; use $D000 bank 1.
	RR	$C089	Read ROM; write RAM; use $D000 bank 1.
	R	$C08A	Read ROM; no write; use $D000 bank 1.
	RR	$C08B	Read and write RAM; use $D000 bank 1.
RDBNK2	R7	$C011	Read whether $D000 bank 2 (1) or bank 1 (0).
RDLCRAM	R7	$C012	Reading RAM (1) or ROM (0).
ALTZP	W	$C008	Off: use main bank, page 0 and page 1.
ALTZP	W	$C009	On: use auxiliary bank, page 0 and page 1.
RDALTZP	R7	$C016	Read whether auxiliary (1) or main (0) bank.

Reading and Writing to RAM Banks: Note that you can't read one RAM bank and write to the other; if you select either RAM bank for reading, you get that one for writing as well.

Reading RAM and ROM: You can't read from ROM in part of the bank-switched memory and read from RAM in the rest: specifically, you can't read the Monitor in ROM while reading bank-switched RAM. If you want to use the Monitor firmware with a program in bank-switched RAM, copy the Monitor from ROM (locations $F800 through $FFCB) into bank-switched RAM. You can't do this from Pascal or ProDOS.

To see how to use these switches, look at the following section of an assembly-language program:

```
AD 83 C0        LDA $C083       *SELECT 2ND 4K BANK & READ/WRITE
AD 83 C0        LDA $C083       *BY TWO CONSECUTIVE READS
A9 D0           LDA #$D0        *SET UP...
85 01           STA BEGIN       *...NEW...
A9 FF           LDA #$FF        *...MAIN-MEMORY...
85 02           STA END         *...POINTERS...
20 97 C9        JSR RAMTST      *...FOR 12K BANK

AD 8B C0        LDA $C08B       *SELECT 1ST 4K BANK
20 97 C9        JSR RAMTST      *USE ABOVE POINTERS

AD 83 C0        LDA $C088       *SELECT 1ST BANK & WRITE PROTECT
A9 80           LDA #$80
E6 10           INC TSTNUM
20 58 C9        JSR WPTSINIT

AD 80 C0        LDA $C080       *SELECT 2ND BANK & WRITE PROTECT
E6 10           INC TSTNUM
A9 01           LDA #PAT12K
20 58 C9        JSR WPTSINIT

AD 8B C0        LDA $C08B       *SELECT 1ST BANK & READ/WRITE
AD 8B C0        LDA $C08B       *BY TWO CONSECUTIVE READS
E6 0E           INC RWMODE      *FLAG RAM IN READ/WRITE
E6 10           INC TSTNUM
A9 08           LDA #PAT4K
20 58 C9        JSR WPTSINIT
```

The LDA instruction, which performs a read operation to the specified memory location, is used for setting the soft switches. The unusual sequence of two consecutive LDA instructions performs the two consecutive reads that write-enable this area of RAM; in this case, the data that are read are not used.

Reading Bank Switches

You can read which language card bank is currently switched in by reading the soft switch at $C011. You can find out whether the language card or ROM is switched in by reading $C012. The only way that you can find out whether the language card RAM is write-enabled or not is by trying to write some data to the card's RAM space.

Auxiliary Memory and Firmware

By installing an optional card in the auxiliary slot, you can add more memory to the Apple IIe. One such card is the Apple IIe 80-Column Text Card, which has 1K bytes of additional RAM for expanding the text display from 40 columns to 80 columns.

Another optional card, the Apple IIe Extended 80-Column Text Card, has 64K of additional RAM. A 1K-byte area of this memory serves the same purpose as the memory on the 80-Column Text Card: expanding the text display to 80 columns. The other 63K bytes can be used as auxiliary program and data storage. If you use only 40-column displays, the entire 64K bytes is available for programs and data.

▲Warning

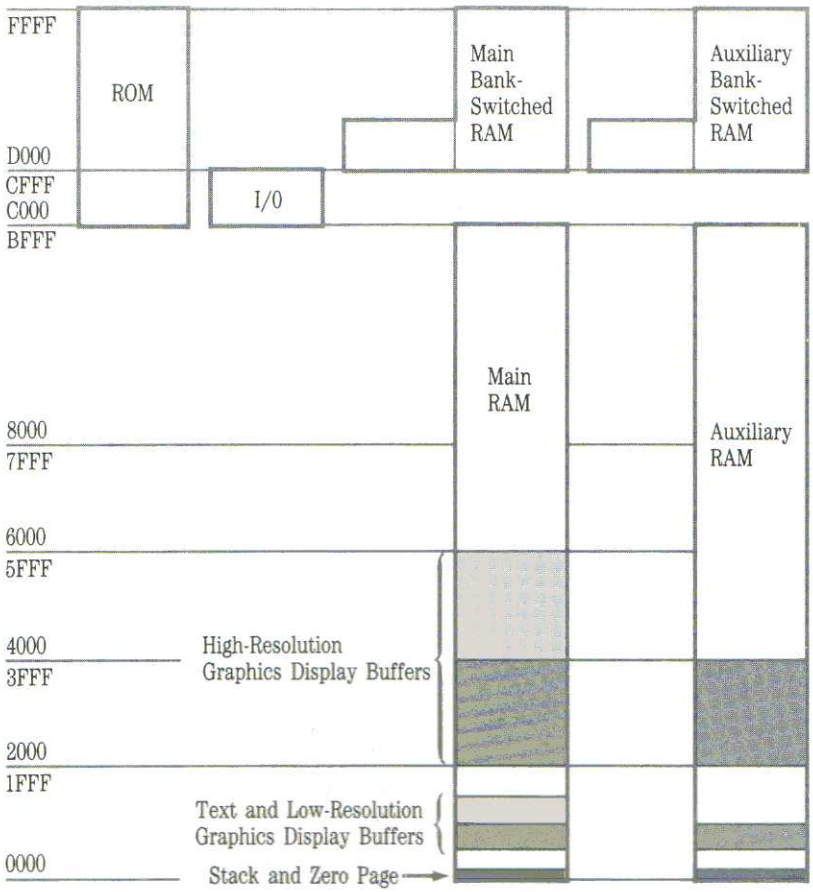
Do not attempt to use the auxiliary memory from a BASIC program. The BASIC interpreter uses several areas in main RAM, including the stack and the zero page. If you switch to auxiliary memory in these areas, the BASIC interpreter fails and you must reset the system and start over.

As you can see by studying the memory map in Figure 4-4, the auxiliary memory is broken into two large sections and one small one. The largest section is switched into the memory address space from 512 to 49151 ($0200 through $BFFF). This space includes the display buffer pages: as described in the section "Text Modes" in Chapter 2, space in auxiliary memory is used for one half of the 80-column text display. You can switch to the auxiliary memory for this entire memory space, or you can switch just the display pages: see the next section, "Memory Mode Switching."

Soft Switches: If the only reason you are using auxiliary memory is for the 80-column display, note that you can store into the display page in auxiliary memory by using the 80STORE and PAGE2 soft switches described in the section "Display Mode Switching" in Chapter 2.

The other large section of auxiliary memory is switched into the memory address space from 52K to 64K ($D000 through $FFFF). This memory space and the switches that control it are described earlier in this chapter in the section "Bank-Switched Memory." If you use the auxiliary RAM in this space, the soft switches have the same effect on the auxiliary RAM that they do on the main RAM: the bank switching is independent of the auxiliary-RAM switching.

Figure 4-4. Memory Map With Auxiliary Memory



Bank Switches: Note that the soft switches for the bank-switched memory, described in the previous section, do not change when you switch to auxiliary RAM. In particular, if ROM is enabled in the bank-switched memory space before you switch to auxiliary memory, the ROM will still be enabled after you switch. Any time you switch the bank-switched section of auxiliary memory in and out, you must also make sure that the bank switches are set properly.

When you switch in the auxiliary RAM in the bank-switched space, you also switch the first two pages, from 0 to 511 ($0000 through $01FF). This part of memory contains page zero, which is used for important data and base addresses, and page one, which is the 65C02 stack. The stack and zero page are switched this way so that system software running in the

bank-switched memory space can maintain its own stack and zero page while it manipulates the 48K address space (from $0200 to $BFFF) in either main memory or auxiliary memory.

Memory Mode Switching

Switching the 48K section of memory is performed by two soft switches: the switch named RAMRD selects main or auxiliary memory for reading, and the one named RAMWRT selects main or auxiliary memory for writing. As shown in Table 4-7, each switch has a pair of memory locations dedicated to it, one to select main memory, and the other to select auxiliary memory. Enabling the read and write functions independently makes it possible for a program whose instructions are being fetched from one memory space to store data into the other memory space.

▲Warning

Do not use these switches without careful planning. Careless switching between main and auxiliary memories is almost certain to have catastrophic effects on the operation of the Apple IIe. For example, if you switch to auxiliary memory with no card in the slot, the program that is running will stop and you will have to reset the Apple IIe and start over.

Writing to the soft switch at location $C003 turns RAMRD on and enables auxiliary memory for reading; writing to location $C002 turns RAMRD off and enables main memory for reading. Writing to the soft switch at location $C005 turns RAMWRT on and enables the auxiliary memory for writing; writing to location $C004 turns RAMWRT off and enables main memory for writing. By setting these switches independently, you can use any of the four combinations of reading and writing in main or auxiliary memory.

Auxiliary memory corresponding to text Page 1 and high-resolution graphics Page 1 can be used as part of the address space from $0200 to $BFFF by using RAMRD and RAMWRT as described above. These areas in auxiliary RAM can also be controlled separately by using the switches described in the section "Display Mode Switching" in Chapter 2. Those switches are named 80STORE, PAGE2, and HIRES.

As shown in Table 4-7, the 80STORE switch functions as an enabling switch: with it on, the PAGE2 switch selects main memory or auxiliary memory. With the HIRES switch off, the memory space switched by PAGE2 is the text Page 1, from $0400 to $07FF; with HIRES on, PAGE2 switches both text Page 1 and high-resolution graphics Page 1, from $2000 to $3FFF.

If you are using both the auxiliary-RAM control switches and the auxiliary-display-page control switches, the display-page control switches take priority: if 80STORE is off, RAMRD and RAMWRT work for the entire

memory space from $0200 to $BFFF, but if 80STORE is on, RAMRD and RAMWRT have no effect on the display page. Specifically, if 80STORE is on and HIRES is off, PAGE2 controls text Page 1 regardless of the settings of RAMRD and RAMWRT. Likewise, if 80STORE and HIRES are both on, PAGE2 controls both text Page 1 and high-resolution graphics Page 1, again regardless of RAMRD and RAMWRT.

A single soft switch named ALTZP (for alternate zero page) switches the bank-switched memory and the associated stack and zero page area between main and auxiliary memory. As shown in Table 4-7, writing to location $C009 turns ALTZP on and selects auxiliary-memory stack and zero page; writing to the soft switch at location $C008 turns ALTZP off and selects main-memory stack and zero page for both reading and writing.

The next section, "Auxiliary-Memory Subroutines," describes firmware that you can call to help you switch between main and auxiliary memory.

Table 4-7. Auxiliary-Memory Select Switches.

Name	Function	Location			Notes
		Hex	Decimal		
RAMRD	Read auxiliary memory	$C003	49155	-16381	Write
	Read main memory	$C002	49154	-16382	Write
	Read RAMRD switch	$C013	49171	-16365	Read
RAMWRT	Write auxiliary memory	$C005	49157	-16379	Write
	Write main memory	$C004	49156	-16380	Write
	Read RAMWRT switch	$C014	49172	-16354	Read
80STORE	On: access display page	$C001	49153	-16383	Write
	Off: use RAMRD, RAMWRT	$C000	49152	-16384	Write
	Read 80STORE switch	$C018	49176	-16360	Read
PAGE2	Page 2 on (aux. memory)	$C055	49237	-16299	*
	Page 2 off (main memory)	$C054	49236	-16300	*
	Read PAGE2 switch	$C01C	49180	-16356	Read
HIRES	On: access high-res. pages	$C057	49239	-16297	†
	Off: use RAMRD, RAMWRT	$C056	49238	-16298	†
	Read HIRES switch	$C01D	49181	-16355	Read
ALTZP	Auxiliary stack & z.p.	$C009	49161	-16373	Write
	Main stack & zero page	$C008	49160	-16374	Write
	Read ALTZP switch	$C016	49174	-16352	Read

* When 80STORE is on, the PAGE2 switch selects main or auxiliary display memory.

† When 80STORE is on, the HIRES switch enables you to use the PAGE2 switch to switch between the high-resolution Page-1 area in main memory or auxiliary memory.

When these switches are on, auxiliary memory is being used; when they are off, main memory is being used.

There are three more locations associated with the auxiliary-memory switches. The high-order bits of the bytes you read at these locations tell you the settings of the three soft switches described above. The byte you read at location $C013 has its high bit set to 1 if RAMRD is on (auxiliary memory is read-enabled), or 0 if RAMRD is off (the 48K block of main memory is read-enabled). The byte at location $C014 has its high bit set to 1 if RAMWRT is on (auxiliary memory is write-enabled), or 0 if RAMWRT is off (the 48K block of main memory is write-enabled). The byte at location $C016 has its high bit set to 1 if ALTZP is on (the bank-switched area, stack, and zero page in the auxiliary memory are selected), or 0 if ALTZP is off (these areas in main memory are selected).

Sharing Memory: In order to have enough memory locations for all of the soft switches and remain compatible with the Apple II and Apple II Plus, the soft switches listed in Table 4-7 share their memory locations with the keyboard functions listed in Table 2-2. The operations—read or write—shown in Table 4-7 for controlling the auxiliary memory are just the ones that are not used for reading the keyboard and clearing the strobe.

Auxiliary-Memory Subroutines

If you want to write assembly-language programs that use auxiliary memory but you don't want to manage the auxiliary memory yourself, you can use the built-in auxiliary-memory subroutines. These subroutines make it possible to use the auxiliary memory without having to manipulate the soft switches described in the previous section.

Important!

The subroutines described below make it easier to use auxiliary memory, but they do not protect you from errors. You still have to plan your use of auxiliary memory to avoid catastrophic effects on your program.

You use these built-in subroutines the same way you use the I/O subroutines described in Chapter 3: by making subroutine calls to their starting locations. Those locations are shown in Table 4-8.

Table 4-8. 48K RAM Transfer Routines

Name	Action	Hex	Function
AUXMOVE	JSR	$C312	Moves data blocks between main and auxiliary 48K memory.
XFER	JMP	$C314	Transfers program control between main and auxiliary 48K memory.

Moving Data to Auxiliary Memory

In your assembly-language programs, you can use the built-in subroutine named AUXMOVE to copy blocks of data from main memory to auxiliary memory or from auxiliary memory to main memory. Before calling this routine, you must put the data addresses into byte pairs in page zero and set the carry bit to select the direction of the move—main to auxiliary or auxiliary to main.

▲Warning

Don't try to use AUXMOVE to copy data in page zero or page one (the 65C02 stack) or in the bank-switched memory ($D000-$FFFF). AUXMOVE uses page zero all during the copy, so it can't handle moves in the memory space switched by ALTZP.

The pairs of bytes you use for passing addresses to this subroutine are called A1, A2, and A4, and they are used for parameter passing by several of the Apple IIe's built-in routines. The addresses of these byte pairs are shown in Table 4-9.

Table 4-9. Parameters for AUXMOVE Routine

Note: The X, Y, and A registers are preserved by AUXMOVE.

Name	Location	Parameter Passed
Carry		1 = Move from main to auxiliary memory 0 = Move from auxiliary to main memory
A1L	$3C	Source starting address, low-order byte
A1H	$3D	Source starting address, high-order byte
A2L	$3E	Source ending address, low-order byte
A2H	$3F	Source ending address, high-order byte
A4L	$42	Destination starting address, low-order byte
A4H	$43	Destination starting address, high-order byte

Put the addresses of the first and last bytes of the block of memory you want to copy into A1 and A2. Put the starting address of the block of memory you want to copy the data to into A4.

The AUXMOVE routine uses the carry bit to select the direction to copy the data. To copy data from main memory to auxiliary memory, set the carry bit; to copy data from auxiliary memory to main memory, clear the carry bit.

When you make the subroutine call to AUXMOVE, the subroutine copies the block of data as specified by the A byte pairs and the carry bit. When it is finished, the accumulator and the X and Y registers are just as they were when you called AUXMOVE.

Transferring Control to Auxiliary Memory

You can use the built-in routine named XFER to transfer control to and from program segments in auxiliary memory. You must set up three parameters before using XFER: the address of the routine you are transferring to, the direction of the transfer (main to auxiliary or auxiliary to main), and which page zero and stack you want to use.

Table 4-10. Parameters for XFER Routine

Note: The X, Y, and A parameters are preserved by XFER.

Name or Location	Parameter Passed
Carry	1 = Transfer from main to auxiliary memory 0 = Transfer from auxiliary to main memory
Overflow	1 = Use page zero and stack in auxiliary memory 0 = Use page zero and stack in main memory
$03ED	Program starting address, low-order byte
$03EE	Program starting address, high-order byte

Put the transfer address into the two bytes at locations $03ED and $03EE, with the low-order byte first, as usual. The direction of the transfer is controlled by the carry bit: set the carry bit to transfer to a program in auxiliary memory; clear the carry bit to transfer to a program in main memory. Use the overflow bit to select which page zero and stack you want to use: clear the overflow bit to use the main memory; set the overflow bit to use the auxiliary memory.

After you have set up the parameters, pass control to the XFER routine by a jump instruction, rather than a subroutine call. XFER saves the accumulator and the transfer address on the current stack, then sets up the soft switches for the parameters you have selected and jumps to the new program.

▲Warning It is the programmer's responsibility to save the current stack pointer at $0100 in main 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.

The Reset Routine

To put the Apple IIe into a known state when it has just been turned on or after a program has malfunctioned, there is a procedure called the reset routine. The reset routine is built into the Apple IIe's firmware, and it is initiated any time you turn power on or press RESET while holding down CONTROL. The reset routine puts the Apple IIe into its normal operating mode and restarts the resident program.

When you initiate a reset, hardware in the Apple IIe sets the memory-controlling soft switches to normal: main board RAM and ROM are enabled, and, if there is an 80-column text card in the auxiliary slot, expansion slot 3 is allocated to the built-in 80-column firmware. Auxiliary RAM is disabled and the bank-switched memory space is set up to read from ROM and write to RAM, using the second bank at $D000.

The reset routine sets the display-controlling soft switches to display 40-column text Page 1 using the primary character set, then sets the window equal to the full 40-column display, puts the cursor at the bottom of the screen, and sets the display format to normal.

For information about the I/O links, see the section "Changing the Standard I/O Links" in Chapter 6.

For more information about peripheral-card ROM, see the section "Peripheral-Card ROM Space" in Chapter 6.

The reset routine sets the keyboard and display as the standard input and output devices by loading the standard I/O links. It turns annunciators 0 and 1 off and annunciators 2 and 3 on, clears the keyboard strobe, turns off any active peripheral-card ROM and outputs a bell (tone).

The Apple IIe has three types of reset: power-on reset, also called cold-start reset; warm-start reset; and forced cold-start reset. The procedure described above is the same for any type of reset. What happens next depends on the reset vector. The reset routine checks the reset vector to determine whether it is valid or not, as described later in this chapter in the section "The Reset Vector." If the reset was caused by turning the power on, the vector will not be valid, and the reset routine will perform the cold-start procedure. If the vector is valid, the routine will perform the warm-start procedure.

The Cold-Start Procedure

If the reset vector is not valid, either the Apple IIe has just been turned on or something has caused memory contents to be changed. The reset routine clears the display and puts the string `Apple //e` (`Apple ][` on an original IIe) at the top of the display. It loads the reset vector and the validity-check byte as described below, then starts checking the expansion slots to see if there is a disk drive controller card in one of them, starting with slot 7 and working down.

If it finds a controller card, it initiates the startup (bootstrap) routine that resides in the controller card's firmware. The startup routine then loads DOS or ProDOS from the disk in drive 1. When the operating system has been loaded, it displays other messages on the screen. If there is no disk in the disk drive, the drive motor just keeps spinning until you press CONTROL-RESET.

For more information about ProDOS and the startup procedure, see the *ProDOS Technical Reference Manual.*

If the reset routine doesn't find a controller card, or if you press CONTROL-RESET again before the startup procedure has been completed, the reset routine will continue without using the disk, and pass control to the built-in Applesoft interpreter.

The Warm-Start Procedure

Whenever you press CONTROL-RESET when the Apple IIe has already completed a cold-start reset, the reset vector is still valid and it is not necessary to reinitialize the entire system. The reset routine simply uses the vector to transfer control to the resident program, which is normally the built-in Applesoft interpreter. If the resident program is indeed Applesoft, your Applesoft program and variables are still intact. If you are using DOS, it is the resident program and it restarts either Applesoft or Integer BASIC, whichever you were using when you pressed CONTROL-RESET.

Important!

A program in bank-switched RAM cannot use the reset vector to regain control after a reset, because the Apple IIe hardware enables ROM in the bank-switched memory space. If you are using Integer BASIC, which is in the bank-switched RAM, you are also using DOS, and it is DOS that controls the reset vector and restarts BASIC.

Forced Cold Start

If a program has loaded the reset vector to point to the beginning of the program, as described in the next section, pressing CONTROL-RESET causes a warm-start reset that uses the vector to transfer control to that program. If you want to stop such a program without turning the power off and on, you can force a cold-start reset by holding down ⍁ and CONTROL, then pressing and releasing RESET.

> *Unconditional Restart:* When you want to stop a program unconditionally—for example, to start up the Apple IIe with some other program—you should use the forced cold-start reset, ⍁-CONTROL-RESET, instead of turning the power off and on.

Whenever you press CONTROL-RESET, firmware in the Apple IIe always checks to see whether either Apple key is down. If the key is down, with or without the ⍁ key, the firmware performs the self-test described later in this chapter. If only the ⍁ key is down, the firmware starts a forced cold-start reset. First, it destroys the program or data in memory by writing two bytes of arbitrary data into each page of main RAM. The two bytes that get written over in page 3 are the ones that contain the reset vector. The reset routine then performs a normal cold-start reset.

The Reset Vector

When you reset the Apple IIe, the reset routine transfers control to the resident program by means of an address stored in page 3 of main RAM. This address is called a vector because it directs program control to a specified destination. There are several other vector addresses stored in page 3, as shown in Table 4-11, including the interrupt vectors described in the section "Interrupts on the Enhanced Apple IIe" in Chapter 6, and the ProDOS and DOS vectors described in the *ProDOS Technical Reference Manual* and the *Apple II DOS Programmer's Manual*.

The cold-start reset routine stores the starting address of the built-in Applesoft interpreter, low-order byte first, in the reset vector address at locations 1010 and 1011 (hexadecimal $03F2 and $03F3). It then stores a validity-check byte, also called the power-up byte, at location 1012 (hexadecimal $03F4). The validity-check byte is computed by performing an exclusive-OR of the second byte of the vector with the constant 165 (hexadecimal $A5). Each time you reset the Apple IIe, the reset routine uses this byte to determine whether the reset vector is still valid.

You can change the reset vector so that the reset routine will transfer control to your program instead of to the Applesoft interpreter. For this to work, you must also change the validity-check byte to the exclusive-OR of the high-order byte of your new reset vector with the constant 165 ($A5). If you fail to do this, then the next time you reset the Apple IIe, the reset routine will determine that the reset vector is invalid and perform a cold-start reset, eventually transferring control to the disk startup routine or to Applesoft.

The reset routine has a subroutine that generates the validity-check byte for the current reset vector. You can use this subroutine by doing a subroutine call to location -1169 (hexadecimal $FB6F). When your program finishes, it can return the Apple IIe to normal operation by restoring the original reset vector and again calling the subroutine to fix up the validity-check byte.

Table 4-11. Page 3 Vectors

Vector Address	Vector Function
$3F0 $3F1	Address of the subroutine that handles BRK requests (normally $59, $FA).
$3F2 $3F3	Reset vector (see text).
$3F4	Power-up byte (see text).
$3F5 $3F6 $3F7	Jump instruction to the subroutine that handles Applesoft & commands (normally $4C, $58, $FF).
$3F8 $3F9 $3FA	Jump instruction to the subroutine that handles user CONTROL-Y commands.
$3FB $3FC $3FD	Jump instruction to the subroutine that handles non-maskable interrupts.
$3FE $3FF	Interrupt vector (address of the subroutine that handles interrupt requests).

See "The User's Interrupt Handler at $3FE" in Chapter 6.

Automatic Self-Test

If you reset the Apple IIe by holding down [ ] and [CONTROL] while pressing and releasing [RESET], the reset routine will start running the built-in self-test. Successfully running this test assures you that the Apple IIe is operational.

▲Warning The self-test routine tests the Apple IIe's programmable memory by writing and then reading it. All programs and data in programmable memory when you run the self-test are destroyed.

The self-test takes several seconds to run. The screen will display some patterns in low resolution mode which will change rapidly just before the self-test finishes. If the test finishes normally, the Apple IIe displays `System OK` and waits for you to restart the system.

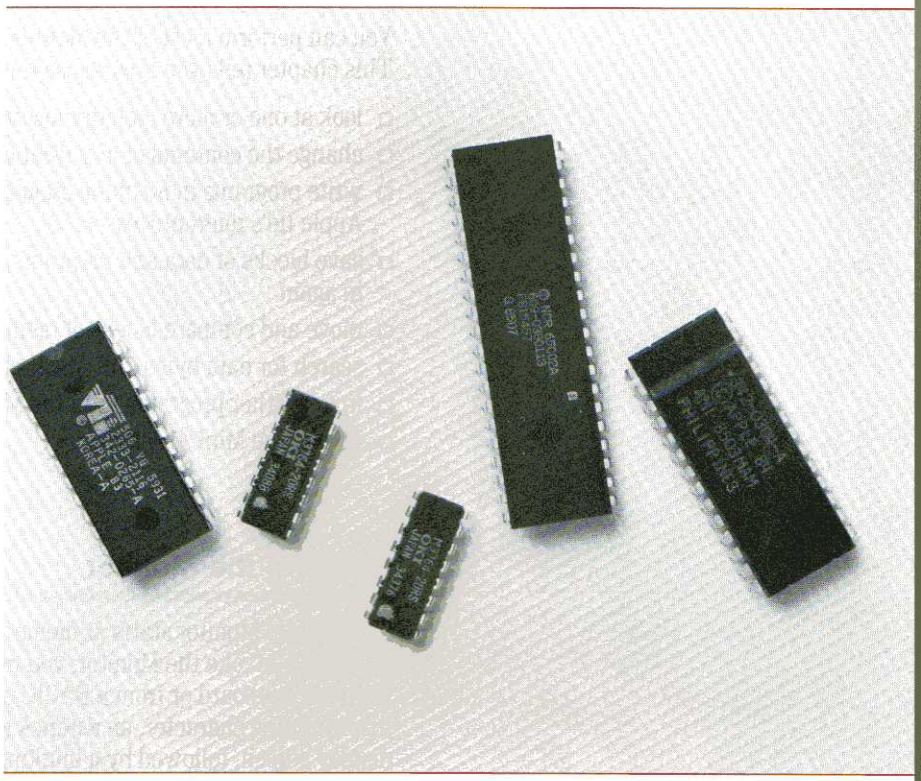
If you have been running a program, some soft switches might be on when you run the self-test. If this happens, the self-test will display a message such as

```
IOU FLAG ES:1
```

Turn the power off for several seconds, then turn it back on and run the self-test again. If it still fails, there is really something wrong; to get it corrected, contact your authorized Apple dealer for service.

Chapter 5

Using the Monitor



The System Monitor is a set of subroutines in the Apple IIe firmware. The Monitor provides a standard interface to the built-in I/O devices described in Chapter 2. The I/O subroutines described in Chapter 3 are part of the System Monitor.

ProDOS, DOS 3.3, and the BASIC interpreters use these subroutines by direct calls to their starting locations, as described for the I/O subroutines in Chapter 3.

The starting addresses for all of the standard subroutines are listed in Appendix B.

If you wish, you can call the standard subroutines from your programs in the same fashion.

You can perform most of the Monitor functions directly from the keyboard. This chapter tells you how to use the Monitor to

- look at one or more memory locations
- change the contents of any location
- write programs in machine language to be executed directly by the Apple IIe's microprocessor
- save blocks of data and programs onto cassette tape and read them back in again
- move and compare blocks of memory
- search for data bytes and ASCII characters in memory
- invoke other programs from the Monitor
- invoke the Mini-Assembler.

Invoking the Monitor

The System Monitor starts at memory location $FF69 (decimal 65385 or -151). To invoke the Monitor, you make a CALL statement to this location from the keyboard or from a BASIC program. When the Monitor is running, its prompting character, an asterisk (*), appears on the left side of the display screen, followed by a blinking cursor.

To use the Monitor, you type commands at the keyboard. When you have finished using the Monitor, you return to the BASIC language you were previously using by pressing CONTROL-RESET, by pressing CONTROL-C then RETURN, or by typing `3D0G`, which executes the resident program—usually Applesoft—whose address is stored in a jump instruction at location $3D0.

Syntax of Monitor Commands

To give a command to the Monitor, you type a line on the keyboard, then press RETURN. The Monitor accepts the line using the standard I/O subroutine GETLN, described in Chapter 3. A Monitor command can be up to 255 characters in length, ending with a carriage return.

A Monitor command can include three kinds of information: addresses, data values, and command characters. You type addresses and data values in hexadecimal notation. Hexadecimal notation uses the ten decimal digits (0-9) and the first six letters (A-F) to represent the sixteen values from 0 to 15. A pair of hexadecimal digits represent values from 0 to 255, corresponding to a byte, and a group of four hexadecimal digits can represent values from 0 to 65,536, corresponding to a word. Any address in the Apple IIe can be represented by four hexadecimal digits.

When the command you type calls for an address, the Monitor accepts any group of hexadecimal digits. If there are fewer than four digits in the group, it adds leading zeros; if there are more than four hexadecimal digits, the Monitor uses only the last four digits. It follows a similar procedure when the command syntax calls for two-digit data values.

See "Summary of Monitor Commands" at the end of this chapter.

Each command you type consists of one command character, usually the first letter of the command name. When the command is a letter, it can be either uppercase or lowercase. The Monitor recognizes 23 different command characters. Some of them are punctuation marks, some are letters, and some are control characters.

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

This chapter contains many examples of the use of Monitor commands. In the examples, the commands and values you type are shown in a normal typeface and the responses of the Monitor are in a computer typeface. Of course, when you perform the examples, all of the characters that appear on the display screen will be in the same typeface. Some of the data values displayed by your Apple IIe may differ from the values printed in these examples, because they are variables stored in programmable memory.

Monitor Memory Commands

When you use the Monitor to examine and change the contents of memory, it keeps track of the address of the last location whose value you inquired about and the address of the location that is next to have its value changed. These are called the last opened location and the next changeable location.

Examining Memory Contents

When you type the address of a memory location and press RETURN, the Monitor responds with the address you typed, a dash, a space, and the value stored at that location, like this:

```
*E000
E000- 20

*33
0033- AA
*
```

Each time the Monitor displays the value stored at a location, it saves the address of that location as the last opened location and as the next changeable location.

Memory Dump

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

```
*20
0020- 00

*.2B
0021- 28 00 18 0F 0C 00 00
0028- A8 06 D0 07

*300
0300- 99

*.315
0301- B9 00 08 0A 0A 0A 99
0308- 00 08 C8 D0 F4 A6 2B A9
0310- 09 85 27 AD CC 03

*.32A
0316- 85 41
0318- 84 40 8A 4A 4A 4A 4A 09
0320- C0 85 3F A9 5D 85 3E 20
0328- 43 03 20
*
```

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

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

You can combine the two commands, opening a location and dumping memory, by simply concatenating them: type the first address, a period, and the second address. This combination of two addresses separated by a period is called a memory range.

```
*300.32F
0300- 99 B9 00 08 0A 0A 0A 99
0308- 00 08 C8 D0 F4 A6 2B A9
0310- 09 85 27 AD CC 03 85 41
0318- 84 40 8A 4A 4A 4A 4A 09
0320- C0 85 3F A9 5D 85 3E 20
0328- 43 03 20 46 03 A5 3D 4D

*30.40
0030- AA 00 FF AA 05 C2 05 C2
0038- 1B FD D0 03 3C 00 40 00
0040- 30

*E015.E025
E016- 4C ED FD
E018- A9 20 C5 24 B0 0C A9 8D
E020- A0 07 20 ED FD A9
*
```

Pressing RETURN by itself causes the Monitor to display one line of a memory dump; that is, a memory dump from the location following the last opened location to the next multiple-of-eight boundary. The Monitor saves the address of the last location displayed as the last opened location and the next changeable location.

```
*5
0005- 00
*RETURN
00 00

*RETURN
0008- 00 00 00 00 00 00 00 00
*32
0032- FF

*RETURN
AA 00 C2 05 C2
*RETURN
0038- 1B FD D0 03 3C 00 3F 00
*
```

Changing Memory Contents

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

▲Warning Use these commands carefully. If you change the zero-page locations used by Applesoft, ProDOS, or DOS, you may lose programs or data stored in memory.

Changing One Byte

The previous commands keep track of the next changeable location; these commands make use of it. In the next example, you open location 0, then type a colon (:) followed by a value.

```
*0

0000- 00
*:5F
```

The contents of the next changeable location have just been changed to the value you typed, as you can see by examining that location:

```
*0

0000- 5F
*
```

You can also combine opening and changing into one operation by typing an address followed by a colon and a value. In the example, you type the address again to verify the change.

```
*302:42

*302

0302- 42
*
```

When you change the contents of a location, the value that was contained in that location disappears, never to be seen again. The new value will remain until you replace it with another value.

Changing Consecutive Locations

You don't have to type a separate command with an address, a colon, a value, and RETURN for each location you want to change. You can change the values of up to 85 consecutive locations at a time (or even more, if you omit leading zeros from the values) by typing only the initial address and colon followed by all the values separated by spaces, and ending with RETURN. The Monitor will duly store the consecutive values in consecutive locations, starting at the location whose address you typed. After it has processed the string of values, it takes the location following the last changed location as the next changeable location. Thus, you can continue changing consecutive locations without typing an address on the next input line by typing another colon and more values. In these examples, you first change some locations, then examine them to verify the changes.

```
*300:69 01 20 ED FD 4C 0 3
*300
0300- 69
*RETURN
01 20 ED FD 4C 00 03
*10:0 1 2 3
*:4 5 6 7
*10.17
0010- 00 01 02 03 04 05 06 07
*
```

ASCII Input Mode

The enhanced Apple IIe has an ASCII input mode that lets you enter ASCII characters just as you can their hexadecimal ASCII equivalents by preceding the literal character with an apostrophe ('). This means that 'A is the same as $C1 and 'B is the same as $C2 to the Monitor. The ASCII value for *any* character following an apostrophe is used by the Monitor.

Each character to be placed in memory should be delimited by a leading apostrophe (') and a trailing space. The only exception to this rule is that the last character in the line is followed with a return character instead of a space. The following example would enter the string "Hooray for sushi!" at $0300 in memory.

```
*300:'H 'o 'o 'r 'a 'y '  'f 'o 'r '  's 'u 's 'h 'i '!
```

Important! ASCII input mode sets the high bit of the code for a character that you enter. So 'A will equal $C1, not $41.

Original IIe The original Apple IIe does not have an ASCII input mode.

Moving Data in Memory

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

{destination} < {start} . {end} M

When you type the actual command, the words in braces should be replaced by hexadecimal addresses, and the braces and spaces should be omitted.

Here are some examples of Monitor commands, including some memory moves. First, you examine the values stored in one range of memory, then store several values in another range of memory; the actual MOVE commands end with the letter M.

```
*0.F

0000- 5F 00 05 07 00 00 00 00
0008- 00 00 00 00 00 00 00 00

*300:A9 8D 20 ED FD A9 45 20 DA FD 4C 00 03

*300.30C

0300- A9 8D 20 ED FD A9 45 20
0308- DA FD 4C 00 03

*0<300.30CM

*0.C

0000- A9 8D 20 ED FD A9 45 20
0008- DA FD 4C 00 03

*310<8.AM

*310.312

0310- DA FD 4C

*2<7.9M

*0.C

0000- A9 8D 20 DA FD A9 45 20
0008- DA FD 4C 00 03
*
```

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

If the destination address of the MOVE command is inside the source range of addresses, then strange (and sometimes wonderful) things happen: the locations between the beginning of the source range and the destination address are treated as a sub-range and the values in this sub-range are replicated throughout the source range.

See the section "Special Tricks With the Monitor" later in this chapter for an interesting application of this feature.

Comparing Data in Memory

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

The VERIFY command, like the MOVE command, needs a range and a destination. The syntax of the VERIFY command is

{destination} < {start} . {end} V

The Monitor compares the values in the source locations with the values in the locations beginning at the destination address. If any values don't match, the Monitor displays the address at which the discrepancy was found and the two values that differ. In the example, you store data values in the range of locations from 0 to $D, copy them to locations starting at $300 with the MOVE command, and then compare them using the VERIFY command. When you use the VERIFY command after you change the value at location 6 to $E4, it detects the change.

```
*0:D7 F2 E9 F4 F4 E5 EE A0 E2 F9 A0 C3 C4 C5

*300<0.DM

*300<0.DV

*6:E4

*300<0.DV

0006-E4 (EE)
*
```

If the VERIFY command finds a discrepancy, it displays the address of the location in the source range whose value differs from its counterpart in the destination range. If there is no discrepancy, VERIFY displays nothing. The VERIFY command leaves the values in both ranges unchanged. The last opened location is the last location in the source range, and the next changeable location is the first location in the source range, just as in the MOVE command. If the ending address of the range is less than the starting address, the values of only the first locations in the ranges will be compared. Like the MOVE command, the VERIFY command also does unusual things if the destination address is within the source range.

See the section "Special Tricks With the Monitor" later in this chapter.

Searching for Bytes in Memory

The SEARCH command lets you search for one or two bytes (either hexadecimal values or ASCII characters) in a range of memory. You must type in the ASCII string (or hexadecimal number or numbers) in reverse of the order that they appear in memory. Think of the SEARCH command as looking for items in a last-in, first-out queue.

The syntax of the SEARCH command is

{value or ASCII}<{start}.{end}S

If the byte (or two byte sequence) that you specify is in the specified memory range, the Monitor will return with a list of the addresses where that byte (or byte sequence) occurs. If the byte (or byte sequence) is not in the range, the Monitor just displays the prompt.

The following example looks for the character string *LO* in memory between $0300 and $03FF.

```
*'O'L<300.3FFS
```

High Bit Set: Remember that ASCII input mode sets the high-order bit of each character that you enter.

The next example searches for the two-byte sequence $FF11.

```
*11FF<300.3FFS
```

You can't search for a two-byte sequence with a high byte of 0. The Monitor ignores the high byte and searches for the low byte only. The sequence 00FF is seen by the Monitor SEARCH command as FF.

Original IIe

The Monitor in the original Apple IIe does not recognize the SEARCH command.

Examining and Changing Registers

The microprocessor's register contents change continuously whenever the Apple IIe is running any sort of program, such as the Monitor. The Monitor lets you see what the register contents were when you invoked the Monitor or a program that you were debugging stopped at a break (BRK). The Monitor also lets you set 65C02 register values before you execute a program with the GO command.

When you call the Monitor, it stores the contents of the microprocessor's registers in memory. The registers are stored in the order A, X, Y, P (processor status register), and S (stack pointer), starting at location $45 (decimal 69). When you give the Monitor a GO command, the Monitor loads the registers from these five locations before it executes the first instruction in your program.

Pressing CONTROL-E and then RETURN invokes the Monitor's EXAMINE command, which displays the stored register values and sets the location containing the contents of the A register as the next changeable location. After using the EXAMINE command, you can change the values in these locations by typing a colon and then typing the new values separated by spaces. In the following example, you display the registers, change the first two, and then display them again to verify the change.

```
*CONTROL-E

A=0A X=FF Y=D8 P=B0 S=F8
*:B0 02

*CONTROL-E

A=B0 X=02 Y=D8 P=B0 S=F8
*
```

Monitor Cassette Tape Commands

The Apple IIe has two jacks for connecting an audio cassette tape recorder. With a recorder connected, you can use the Monitor commands described later in this section to save the contents of a range of memory onto a standard cassette and recall it for later use.

Saving Data on Tape

The Monitor's WRITE command saves the contents of up to 65,536 memory locations on cassette tape. To save a range of memory on tape, give the Monitor the starting and ending addresses of the range, followed by the letter W (for WRITE), like this:

{start} . {end} W

Don't press RETURN yet: first, put the tape recorder in record mode and let the tape run for a second, then press RETURN. The Monitor will write a ten-second tone onto the tape and then write the data. The tone acts as a leader: later, when the Monitor reads the tape, the leader enables the Monitor to get in step with the signal from the tape. When the Monitor is finished writing the range you specified, it will sound a bell (beep) and display a prompt. You should rewind the tape and label it with the memory range that's on the tape and what it's supposed to be.

Here's a small example you can save and use later to try out the READ command. Remember that you must start the cassette recorder in record mode before you press RETURN after typing the WRITE command.

```
*0:FF FF AD 30 C0 88 D0 04 C6 01 F0 08 CA
D0 F6 A6 00 4C 02 00 60

*0.14

0000- FF FF AD 30 C0 88 D0 04
0008- C6 01 F0 08 CA D0 F6 A6
0010- 00 4C 02 00 60

*0.14W

*
```

It takes about 35 seconds total to save the values of 4,096 memory locations preceded by the ten-second leader onto tape. This works out to an average data transfer rate of about 1,350 bits per second.

The WRITE command writes one extra value on the tape after it has written the values in the memory range. This extra value is the checksum, which is the eight-bit partial sum of all values in the range. When the Monitor reads the tape, it uses this value to determine if the data has been written and read correctly. (See the next section.)

Reading Data From Tape

Once you've saved a memory range onto tape with the Monitor's WRITE command, you can read that memory range back into the computer by using the Monitor's READ command. The data values you've stored on the tape need not be read back into the same memory range from whence they came; you can tell the Monitor to put those values into any memory range in the computer's memory, provided that it's the same size as the range you saved.

The format of the READ command is the same as that of the WRITE command, except that the command letter is R:

{start} . {end} R

Once again, after typing the command, don't press RETURN. Instead, start the tape recorder in play mode and wait a few seconds. Although the WRITE command puts a ten-second leader tone on the beginning of the tape, the READ command needs only three seconds of this leader to lock on to the signal from the tape. You should let a few seconds of tape go by before you press RETURN to allow the tape recorder's output to settle down to a steady tone.

This example has two parts. First, you set a range of memory to zero, verify the contents of memory, and then type the READ command, but don't press RETURN.

```
*0:0000000000000000000000
```

```
*0.14
```

```
0000- 00 00 00 00 00 00 00 00
0008- 00 00 00 00 00 00 00 00
0010- 00 00 00 00 00
```

```
*0.14R
```

Now start the cassette running in play mode, wait a few seconds, and press RETURN. After the Monitor sounds the bell (beep) and displays the prompt, examine the range of memory to see that the values from the tape were read correctly:

```
*0.14
```

```
0000- FF FF AD 30 C0 88 D0 04
0008- C6 01 F0 08 CA D0 F6 A6
0010- 00 4C 02 00 60
*
```

After the Monitor has read all the data values on the tape, it reads the checksum value. It computes the checksum on the data it read and compares it to the checksum from the tape. If the two checksums differ, the Monitor sends a beep to the speaker and displays **ERR**. This warns you that there was a problem reading the tape and that the values stored in memory aren't the values that were recorded on the tape. If the two checksums match, the Monitor will just send out a beep and display a prompt.

Miscellaneous Monitor Commands

These Monitor commands enable you to change the video display format from normal to inverse and back, and to assign input and output to accessories in expansion slots.

Inverse and Normal Display

You can control the setting of the inverse-normal mask location used by the COUT subroutine (described in Chapter 3) from the Monitor so that all of the Monitor's output will be in inverse format. The INVERSE command, I, sets the mask such that all subsequent inputs and outputs are displayed in inverse format. To switch the Monitor's output back to normal format, use the NORMAL command, N.

```
*0.F
0000- 0A 0B 0C 0D 0E 0F D0 04
0008- C6 01 F0 08 CA D0 F6 A6
*I
*0.F
0000- 0A 0B 0C 0D 0E 0F D0 04
0008- C6 01 F0 08 CA D0 F6 A6
*N
*0.F
0000- 0A 0B 0C 0D 0E 0F D0 04
0008- C6 01 F0 08 CA D0 F6 A6
*
```

Back to BASIC

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

If you are using DOS 3.3 or ProDOS, press CONTROL-RESET or type

```
3DØG
```

to return to the language you were using, with your program and variables intact.

> ***That's a Number Not a Letter:*** If you use 3D0G, make sure that the third character you type is a zero, not a letter *O*. The letter *G* is the Monitor's GO command, described in the section "Machine-Language Programs" later in this chapter.

Redirecting Input and Output

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

{slot number} CONTROL-P

A PRINTER command to slot number 0 will switch the stream of output characters back to the Apple IIe's video display.

▲Warning

> Don't give the PRINTER command with slot number 0 to deactivate the 80-column firmware, even though you used this command to activate it in slot 3. The command works, but it just disconnects the firmware, leaving some of the soft switches set for 80-column display.

In much the same way that the PRINTER command switches the output stream, the KEYBOARD command substitutes the interface card in a specified expansion slot for the Apple IIe's normal input device, the keyboard. The format for the KEYBOARD command is

{slot number} CONTROL-K

A slot number of 0 for the KEYBOARD command directs the Monitor to accept input from the Apple IIe's built-in keyboard.

The PRINTER and KEYBOARD commands are the exact equivalents of the BASIC commands PR# and IN#.

Hexadecimal Arithmetic

The Monitor will also perform one-byte hexadecimal addition and subtraction. Just type a line in one of these formats:

{value} + {value}
{value} - {value}

The Apple IIe performs the arithmetic and displays the result, as shown in these examples:

```
*20+13
=33
*4A-C
=3E
*FF+4
=03
*3-4
=FF
*
```

Special Tricks With the Monitor

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

Multiple Commands

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

You can freely intermix all of the commands except the STORE (:) command. Since the Monitor takes all values following a colon and places them in consecutive memory locations, the last value in a STORE must be followed by a letter command before another address is encountered. You can use the NORMAL command as the required letter command in such cases; it usually has no effect and can be used anywhere.

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

```
*300.307 300:18 69 1 N 300.302

0300- 00 00 00 00 00 00 00 00
0300- 18 69 01
*
```

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

Filling Memory

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

```
*300:11 22 33

*
```

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

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

This MOVE command will first replicate the pattern at the locations immediately following the original pattern, then replicate that pattern following itself, and so on until it fills the entire range.

```
*303<300.32DM

*300.32F

0300- 11 22 33 11 22 33 11 22
0308- 33 11 22 33 11 22 33 11
0310- 22 33 11 22 33 11 22 33
0318- 11 22 33 11 22 33 11 22
0320- 33 11 22 33 11 22 33 11
0328- 22 33 11 22 33 11 22 33
*
```

You can do a similar trick with the VERIFY command to check whether a pattern repeats itself through memory. This is especially useful to verify that a given range of memory locations all contain the same value. In this example, you first fill the memory range from $0300 to $0320 with zeros and verify it, then change one location and verify again, to see the VERIFY command detect the discrepancy:

```
*300:0
*301<300.31FM
*301<300.31FV
*304:02
*301<300.31FV
0303-00 (02)
0304-02 (00)
*
```

Repeating Commands

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

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

The only way to stop a loop like this is to press CONTROL-RESET; that is how this example ends.

*N 300 302 34:0

```
0300- 11
0302- 33
0300- 11
0302- 33
0300- 11
0302- 33
0300- 11
0302- 33
0300- 11
0302- 33
0300- 11
0302- 33
030
*
```

Creating Your Own Commands

The USER command, CONTROL-Y, forces the Monitor to jump to memory location $03F8. You can put a JMP instruction there that jumps to your own machine-language program. Your program can then examine the Monitor's registers and pointers or the input buffer itself to obtain its data. For example, here is a program that displays everything on the input line after the CONTROL-Y. The program starts at location $0300; the command line that starts with $03F8 stores a jump to $0300 at location $03F8.

*300:A4 34 B9 00 02 20 ED FD C8 C9 8D D0 F5 4C 69 FF

*3F8:4C 00 03

*CONTROL-Y THIS IS A TEST

```
THIS IS A TEST

*
```

Machine-Language Programs

The main reason to program in machine language is to get more speed. A program in machine language can run much faster than the same program written in high-level languages such as BASIC or Pascal, but the machine-language version usually takes a lot longer to write. There are other reasons to use machine language: you might want your program to do something that isn't included in your high-level language, or you might just enjoy the challenge of using machine language to work directly on the bits and bytes.

> *Boning Up on Machine Language:* If you have never used machine language before, you'll need to learn the 65C02 instructions listed in Appendix A. To become proficient at programming in machine language, you'll have to spend some time at it and study at least one of the books on 6502 programming listed in the bibliography. With the books and Appendix A, you'll have the needed information to program the 65C02.

You can get a hexadecimal dump of your program, move it around in memory, or save it on tape and recall it using the commands described in the previous sections. The Monitor commands in this section are intended specifically for you to use in creating, writing, and debugging machine-language programs.

Running a Program

The Monitor command you use to start execution of your machine-language program is the GO command. When you type an address and the letter G, the Apple IIe starts executing machine language instructions starting at the specified location. If you just type the G, execution starts at the last opened location. The Monitor treats this program as a subroutine: it should end with an RTS (return from subroutine) instruction to transfer control back to the Monitor.

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

```
*300:A9 C1 20 ED FD 18 69 1 C9 DB D0 F6 60

*300.30C

0300- A9 C1 20 ED FD 18 69 01
0308- C9 DB D0 F6 60

*300G
ABCDEFGHIJKLMNOPQRSTUVWXYZ
*
```

Disassembled Programs

Machine-language code in hexadecimal isn't the easiest thing in the world to read and understand. To make this job a little easier, machine-language programs are usually written in assembly language and converted into machine-language code by programs called **assemblers**.

Since programs that translate assembly language into machine language are called assemblers, a program like the Monitor's LIST command that translates machine language into assembly language is called a **disassembler**.

The Monitor's LIST command displays machine-language code in assembly-language form. Instead of unformatted hexadecimal gibberish, the LIST command displays each instruction on a separate line, with a three-letter instruction name, or **mnemonic**, and a formatted hexadecimal operand. The LIST command also converts the relative addresses used in branch instructions to absolute addresses.

The word **mnemonic** comes from the same root as *memory* and refers to abbreviations that are easier to remember than the hexadecimal operation codes themselves: for example, for *clear carry* you write CLC instead of $18.

The Monitor LIST command has the format

{location} L

The LIST command starts at the specified location and displays as much memory as it takes to make up a screenfull (20 lines) of instructions, as shown in the following example:

```
*300L
0300-   A9 C1      LDA   #$C1
0302-   20 ED FD   JSR   $FDED
0306-   18         CLC
0306-   69 01      ADC   #$01
0308-   C9 DB      CMP   #$DB
030A-   D0 F6      BNE   $0302
030C-   60        RTS
030D-   00        BRK
030E-   00        BRK
030F-   00        BRK
0310-   00        BRK
0311-   00        BRK
0312-   00        BRK
0313-   00        BRK
0314-   00        BRK
0316-   00        BRK
0316-   00        BRK
0317-   00        BRK
0318-   00        BRK
0319-   00        BRK
*
```

The first seven lines of this example are the assembly-language form of the program you typed in the previous example. The rest of the lines are BRK instructions only if this part of memory has zeros in it: other values will be disassembled as other instructions.

The Monitor saves the address that you specify in the LIST command, but not as the last opened location used by the other commands. Instead, the Monitor saves this address as the program counter, which it uses only to point to locations within programs. Whenever the Monitor performs a LIST command, it sets the program counter to point to the location immediately following the last location displayed on the screen, so that if you type another LIST command it will display another screenful of instructions, starting where the previous display left off.

The Mini-Assembler

Without an assembler, you have to write your machine language program, take the hexadecimal values for the opcodes and operands, and store them in memory using the commands covered in the previous sections. That is exactly what you did when you ran the previous examples.

The Monitor includes an assembler called the Mini-Assembler that lets you enter machine-language programs directly from the keyboard of your Apple. ASCII characters can be entered in Mini-Assembler programs, exactly as you enter them in the Monitor. Note that the Mini-Assembler doesn't accept labels; you must use actual values and addresses.

Starting the Mini-Assembler

To start the Mini-Assembler first invoke the Monitor by typing `CALL-151` RETURN, and then from the Monitor, type `!` followed by RETURN. The Monitor prompt character then changes from `*` to `!`.

When you finish using the Mini-Assembler, press RETURN from a blank line to return to the Monitor.

Restrictions

The Mini-Assembler supports only the subset of 65C02 instructions that are found on the 6502.

Original IIe

Before you can use the Mini-Assembler on the original Apple IIe, you have to be running Integer BASIC. When you start up the computer using DOS or either BASIC, the Apple IIe loads the Integer BASIC interpreter from the file named INTBASIC into the bank-switched RAM. Here's how to start the Mini-Assembler on an original Apple IIe:

1. Start Integer BASIC from DOS 3.3 by typing `INT` RETURN.
2. After the Integer prompt character (>) and a cursor appear, enter the Monitor by typing `CALL -151` RETURN.
3. Now start the Mini-Assembler by typing `F666G` RETURN.

Using the Mini-Assembler

The Mini-Assembler saves one address, that of the program counter. Before you start to type a program, you must set the program counter to point to the location where you want the Mini-Assembler to store your program. Do this by typing the address followed by a colon.

After the colon, type the mnemonic for the first instruction in your program, followed by a space and the operand of the instruction. Now press RETURN. The Mini-Assembler converts the line you typed into hexadecimal, stores it in memory beginning at the location of the program counter, and then disassembles it again and displays the disassembled line. It then displays a prompt on the next line.

Now the Mini-Assembler is ready to accept the second instruction in your program. To tell it that you want the next instruction to follow the first, don't type an address or a colon: just type a space and the next instruction's mnemonic and operand, then press RETURN. The Mini-Assembler assembles that line and waits for another.

Formats for operands are listed in Table 5-1.

```
!300:LDX #02

0300-   A2 02     LDX   #$02
! LDA $0,X

0302-   B5 00     LDA   $00,X
! STA $10,X

0304    95 10     STA   $10,X
! DEX

0306-   CA        DEX
! STA $C030

0307-   8D 30 C0  STA   $C030
! BPL $302

030A-   10 F6     BPL   $0302
! BRK

030C-   00        BRK
!
```

If the line you type has an error in it, the Mini-Assembler beeps loudly and displays a caret (^) under or near the offending character in the input line. Most common errors are the result of typographical mistakes: misspelled mnemonics, missing parentheses, and so forth. The Mini-Assembler also rejects the input line if you forget the space before or after a mnemonic or

include an extraneous character in a hexadecimal value or address. If the destination address of a branch instruction is out of the range of the branch (more than 127 locations distant from the address of the instruction), the Mini-Assembler flags this as an error.

There are several different ways to leave the Mini-Assembler and reenter the Monitor. On an enhanced Apple IIe only, simply press RETURN at a blank line.

Original IIe | On an original Apple IIe, type the Monitor command $FF69G.

On any Apple IIe, you can press CONTROL-RESET, which warm starts BASIC, then type

```
CALL -151
```

Your assembly-language program is now stored in memory. You can display it with the LIST command:

```
*300l
0300-  A2 02     LDX   #$02
0302-  B5 00     LDA   $00,X
0304-  95 10     STA   $10,X
0306-  CA        DEX
0307-  8D 30 C0  STA   $C030
030A-  10 F6     BPL   $0302
030C-  00        BRK
030D-  00        BRK
030E-  00        BRK
030F-  00        BRK
0310-  00        BRK
0311-  00        BRK
0312-  00        BRK
0313-  00        BRK
0314-  00        BRK
0316-  00        BRK
0316-  00        BRK
0317-  00        BRK
0318-  00        BRK
0319-  00        BRK
*
```

Mini-Assembler Instruction Formats

See Appendix A for more information about 65C02 (and 6502) instructions.

The Apple Mini-Assembler recognizes 56 mnemonics and 13 addressing formats. These constitute the 6502 subset of the 65C02 instruction set. The mnemonics are standard, as used in the *Synertek Programming Manual* (Apple part number A2L0003), but the addressing formats are somewhat different. Table 5-1 shows the Apple standard address-mode formats for 6502 assembly language.

Table 5-1. Mini-Assembler Address Formats

Addressing Mode	Format
Accumulator	*
Implied	*
Immediate	#${value}
Absolute	${address}
Zero page	${address}
Indexed zero page	${address},X ${address},Y
Indexed absolute	${address},X ${address},Y
Relative	${address}
Indexed indirect	(${address},X)
Indirect indexed	(${address}),Y
Absolute indirect	(${address})

* These instructions have no operands.

An address consists of one or more hexadecimal digits. The Mini-Assembler interprets addresses the same way the Monitor does: if an address has fewer than four digits, the Mini-Assembler adds leading zeros; if the address has more than four digits, then it uses only the last four.

> *Dollar Signs:* In this manual, dollar signs ($) in addresses signify that the addresses are in hexadecimal notation. They are ignored by the Mini-Assembler and may be omitted when typing programs.

There is no syntactical distinction between the absolute and zero-page addressing modes. If you give an instruction to the Mini-Assembler that can be used in both absolute and zero-page mode, the Mini-Assembler assembles that instruction in absolute mode if the operand for that instruction is greater than $FF, and it assembles it in zero-page mode if the operand is less than $0100.

Instructions in accumulator mode and implied addressing mode need no operands.

Branch instructions, which use the relative addressing mode, require the target address of the branch. The Mini-Assembler calculates the relative distance to use in the instruction automatically. If the target address is more than 127 locations distant from the instruction, the Mini-Assembler sounds a bell (beep), displays a caret (^) under the target address, and does not assemble the line.

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

Summary of Monitor Commands

Here is a summary of the Monitor commands, showing the syntax for each one.

Examining Memory

{adrs}	Examines the value contained in one location.
{adrs1}.{adrs2}	Displays the values contained in all locations between {adrs1} and {adrs2}.
RETURN	Displays the values in up to eight locations following the last opened location.

Changing the Contents of Memory

{adrs}:{val} {val}...	Stores the values in consecutive memory locations starting at {adrs}.
:{val}{val}...	Stores values in memory starting at the next changeable location.

Moving and Comparing

{dest}<{start}.{end}M	Copies the values in the range {start}.{end} into the range beginning at {dest}.
{dest}<{start}.{end}V	Compares the values in the range {start}.{end} to those in the range beginning at {dest}.

The Examine Command

CONTROL-E	Displays the locations where the contents of the 65C02's registers are stored and opens them for changing.

The Search Command

{val}<{start}.{end}S	Displays the address of the first occurrence of {val} in the specified range beginning at {start}.

Cassette Tape Commands

{start}.{end}W	Writes the values in the memory range {start}.{end} onto tape, preceded by a ten-second leader.
{start}.{end}R	Reads values from tape, storing them in memory beginning at {start} and stopping at {end}. Prints ERR if an error occurs.

Miscellaneous Monitor Commands

I	Sets inverse display mode.
N	Sets normal display mode.
CONTROL B	Enters the language currently active (usually Applesoft).
CONTROL C	Returns to the language currently active (usually Applesoft).
{val}+{val}	Adds the two values and prints the hexadecimal result.
{val}-{val}	Subtracts the second value from the first and prints the result.
{slot} CONTROL P	Diverts output to the device whose interface card is in slot number {slot}. If {slot}=0, accepts input from the keyboard.
CONTROL Y	Jumps to the machine-language subroutine at location $3F8.

Running and Listing Programs

{adrs}G	Transfers control to the machine language program beginning at {adrs}.
{adrs}L	Disassembles and displays 20 instructions, starting at {adrs}. Subsequent LIST commands display 20 more instructions.

The Mini-Assembler

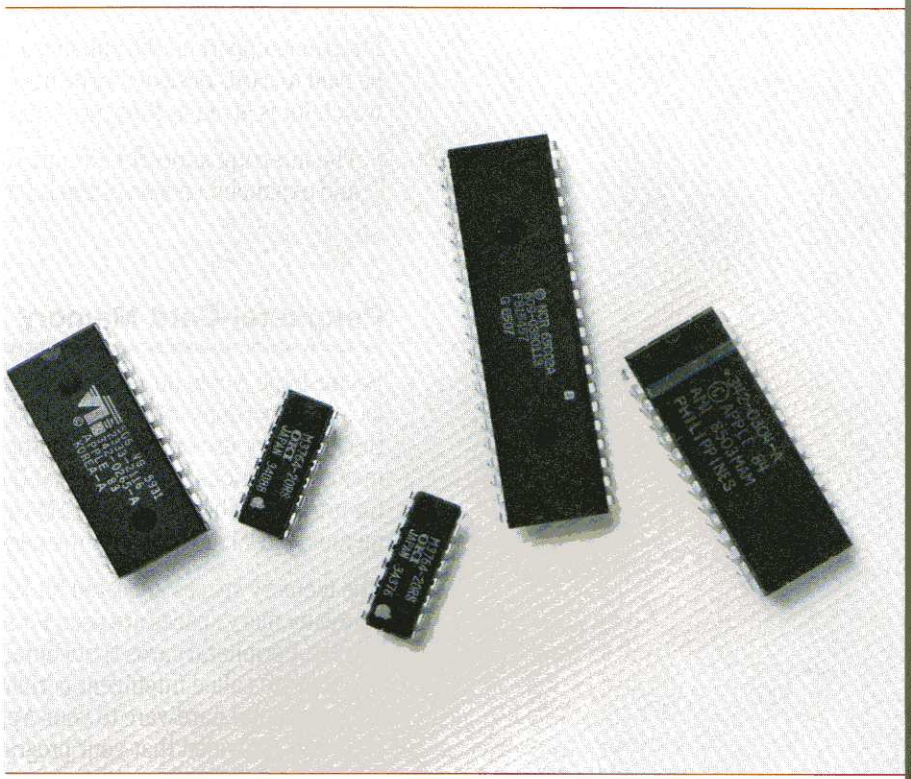
Original IIe

The Mini-Assembler is available on an original Apple IIe only when Integer BASIC is active. See the earlier section "The Mini-Assembler."

F666G	Invokes the Mini-Assembler on the original Apple IIe.
!	Invokes the Mini-Assembler on the enhanced Apple IIe.
$|command|	Executes a Monitor command from the Mini-Assembler on the original Apple IIe.
$FF69G	Leaves the Mini-Assembler on the original Apple IIe.
RETURN	Leaves the Mini-Assembler on the enhanced Apple IIe.

Chapter 6

Programming for Peripheral Cards



The seven expansion slots on the Apple IIe's main circuit board are used for installing circuit cards containing the hardware and firmware needed to interface peripheral devices to the Apple IIe. These slots are not simple I/O ports; peripheral cards can access the Apple IIe's data, address, and control lines via these slots. The expansion slots are numbered from 1 to 7, and certain signals, described below, are used to select a specific slot.

II Plus, II

The Apple II and Apple II Plus have an eighth expansion slot: slot number 0. On those models, slot 0 is normally used for a language card or a ROM card; the functions of the Apple II Language Card are built into the main circuit board of the Apple IIe.

Interrupt support on the enhanced Apple IIe requires that special attention be paid to cards designed to be in slot 3. A description of what you need to watch for is given at the end of this chapter.

Original IIe

The interrupt support built into the enhanced Apple IIe is an enhanced and expanded version of the interrupt support in the original Apple IIe.

Peripheral-Card Memory Spaces

Because the Apple IIe's microprocessor does all of its I/O through memory locations, portions of the Apple IIe's memory space have been allocated for the exclusive use of the cards in the expansion slots. In addition to the memory locations used for actual I/O, there are memory spaces available for programmable memory (RAM) in the main memory and for read-only memory (ROM or PROM) on the peripheral cards themselves.

The memory spaces allocated for the peripheral cards are described below. Those memory spaces are used for small dedicated programs such as I/O drivers. Peripheral cards that contain their own driver routines in firmware like this are called intelligent peripherals. They make it possible for you to add peripheral hardware to your Apple IIe without having to change your programs, provided that your programs follow normal practice for data input and output.

Peripheral-Card I/O Space

Each expansion slot has the exclusive use of sixteen memory locations for data input and output in the memory space beginning at location $C090. Slot 1 uses locations $C090 through $C09F, slot 2 uses locations $C0A0 through $C0AF, and so on through location $C0FF, as shown in Table 6-1.

Signals for which the active state is low are marked with a prime (′).

These memory locations are used for different I/O functions, depending on the design of each peripheral card. Whenever the Apple IIe addresses one of the sixteen I/O locations allocated to a particular slot, the signal on pin 41 of that slot, called DEVICE SELECT′, switches to the active (low) state. This signal can be used to enable logic on the peripheral card that uses the four low-order address lines to determine which of its sixteen I/O locations is being accessed.

Table 6-1. Peripheral-Card I/O Memory Locations Enabled by DEVICE SELECT′

Slot	Locations	Slot	Locations
1	$C090-$C09F	5	$C0D0-$C0DF
2	$C0A0-$C0AF	6	$C0E0-$C0EF
3	$C0B0-$C0BF	7	$C0F0-$C0FF
4	$C0C0-$C0CF		

Peripheral-Card ROM Space

One 256-byte page of memory space is allocated to each accessory card. This space is normally used for read-only memory (ROM or PROM) on the card with driver programs that control the operation of the peripheral device connected to the card.

The page of memory allocated to each expansion slot begins at location $Cn00, where n is the slot number, as shown in Table 6-2 and Figure 6-3. Whenever the Apple IIe addresses one of the 256 ROM memory locations allocated to a particular slot, the signal on pin 1 of that slot, called I/O SELECT′, switches to the active (low) state. This signal enables the ROM or PROM devices on the card, and the eight low-order address lines determine which of the 256 memory locations is being accessed.

Table 6-2. Peripheral-Card ROM Memory Locations Enabled by I/O SELECT′

Slot	Locations	Slot	Location
1	$C100-$C1FF	5	$C500-$C5FF
2	$C200-$C2FF	6	$C600-$C6FF
3	$C300-$C3FF	7	$C700-$C7FF
4	$C400-$C4FF		

Expansion ROM Space

In addition to the small areas of ROM memory allocated to each expansion slot, peripheral cards can use the 2K-byte memory space from $C800 to $CFFF for larger programs in ROM or PROM. This memory space is called expansion ROM space. (See the memory map in Figure 6-3). Besides being larger, the expansion ROM memory space is always at the same locations regardless of which slot is occupied by the card, making programs that occupy this memory space easier to write.

See the section "I/O Programming Suggestions" later in this chapter.

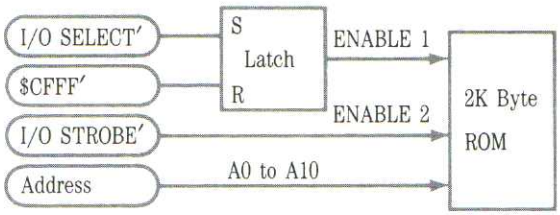
This memory space is available to any peripheral card that needs it. More than one peripheral card can have expansion ROM on it, but only one of them can be active at a time.

Each peripheral card that uses expansion ROM must have a circuit on it to enable the ROM. The circuit does this by a two-stage process: first, it sets a flip-flop when the I/O SELECT′ signal, pin 1 on the slot, becomes active (low); second, it enables the expansion ROM devices when the I/O STROBE′ signal, pin 20 on the slot, becomes active (low). Figure 6-1 shows a typical ROM-enable circuit.

The I/O SELECT′ signal on a particular slot becomes active whenever the Apple IIe's microprocessor addresses a location in the 256-byte ROM address space allocated to that slot. The I/O STROBE′ signal on all of the expansion slots becomes active (low) when the microprocessor addresses a location in the expansion-ROM memory space, $C800-$CFFF. The I/O STROBE′ signal is used to enable the expansion-ROM devices on a peripheral card. (See Figure 6-1.)

Important! If there is an 80-column text card installed in the auxiliary slot, some of the functions normally associated with slot 3 are performed by the 80-column text card and the built-in 80-column firmware. With the 80-column text card installed, the I/O STROBE′ signal is not available on slot 3, so firmware in expansion ROM on a card in slot 3 will not run.

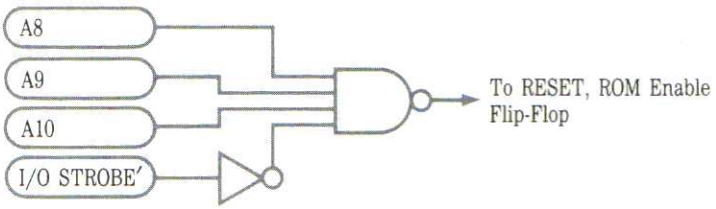
Figure 6-1. Expansion ROM Enable Circuit



A program on a peripheral card can get exclusive use of the expansion ROM memory space by referring to location $CFFF in its initialization phase. This location is special: all peripheral cards that use expansion ROM must recognize a reference to $CFFF as a signal to reset their ROM-enable flip-flops and disable their expansion ROMs. Of course, doing so also disables the expansion ROM on the card that is about to use it, but the next instruction in the initialization code sets the flip-flop in the expansion-ROM enable circuit on the card.

A card that needs to use the expansion ROM space must first insert its slot address ($Cn) in $07F8 before it refers to $CFFF. This allows interrupting devices to reenable the card's expansion ROM after interrupt handling is finished. Once its slot address has been inserted in $07F8, the peripheral card has exclusive use of the expansion memory space and its program can jump directly into the expansion ROM.

Figure 6-2. ROM Disable Address Decoding



As described earlier, the expansion-ROM disable circuit resets the enable flip-flop whenever the 65C02 addresses location $CFFF. To do this, the peripheral card must detect the presence of $CFFF on the address bus. You can use the I/O STROBE′ signal for part of the address decoding, since it is active for addresses from $C800 through $CFFF. If you can afford to sacrifice some ROM space, you can simplify the address decoding even further and save circuitry on the card. For example, if you give up the last 256 bytes of expansion ROM space, your disable circuit only needs to detect addresses of the form $CFxx, and you can use the minimal disable-decoding circuitry shown in Figure 6-2.

Important!

Applesoft addresses two locations in the $CFxx space, thereby resetting the enable flip-flop. If your peripheral device is going to be used with Applesoft programs, you must either use the full address decoding or else enable the expansion ROM each time it is needed.

Peripheral-Card RAM Space

There are 56 bytes of main memory allocated to the peripheral cards, eight bytes per card, as shown in Table 6-3. These 56 locations are actually in the RAM memory reserved for the text and low-resolution graphics displays, but these particular locations are not displayed on the screen and their contents are not changed by the built-in output routine COUT1. Programs in ROM on peripheral cards use these locations for temporary data storage.

Table 6-3. Peripheral-Card RAM Memory Locations

Base Address	Slot Number 1	2	3*	4	5	6	7
$0478	$0479	$047A	$047B*	$047C	$047D	$047E	$047F
$04F8	$04F9	$04FA	$04FB*	$04FC	$04FD	$04FE	$04FF
$0578	$0579	$057A	$057B*	$057C	$057D	$057E	$057F
$05F8	$05F9	$05FA	$05FB*	$05FC	$05FD	$05FE	$05FF
$0678	$0679	$067A	$067B*	$067C	$067D	$067E	$067F
$06F8	$06F9	$06FA	$06FB*	$06FC	$06FD	$06FE	$06FF
$0778	$0779	$077A	$077B*	$077C	$077D	$077E	$077F
$07F8	$07F9	$07FA	$07FB*	$07FC	$07FD	$07FE	$07FF

* If there is a card in the auxiliary slot, it takes over these locations.

A program on a peripheral card can use the eight base addresses shown in the table to access the eight RAM locations allocated for its use, as shown in the next section, "I/O Programming Suggestions."

▲Warning

The Apple IIe firmware sets the value of $04FB to $FF on a reset, even if there is no 80-column card installed.

I/O Programming Suggestions

A program in ROM on a peripheral card should work no matter which slot the card occupies. If the program includes a jump to an absolute location in one of the 256-byte memory spaces, then the card will work only when it is plugged into the slot that uses that memory space. If you are writing the program for a peripheral card that will be used by many people, you should avoid placing such a restriction on the use of the card.

Important!

To function properly no matter which slot a peripheral card is installed in, the program in the card's 256-byte memory space must not make any absolute references to itself. Instead of using jump instructions, you should force conditions on branch instructions, which use relative addressing.

The first thing a peripheral-card used as an I/O device must do when called is to save the contents of the Apple IIe's microprocessor's registers. (Peripheral cards not being used as I/O devices do not need to save the registers.) The device should save the register's contents on the stack, and restore them just before returning control to the calling program. If there is RAM on the peripheral card, the information may be stored there.

Most single-character I/O is done via the microprocessor's accumulator. A character being output through your subroutine will be in the accumulator with its high bit set when your subroutine is called. Likewise, if your subroutine is performing character input, it must leave the character in the accumulator with its high bit set when it returns to the calling program.

Finding the Slot Number With ROM Switched In

The memory addresses used by a program on a peripheral card differ depending on which expansion slot the card is installed in. Before it can refer to any of those addresses, the program must somehow determine the correct slot number. One way to do this is to execute a JSR (jump to subroutine) to a location with an RTS (return from subroutine) instruction in it, and then derive the slot number from the return address saved on the stack, as shown in the following example.

```
PHP              ; save status
SEI              ; inhibit interrupts
JSR KNOWNRTS     ; -> a known RTS instruction...
                 ; ...that you set up
TSX              ; get high byte of the...
LDA $0100,X      ; ...return address from stack
AND #$0F         ; low-order digit is slot no.
PLP              ; restore status
```

The slot number can now be used in addressing the memory allocated to the peripheral card, as shown in the next section.

I/O Addressing

Once your peripheral-card program has the slot number, the card can use the number to address the I/O locations allocated to the slot. Table 6-4 shows how these locations are related to sixteen base addresses starting with $C080. Notice that the difference between the base address and the desired I/O location has the form $n0, where n is the slot number. Starting with the slot number in the accumulator, the following example computes this difference by four left shifts, then loads it into an index register and uses the base address to specify one of sixteen I/O locations.

```
ASL              ; get n into...
ASL              ;
ASL              ;
ASL              ; ...high-order nybble...
TAX              ; ... of index register.
LDA     $C080,X  ; load from first I/O location
```

See the section "Setting Bank Switches" in Chapter 4 for more information.

Selecting Your Target: You must make sure that you get an appropriate value into the index register when you address I/O locations this way. For example, starting with 1 in the accumulator, the instructions in the above example perform an LDA from location $C090, the first I/O location allocated to slot 1. If the value in the accumulator had been 0, the LDA would have accessed location $C080, thereby setting the soft switch that selects the second bank of RAM at location $D000 and enables it for reading.

Table 6-4. Peripheral-Card I/O Base Addresses

Base Address	Connector Number 1	2	3	4	5	6	7
$C080	$C090	$C0A0	$C0B0	$C0C0	$C0D0	$C0E0	$C0F0
$C081	$C091	$C0A1	$C0B1	$C0C1	$C0D1	$C0E1	$C0F1
$C082	$C092	$C0A2	$C0B2	$C0C2	$C0D2	$C0E2	$C0F2
$C083	$C093	$C0A3	$C0B3	$C0C3	$C0D3	$C0E3	$C0F3
$C084	$C094	$C0A4	$C0B4	$C0C4	$C0D4	$C0E4	$C0F4
$C085	$C095	$C0A5	$C0B5	$C0C5	$C0D5	$C0E5	$C0F5
$C086	$C096	$C0A6	$C0B6	$C0C6	$C0D6	$C0E6	$C0F6
$C087	$C097	$C0A7	$C0B7	$C0C7	$C0D7	$C0E7	$C0F7
$C088	$C098	$C0A8	$C0B8	$C0C8	$C0D8	$C0E8	$C0F8
$C089	$C099	$C0A9	$C0B9	$C0C9	$C0D9	$C0E9	$C0F9
$C08A	$C09A	$C0AA	$C0BA	$C0CA	$C0DA	$C0EA	$C0FA
$C08B	$C09B	$C0AB	$C0BB	$C0CB	$C0DB	$C0EB	$C0FB
$C08C	$C09C	$C0AC	$C0BC	$C0CC	$C0DC	$C0EC	$C0FC
$C08D	$C09D	$C0AD	$C0BD	$C0CD	$C0DD	$C0ED	$C0FD
$C08E	$C09E	$C0AE	$C0BE	$C0CE	$C0DE	$C0EE	$C0FE
$C08F	$C09F	$C0AF	$C0BF	$C0CF	$C0DF	$C0EF	$C0FF

RAM Addressing

A program on a peripheral card can use the eight base addresses shown in Table 6-3 to access the eight RAM locations allocated for its use. The program does this by putting its slot number into the Y index register and using indexed addressing mode with the base addresses. The base addresses can be defined as constants because they are the same no matter which slot the peripheral card occupies.

If you start with the correct slot number in the accumulator (by using the example shown earlier), then the following example uses all eight RAM locations allocated to the slot.

```
TAY
LDA     $0478,Y
STA     $04F8,Y
LDA     $0578,Y
STA     $05F8,Y
LDA     $0678,Y
STA     $06F8,Y
LDA     $0778,Y
STA     $07F8,Y
```

▲Warning

You must be very careful when you have your peripheral-card program store data at the base-address locations themselves since they are temporary storage locations; the RAM at those locations is used by the disk operating system. Always store the first byte of the ROM location of the expansion slot that is currently active ($Cn) in location $7F8, and the first byte of the ROM location of the slot holding the controller card for the startup disk drive in location $5F8.

Changing the Standard I/O Links

There are two pairs of locations in the Apple IIe that are used for controlling character input and output. They are called the I/O links. In a Apple IIe running without a disk operating system, the I/O links normally contain the starting addresses of the standard input and output routines—KEYIN and COUT1 if the 80-column firmware is not active, BASICIN and BASICOUT if the 80-column is active. If a disk operating system is running, one or both of the links will hold the addresses of the operating system input and output routines.

See "The Standard I/O Links" in Chapter 3.

The link at locations $36 and $37 (decimal 54 and 55) is called CSW, for *character output switch*. Individually, location $36 is called CSWL (CSW Low) and location $37 is called CSWH (CSW High). CSW holds the starting address of the subroutine the Apple IIe is currently using for single-character output. This address is normally $FDF0, the address of routine COUT1, or $C307, the address of BASICOUT.

COUT1 and BASICOUT are described in Chapter 3.

When you issue a PR#n from BASIC or an n CONTROL-P from the Monitor, the Apple IIe changes this link address to the first address in the ROM memory space allocated to slot number n. That address has the form $Cn00. Subsequent calls for character output are thus transferred to the program on the peripheral card. That program can use the instruction sequences given above to find its slot number and use the I/O and RAM locations allocated to it. When it is finished, the program can execute an RTS (return from subroutine) instruction to return control to the calling program, or jump to the output routine COUT1 at location $FDF0 to display the output character (which must be in the accumulator) on the screen, then let COUT1 return to the calling program.

A similar link at locations $38 and $39 (decimal 56 and 57) is called KSW, for *keyboard input switch*. Individually, location $38 is called KSWL (for KSW low) and location $39 is called KSWH (KSW high). KSW holds the starting address of the routine currently being used for single-character input. This address is normally $FD1B, the starting address of KEYIN, or $C305, the address of BASICIN.

KEYIN and BASICIN are described in Chapter 3.

When you issue an IN#n command from BASIC or an n CONTROL-K from the Monitor, the Apple IIe changes this link address to $Cn00, the beginning of the ROM memory space that is allocated to slot number n. Subsequent calls for character input are thus transferred to the program on the accessory card. That program can use the instruction sequences given above to find its slot number and use the I/O and RAM locations allocated to it. The program should put the input character, with its high bit set, into the accumulator and execute an RTS instruction to return control to the program that requested input.

When a disk operating system (ProDOS or DOS 3.3) is running, one or both of the standard I/O links hold addresses of the operating system's input and output routines. The operating system has internal locations that hold the addresses of the character input and output routines that are currently active.

Important!

If a program that is running with ProDOS or DOS 3.3 changes the standard link addresses, either directly or via IN# and PR# commands, the operating system is disconnected.

See the *ProDOS Technical Reference Manual* for more about using link addresses.

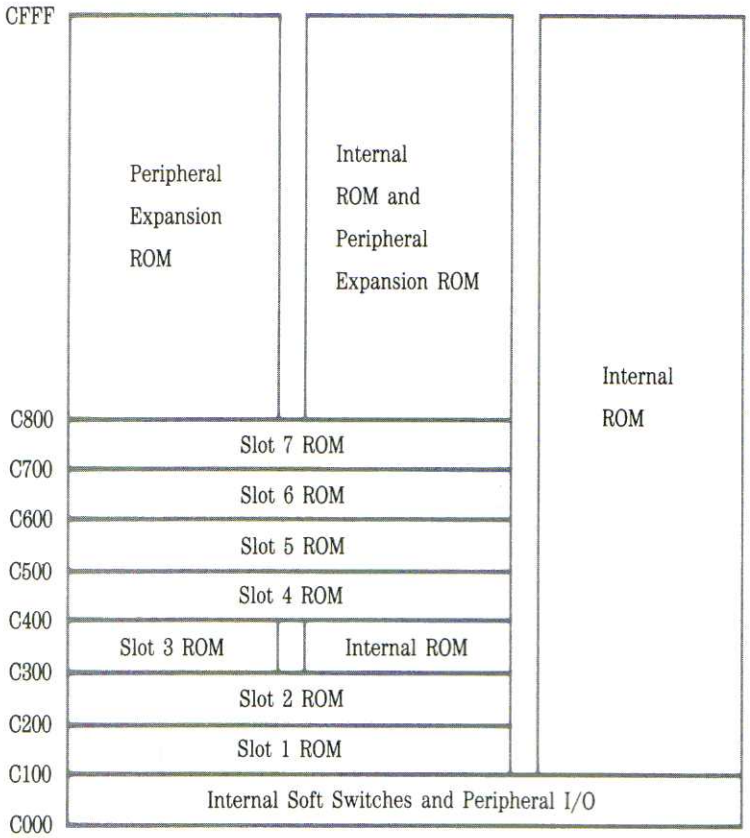
To avoid disconnecting the operating system each time a BASIC program initiates I/O to a slot, it should use either an IN# or a PR# command from inside a PRINT statement that starts with a Control-D character. For assembly-language programs, there is a DOS 3.3 subroutine call to use when changing the link addresses. After changing CSW or KSW, the program calls this subroutine at location $03EA (decimal 1002). The subroutine transfers the link address to a location inside the operating system and then restores the operating system address in the standard link location.

Refer to the section on input and output link registers in the *DOS Programmer's Manual* and the *ProDOS Technical Reference Manual* for further details.

Other Uses of I/O Memory Space

The portion of memory space from location $C000 through $CFFF (decimal 49152 through 53247) is normally allocated to I/O and program memory on the peripheral cards, but there are two other functions that also use this memory space: the built-in self-test firmware and the 80-column display firmware. The soft switches that control the allocation of this memory space are described in the next section.

Figure 6-3. I/O Memory Map



Switching I/O Memory

The built-in firmware uses two soft switches to control the allocation of the I/O memory space from $C000 to $CFFF. The locations of these soft switches, SLOTCXROM and SLOTC3ROM, are given in Table 6-5.

> *Note:* Like the display switches described in Chapter 2, these soft switches share their locations with the keyboard data and strobe functions. The switches are activated only by writing, and the states can be determined only by reading, as indicated in Table 6-5.

Table 6-5. I/O Memory Switches

Name	Function	Location Hex	Location Decimal		Notes
SLOTC3ROM	Slot ROM at $C300	$C00B	49163	-16373	Write
	Internal ROM at $C300	$C00A	49162	-16374	Write
	Read SLOTC3ROM switch	$C017	49175	-16361	Read
SLOTCXROM	Slot ROM at $Cx00	$C006	49159	-16377	Write
	Internal ROM at $Cx00	$C007	49158	-16378	Write
	Read SLOTCXROM switch	$C015	49173	-16363	Read

When SLOTC3ROM is on, the 256-byte ROM area at $C300 is available to a peripheral card in slot 3, which is the slot normally used for a terminal interface. If a card is installed in the auxiliary slot when you turn on the power or reset the Apple IIe, the SLOT3ROM switch is turned off. Turning SLOTC3ROM off disables peripheral-card ROM in slot 3 and enables the built-in 80-column firmware, as shown in Figure 6-3. The 80-column firmware is assigned to slot-3 address space because slot 3 is normally used with a terminal interface, so the built-in firmware will work with programs that use slot 3 this way.

The bus and I/O signals are always available to a peripheral card in slot 3, even when the 80-column hardware and firmware are operating. Thus it is always possible to use this slot for any I/O peripheral that does *not* have built-in firmware.

When SLOTCXROM is active (high), the I/O memory space from $C100 to $C7FF is allocated to the expansion slots, as described previously. Setting SLOTCXROM inactive (low) disables the peripheral-card ROM and selects built-in ROM in all of the I/O memory space except the part from $C000 to $C0FF (used for soft switches and data I/O), as shown in Figure 6-3. In addition to the 80-column firmware at $C300 and $C800, the built-in ROM includes firmware that performs the self-test of the Apple IIe's hardware.

> *Note:* Setting SLOTCXROM low enables built-in ROM in all of the I/O memory space (except the soft-switch area), including the $C300 space, which contains the 80-column firmware.

Developing Cards for Slot 3

Original IIe

In the original Apple IIe firmware, the internal slot 3 firmware was always switched in if there was an 80-column card (either 1K or 64K) in the auxiliary slot. This means that peripheral cards with their own ROM were effectively switched out of slot 3 when the system was turned on.

With the enhanced Apple IIe Monitor ROM, the rules are different. A peripheral card in slot 3 is now switched in when the system is started up or when RESET is pressed *if* the card's ROM has the following ID bytes:

$C305 = $38
$C307 = $18

The enhanced Apple IIe firmware requires that interrupt code be present in the $C3 page (either external or internal). A peripheral card in slot 3 must have the following code to support interrupts. After this segment, the code continues execution in the internal ROM at $C400.

```
$C3F4:   IRQDONE    STA $C081     ;Read ROM, write RAM
                    JMP $FC7A     ;Jump to $F8 ROM

                    IRQ
                    BIT $C015     ;slot or internal ROM
                    STA $C007     ;force in internal ROM
```

When programming for cards in slot 3:

- You must support the AUXMOVE and XFER routines at $C312 and $C314.
- Don't use unpublished entry points into the internal $Cn00 firmware, because there is no guarantee that they will stay the same.
- If your peripheral card is a character I/O device, you must follow the Pascal 1.1 firmware protocol, described in the next section.

For more information about the $C300 firmware, see the Monitor ROM listing in Appendix I of this manual. Especially note the portion from $C300 through $C420.

Pascal 1.1 Firmware Protocol

The Pascal 1.1 firmware protocol was originally developed to be used with Apple Pascal 1.1 programs. The protocol is followed by all succeeding versions of Apple II Pascal, and can be used by programmers using other languages as well.

The Pascal 1.1 firmware protocol provides Apple IIe programmers with

- a standard way to uniquely identify new peripheral cards
- a standard way to address the firmware routines in peripheral cards.

Device Identification

The Pascal 1.1 firmware protocol uses four bytes near the beginning of the peripheral card's firmware to identify the peripheral card.

Address	Value
$Cs05	$38 (like the old Apple II Serial Interface Card)
$Cs07	$18 (like the old Apple II Serial Interface Card)
$Cs0B	$01 (the generic signature of new cards)
$Cs0C	$ci (the device signature)

The first hexadecimal digit, c, of the device signature byte identifies the device class and the second hexadecimal digit, i, of the device signature byte is a unique identifier for the card, used by some manufacturers for their cards. Table 6-6 shows the device class assignments.

Table 6-6. Peripheral-Card Device-Class Assignment

Digit	Device Class
$0	Reserved
$1	Printer
$2	Joystick or other X-Y input device
$3	Serial or parallel I/O card
$4	Modem
$5	Sound or speech device
$6	Clock
$7	Mass storage device
$8	80-column card
$9	Network or bus interface
$A	Special purpose (none of the above)
$B-F	Reserved for future expansion

For example, the Apple II Super Serial Card has a device signature of $31: the 3 signifies that it is a serial or parallel I/O card, and the 1 is the low-order digit supplied by Apple Technical Support.

Although version 1.1 of Pascal ignores the device signature, applications programs can use them to identify specific devices.

I/O Routine Entry Points

Indirect calls to the firmware in a peripheral card are done through a branch table in the card's firmware. The branch table of I/O routine entry points is located near the beginning of the Cs00 address space (s being the slot number where the peripheral card is installed).

The branch table locations that Pascal 1.1 firmware protocol uses are as follows:

Address	Contains
$Cs0D	Initialization routine offset (required)
$Cs0E	Read routine offset (required)
$Cs0F	Write routine offset (required)
$Cs10	Status routine offset (required)
$Cs11	$00 if optional offsets follow; non-zero if not
$Cs12	Control routine offset (optional)
$Cs13	Interrupt handling routine offset (optional)

Notice that $Cs11 contains $00 only if the control and interrupt handling routines are supported by the firmware. (For example, the SSC does not support these two routines, and so location $Cs11 contains a non-zero firmware instruction.) Apple II Pascal 1.0 and 1.1 do not support control and interrupt requests, but such requests are implemented in Pascal 1.2 and later versions and in ProDOS.

Table 6-7 gives the entry point addresses and the contents of the 65C02 registers on entry to and on exit from Pascal 1.1 I/O routines.

Table 6-7. I/O Routine Offsets and Registers Under Pascal 1.1 Protocol

Addr.	Offset for	X Register	Y Register	A Register
$Cs0D	Initialization			
	On entry	$Cs	$s0	
	On exit	Error code	(unchanged)	(unchanged)
$Cs0E	Read			
	On entry	$Cs	$s0	
	On exit	Error code	(unchanged)	Character read
$Cs0F	Write			
	On entry	$Cs	$s0	Char. to write
	On exit	Error code	(unchanged)	(unchanged)
$Cs10	Status			
	On entry	$Cs	$s0	Request (0 or 1)
	On exit	Error code	(changed)	(unchanged)

Interrupts on the Enhanced Apple IIe

The original Apple IIe offered little firmware support for interrupts. The enhanced Apple IIe's firmware provides improved interrupt support, very much like the Apple IIc's interrupt support. Neither machine disables interrupts for extended periods.

For more about interrupt support in ProDOS, see the *ProDOS Technical Reference Manual.*

For information about interrupt handling with Apple Pascal 1.2, see the *Device and Interrupt Support Tools Manual* which is part of the Apple II Device Support Tools package (A2W0014).

Interrupts work on enhanced Apple IIe systems with an installed 80-column text card (either 1K or 64K) or a peripheral card with interrupt-handling ROM in slot 3. Interrupts are easiest to use with ProDOS and Pascal 1.2 because they have interrupt support built in. DOS 3.3 has no built-in interrupt support.

The new interrupt handler operates like the Apple IIc interrupt handler, using the same memory locations and operating protocols. The main purpose of the interrupt handler is to support interrupts in *any* memory configuration. This is done by saving the machine's state at the time of the interrupt, placing the Apple in a standard memory configuration before calling your program's interrupt handler, then restoring the original state when your program's interrupt handler is finished.

What Is an Interrupt?

An interrupt is a hardware signal that tells the computer to stop what it is currently doing and devote its attention to a more important task. Print spooling and mouse handling are examples of interrupt use, things that don't take up all the time available to the system, but that should be taken care of promptly to be most useful.

For example, the Apple IIe mouse can send an interrupt to the computer every time it moves. If you handle that interrupt promptly, the mouse pointer's movement on the screen will be smooth instead of jerky and uneven.

Interrupt priority is handled by a daisy-chain arrangement using two pins, INT IN and INT OUT, on each peripheral-card slot. As described in Chapter 7, each peripheral card breaks the chain when it makes an interrupt request. On peripheral cards that don't use interrupts, these pins should be connected together.

The daisy chain gives priority to the peripheral card in slot 7: if this card opens the connection between INT IN and INT OUT, or if there is no card in this slot, interrupt requests from cards in slots 1 through 6 can't get through. Similarly, slot 6 controls interrupt requests (IRQ) from slots 1 through 5, and so on down the line.

When the IRQ′ line on the Apple IIe's microprocessor is activated (pulled low), the microprocessor transfers control through the vector in locations $FFFE-$FFFF. This vector is the address of the Monitor's interrupt handler, which determines whether the request is due to an external IRQ or a BRK instruction and transfers control to the appropriate routine via the vectors stored in memory page 3. The BRK vector is in locations $03F0-$03F1 and ProDOS uses the IRQ vector in locations $03FE-$03FF. (See Table 4-11.) The Monitor normally stores the address of its reset routine in the IRQ vector; you should substitute the address of your program's interrupt-handling routine.

Apple Pascal doesn't use the BRK vector at $03F0-$03F1, but it does use the IRQ vector at $03FE-$03FF.

Interrupts on Apple II Series Computers

The interrupt handler built in to the enhanced Apple IIe's firmware saves the contents of the accumulator on the stack. (The original Apple IIe saves the contents of the accumulator at location $45.) DOS 3.3, as well as the Monitor, rely on the integrity of location $45, so this change lets both DOS 3.3 and the Monitor continue to work with active interrupts on the enhanced Apple IIe.

Original IIe

Since the built-in interrupt handler on the original Apple IIe uses location $45 to save the contents of the accumulator, the operating system fails when an interrupt occurs under DOS 3.3 on the original Apple IIe.

If you want to write programs that use interrupts while running on the original Apple IIe, Apple II Plus, or Apple II, you must use either ProDOS or Apple II Pascal 1.2 (or later versions). Both these operating systems give you full interrupt support, even though these versions of the Apple II don't include interrupt support in their firmware. (Versions of Pascal before 1.2 do not work with interrupts enabled on an original Apple IIe.)

Some other manufacturer's hardware, such as co-processor cards, don't work properly in an interrupting environment. If you are trying to develop an application and encounter this problem, check with the manufacturer of the card to see if a later version of the hardware or its software will operate properly with interrupts active. You may not be able to use interrupts if an interrupt-tolerant version isn't available.

Interrupts are effective only if they are enabled most of the time. Interrupts that occur while interrupts are disabled will not be serviced.

Pascal, DOS 3.3, and ProDOS turn off interrupts while performing disk operations because of the critical timing of disk read and write operations. Some peripheral cards used in the Apple IIe disable interrupts while reading and writing.

Original IIe

Although the enhanced Apple IIe firmware never disables interrupts during screen handling, the original Apple IIe periodically turns interrupts off while doing 80-column screen operations. The effect is most noticeable while the screen is scrolling.

Important! Don't use PR#6 to restart your Apple IIe while running ProDOS with interrupts enabled since PR#6 doesn't disable interrupts. If you try it, ProDOS will fail as it starts up since its interrupt handlers aren't yet set up. If you have to restart, use CONTROL-RESET, or make sure that your program disables interrupts before it ends.

Rules of the Interrupt Handler

Unlike the Apple IIc, the enhanced Apple IIe's interrupt handling firmware is not always switched in. Here are the reasons why this is so and the implications that necessarily follow.

There is *no* part of memory in the Apple IIe that is always switched in. Thus, there is no location for an interrupt handler that works for all memory configurations. However, the $C3 page of firmware is present on all systems that have 80-column text cards in their auxiliary slots, so it was selected as the starting location of the built-in interrupt handling routine.

There are two factors that determine if the $C3 firmware is switched in and therefore whether or not interrupts will be usable:

- Is there an 80-column text card in the auxiliary slot?
- If not, is there a peripheral card in slot 3 with built-in ROM with bytes $C305 = $38 and $C307 = $18?

The Apple IIe's memory is switched according to the following rules at both powerup and reset:

- If there is a ROM card in slot 3, but no text card in the auxiliary slot, the firmware on the ROM card is switched in. This is necessary for Pascal to work.
- If there is a text card in the auxiliary slot, but no ROM card in slot 3, the internal $C3 firmware is switched in.
- If there is both a text card in the auxiliary slot and a ROM card in slot 3, the firmware on the ROM card is switched in.

Important! These rules mean that systems without 80-column text cards in the auxiliary slot do not have their internal $C3 firmware switched in. Such systems cannot handle interrupts or breaks (the software equivalent of interrupts). An application program must swap in the $C3 firmware both on initialization and after reset to make interrupts function properly on such a machine configuration. (ProDOS versions 1.1 and later do this for you during startup.)

See the section "Developing Cards for Slot 3" earlier in this chapter.

Another implication of the decision to have interrupt code in the $C3 page affects the shared $C800 space in the Apple IIe. When the $C3 page is referenced, the IIe hardware automatically switches in its own $C800 space. When the interrupt handler finishes, it restores the $C800 space to the original owner using MSLOT ($07F8). This means that it is very important for a peripheral card to place its slot address in MSLOT to support interrupts while code is being executed in its $C800 space.

Interrupt Handling on the 65C02 and 6502

There are three possible conditions that will allow interrupts on the 65C02 and 6502:

- The IRQ line on the microprocessor is pulled low after a CLI instruction has been used (interrupts are not masked). This is the standard technique that devices use when they need immediate attention.
- The microprocessor executes a break instruction (BRK = opcode $00).
- A non-maskable interrupt (NMI) occurs. The microprocessor services this interrupt whether or not the CLI instruction has been used. An NMI is completely independent of the interrupts discussed in this manual.

The microprocessor saves the current program counter and status byte on the stack when an interrupt occurs and then jumps to the routine whose address is stored in $FFFE and $FFFF. The sequence of operations performed by the microprocessor is as follows:

1. It finishes executing the current instruction if an IRQ is encountered. (If a BRK instruction is encountered, the current instruction is already finished.)
2. It pushes the high byte of the program counter onto the stack.
3. It pushes the low byte of the program counter onto the stack.
4. It pushes the processor status byte onto the stack.
5. It executes a JMP ($FFFE) instruction.

The Interrupt Vector at $FFFE

Three separate regions of memory contain address $FFFE in an Apple IIe with an Extended 80-Column Text Card: the built-in ROM, the bank-switched memory in main RAM, and the bank-switched memory in auxiliary RAM. The vector at $FFFE in the ROM points to the built-in interrupt handling routine. You must copy the ROM's interrupt vector to the other banks yourself if you plan to use interrupts with the bank-switched memory switched in.

The Built-in Interrupt Handler

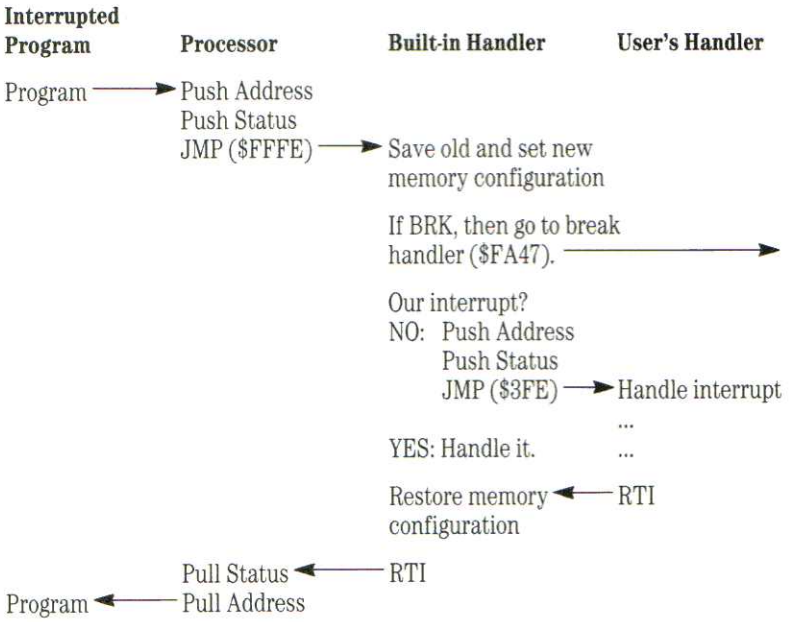
The enhanced Apple IIe's built-in interrupt handler records the computer's current memory configuration, then sets the computer's memory configuration to a standard state so that your program's interrupt handler always begins running in the same memory configuration.

Next the built-in interrupt handler checks to see if the interrupt was caused by a break instruction, and handles it as just described under "Interrupt Handling on the 65C02 and 6502." If it was not a break, it passes control to the interrupt handling routine whose address is stored at $3FE and $3FF of main memory. Normally, that would be the operating system's interrupt handler, unless you have installed one of your own.

Interrupt handler installation is described in the *ProDOS Technical Reference Manual* and the *Device and Interrupt Support Tools Manual*, which is part of the Apple IIe Device Support Tools package (A2W0014).

After your program's interrupt handler returns (with an RTI), the built-in interrupt handler restores the memory configuration, and then does another RTI to return to where it was when the interrupt occurred. Figure 6-4 illustrates this entire process. Each of these steps is explained later in this chapter.

Figure 6-4. Interrupt-Handling Sequence



Saving the Apple IIe's Memory Configuration

The built-in interrupt handler saves the Apple IIe's memory configuration and then sets it to a known state according to these rules:

- Text Page 1 is switched in (PAGE2 off) so that main screen holes are accessible if 80STORE and PAGE2 are on.
- Main memory is switched in for reading (RAMRD off).
- Main memory is switched in for writing (RAMWRT off).
- $D000-$FFFF ROM is switched in for reading (RDLCRAM off).
- Main stack and zero page are switched in (ALTZP off).
- The auxiliary stack pointer is preserved, and the main stack pointer is restored. (See the next section, "Managing Main and Auxiliary Stacks.")

Important! Because main memory is switched in, all memory addresses used later in this chapter are in main memory unless otherwise specified.

Managing Main and Auxiliary Stacks

Apple has adopted a convention that allows the Apple IIe to be run with two separate stack pointers since the Apple IIe with an Extended 80-Column Text Card has two stack pages. Two bytes in the auxiliary stack page are used as storage for inactive stack pointers: $0100 for the main stack pointer when the auxiliary stack is active, and $0101 for the auxiliary stack pointer when the main stack is active.

When a program using interrupts switches in the auxiliary stack for the first time, it must place the value of the main stack pointer at $0100 (in the auxiliary stack) and initialize the auxiliary stack pointer to $FF (the top of the stack). When it subsequently switches from one stack to the other, it must save the current stack pointer before loading the pointer for the other stack.

The current stack pointer is stored at $0101, and the main stack pointer is retrieved from $0100 when an interrupt occurs while the auxiliary stack is switched in. *Then* the main stack is switched in for use. The stack pointer is restored to its original value after the interrupt has been handled.

Important! The built-in XFER routine does not support this procedure. If you are using XFER to swap stacks, you must use code like the following to set up the stack pointers and stack.

```
*
* This example transfers control from a code segment running
* using the main stack to one running using the aux stack.
*

1     XFERALT   PHP             ;preserve interrupt status in A
2               PLA
3               SEI             ;disable interrupts
4               TSX             ;save main stack pointer at $100
5               STA SETALTZP    ;and swap zero pages
6               STX $100
7               LDX $101        ;now restore aux stack pointer
8               TXS
9               PHA             ;and interrupt status
10              PLP

11              LDA #DESTL      ;set destination address
12              STA $3ED
13              LDA #DESTH
14              STA $3EE
15              SEC/CLC         ;set direction of transfer
16              BIT RTS         ;V=1 for alt zero page (RTS=$60)
17              JMP XFER        ;do transfer
```

To transfer control the other direction, change the following lines

```
5               STX $101
6               LDX $100
7               STA SETSTDZP

16              CLV             ;V=0 for main zp
```

The User's Interrupt Handler at $3FE

If your program has an interrupt handler, it must place the entry address of that handler at $03FE. After it sets the machine to a standard state, the IIe's internal interrupt handler transfers control to the routine whose address is in the vector at $03FE.

It is very important for a peripheral card to place its slot address in MSLOT to support interrupts whenever it is executing code in its $C800 space. Whenever the $C3 page is referenced, the IIe automatically switches in its own $C800 ROM space. When the interrupt handler finishes, it restores the $C800 space to the original owner using MSLOT ($07F8).

▲Warning

Be careful to install interrupt handlers according to the rules of the operating system that you are using. Placing the address of your program's interrupt handler at $03FE disconnects the operating system's interrupt handler.

The $03FE interrupt handler must do these things:

1. Verify that the interrupt came from the expected source.
2. Handle the interrupt as desired.
3. Clear the appropriate interrupt soft switch.
4. Return with an RTI.

Here are some things to remember if you are dealing with programs that must run in an interrupt enviroment:

- There is no guaranteed maximum response time for interrupts because the system may be doing a disk operation that lasts for several seconds.
- Once the built-in interrupt handler is called, it takes *at least* 150 to 200 microseconds for it to call your interrupt handling routine. After your routine returns, it takes 40 to 140 microseconds to restore memory and return to the interrupted program.
- If memory is in the standard state when the interrupt occurs, the total overhead for interrupt processing is about 150 microseconds less than if memory is in the worst state. (The worst state is one that requires the most work to set up for: 80STORE and PAGE2 on; auxiliary memory switched in for reading and writing; bank-switched memory page 2 in the auxiliary bank switched in for reading and writing; and internal $Cn00 ROM switched in).
- Interrupt overhead will be greater if your interrupt handler is installed through an operating system's interrupt dispatcher. The length of delay depends on the operating system, and on whether the operating system dispatches the interrupt to other routines before calling yours.

Handling Break Instructions

The 65C02 treats a break instruction (BRK, opcode $00) just like a hardware interrupt. After the interrupt handler sets the memory configuration, it checks to see if the interrupt was caused by a break (bit 4 of the status byte is set), and if it was, jumps to a break handling routine. This routine saves the state of the computer at the time of the break as shown in Table 6-8.

Table 6-8. BRK Handler Information

Information	Location
Program counter (low byte)	$3A
Program counter (high byte)	$3B
Encoded memory state	$44
Accumulator	$45
X register	$46
Y register	$47
Status register	$48

Finally the break routine jumps to the routine whose address is stored at $3F0 and $3F1.

The encoded memory state in location $44 is interpreted as shown in Table 6-9.

Table 6-9. Memory Configuration Information

Bit 7 = 1	if auxiliary zero page and auxiliary stack are switched in
Bit 6 = 1	if 80STORE and PAGE2 both on
Bit 5 = 1	if auxiliary RAM switched in for reading
Bit 4 = 1	if auxiliary RAM switched in for writing
Bit 3 = 1	if bank-switched RAM being read
Bit 2 = 1	if bank-switched $D000 Page 1 switched in and RAMREAD set
Bit 1 = 1	if bank-switched $D000 Page 2 switched in and RAMREAD set
Bit 0 = 1	if internal Cs ROM was switched in (IIe only)

Interrupt Differences: Apple IIe Versus Apple IIc

If you are writing software for both the Apple IIe and the Apple IIc, you should know that there are several important differences between the interrupts on the enhanced Apple IIe and those on the Apple IIc. They are

- In the IIc ROM, $FFFE points to $C803; in the IIe ROM, to $C3FA. To ensure that the proper interrupt vectors are placed into the Language Card RAM space, always copy them to the RAM from the ROM. (When you initialize built-in devices on the IIc, these vectors are automatically updated).
- There is no shared $C800 ROM in the IIc. Peripheral cards share this space in the IIe. Thus it is crucial that the slot address of the peripheral card using the $C800 space is stored in MSLOT ($07F8). When the interrupt handler goes to the internal $C3 space, the IIe hardware switches in its own $C800 space. When the interrupt handler finishes, it restores the $C800 space to the slot whose address is in MSLOT.
- The IIc $C800 space is always switched in. The enhanced IIe's interrupt handler preserves the state of the $C800-space switch and then switches in the slot I/O space. This means that when restoring the state of the system using the value placed in location $44, break handling routines must restore one more value on the Apple IIe than on the Apple IIc.