III BITS
John Jeppson's Guided Tour of Highway III

Softalk Magazine, May 1983, pages 100-112

SOURCE
http://adtpro.sourceforge.net/archive/AppleIIIBits.html
04 December 2008

Here is a ragtag assortment of odds and ends from Apple III, thrown together almost untainted by logical sequence. Some have already been published elsewhere; some are obtainable in the fine print of Apple manuals; and some are the fruit of personal investigation. Accuracy, particularly in the latter category, may not be uniformly high. So be warned.

Let's face it. Extracting information from Apple Computer isn't the easiest thing in the world. In fact, it's usually faster, and more fun, to ask the Apple III itself. There is no obvious reason for Apple's reticence. The folks at Apple intend, they say, to publish the SOS Reference Manual and, eventually, the Driver Writer's Guide. The reference manual exists already, more or less, as a textbook for the Apple III Technical Workshop. If you're keen on writing assembly language for the III, by all means take that course. It tells you lots and lots, although not quite everything you might wish to know. Inquiries, however, seem to drift off into never-never land.

Why so secretive? The effect seems primarily to impede the efforts of would-be Apple III programmers, which you might suppose would not be in Apple's interest. Maybe they are protecting something else? The techniques of RAM-based operating systems may have a more general applicability... perhaps to the Lisa?

No doubt a recognized software development firm, prepared to sign certain agreements, can obtain source materials and technical assistance. But many bright ideas must incubate and grow and be played with on the machine before they become sufficiently clear and explicit to warrant a formal approach to Apple. A lot of maybe-

we-coulds must simply have vanished because the programmer had insufficient information to permit experimentation.

A Memory Map.

At any one time, the 6502 cpu works with 64K addresses arranged as follows:

0000..1FFF	**2000..9FFF**	**A000..FFFF**
lower s-bank	**user bank 0-6**	**upper s-bank**

The system bank is always on-line. It contains SOS.Kernel and other goodies. The user banks are switched in and out. Only one user bank is on-line at any given moment.

Table 1 describes the function of pages in the lower system bank.

Table 1. Lower system bank: pages $00..1F

00:	**"True" zero page. Used early in boot sequence, and as the zero page for interrupt handlers.**
01:	**"Normal" 6502 stack. Addressed by PHA, JSR, and so on, whenever bit 2 of environment register ($FFDF) is set. Used as stack page by interrupt handlers, drivers, and by SOS.Kernel itself.**
02..03:	**I/O buffers for floppy drivers.**
04..07:	**Text page 1. In eighty-column mode holds screen memory for even-numbered columns 0,2,..78 (decimal).**
08..0B:	**Text page 2. Memory for odd-numbered columns 1,3,..79 (decimal). Note: Corresponding addresses in TextPage1 and TextPage2 are interchanged by the relation: (high byte) XOR $0C.**
0C..0F:	**Character set.**
10..11:	**File names, prefix, ? access routes to files.**
12..13:	**Used as I/O buffer for reading directories.**
14:	**Xbyte page when zero page is $18. Used by SOS.Kernel and by drivers.**
15:	**Typeahead buffer.**
16:	**Xbyte page when zero page is $1A. Used by interpreter and by assembly modules included in user programs.**
17:	**Keyboard layout.**
18:	**System zero page. Used by SOS.Kernel and by drivers.**
19:	**SOS data and jump tables.**
1A:	**"User" zero page. Used by interpreter.**
1B:	**"Alternate" 6502 stack when zero page is $1A (zero page XOR $01).**

Used by interpreter. Alternate stack is addressed by PHA, JSR, and so on, whenever bit 2 of environment register ($FFDF) is clear.

1C..1D: Route information for open files.

1E..1F: Available for use by interpreter.

Upper System Bank: $A000..FFFF. SOS.Kernel occupies $BC00..FFFF. In the future SOS.Kernel may get longer and extend down as far as $B800. SOS.Interp, which is "absolute" code, is normally loaded below SOS.Kernel. Actually, it is loaded into the highest user bank (bank 6 in a 256K machine) beginning at some predetermined location ($7600 for Pascal). It may then extend upward for any length, up to the lower end of SOS.Kernel (presently $BC00). Thus it usually overlaps from bank 6, a user bank, into system bank $A000..BBFF. This is important because the overlap gives the interpreter a sizable area in system bank for code that is always on-line. Bank-switching must always be done from system bank. If you switch banks while running in a user bank -- puuff! Suddenly you aren't.

A very short interpreter might lie only in user bank or only in system bank. The loading site and length are determined by the writer when the interpreter is created. It is absolute code ".org'd" on the intended loading site.

Usually the upper system bank is all RAM, except for $FFD0..FFDF and $FFE0..FFEF, which are the onboard D and E VIAs (versatile interface adapters). In particular, if bit 6 of the environment register is clear, then $C000..CFFF is RAM. If that bit is set, then this area is I/O. There are also $20 bytes of RAM "under" the VIAs at $FFD0..FFEF. Normally they are off-line. These RAM bytes can be accessed only by "8F" extended addressing. This small area of RAM is unique in that it is not disturbed by a control-reset reboot, so that's where they keep the last valid clock value -- less useful, since you obtained your functioning clock chip.

The RAM of SOS.Kernel area $C000..FFFF can be write-protected by setting bit 3 of the environment register. Normally this RAM is protected while the interpreter is running. This is the user environment, and Apple doesn't trust you. It is unprotected in the driver, SOS.Kemel, and interrupt handler environments. Write-protection doesn't affect I/O $C000..CFFF when that is enabled, nor the VIA registers $FFD0..FFEF.

Highest User Bank: Bank 6 in a 256K Machine. At boot time SOS.Interp is loaded here, at whatever site the writer has designated, assuming, as is usually the case, that the interpreter is not confined entirely to $A000..BBFF in system bank. Next the drivers are loaded below SOS.Interp, one after another, in whatever order they are

encountered in SOS.Driver, which is just the reverse of the order listed by the System Configuration Program. For this purpose a modular driver is just one driver, no matter how many modular units it may contain. The drivers extend, if necessary, down to the bottom of bank 6. If more space is needed, they continue from the top of the next lower user bank ($9FFF with bank 5 switched in). Each driver, however, must be completely contained within its bank, so if there is insufficient room for the complete driver in bank 6 it will be placed entirely in bank 5. Any space left over below the drivers is free and available for requisition by the interpreter.

Interpreter Strategy. Interpreters should never assume they are resident in the highest bank, even though that is where they are normally loaded. Interpreters will run perfectly well in other banks. All you need to do is copy the user bank portion of the interpreter into the corresponding bytes in another (free and available) bank. Then switch in that bank (being careful not to self-destruct) and perform a jump to the first byte of interpreter code. The interpreter still overlaps into system bank ($A000..BBFF) just the way it always did. Bank-switching affects only the user bank. Interpreters, therefore, should always find out where they are by checking the bank register ($FFEF), never by making assumptions or by using location $1901, which does contain the highest bank number.

This relocation trick can be used for interpreter-switching schemes. A small switching interpreter is loaded from disk as the original SOS.Interp. It is placed entirely in the upper part of the highest user bank, with the drivers immediately below. The switching interpreter, in turn, loads in another interpreter (perhaps Pascal) but places the user-bank portion in a lower bank, where it runs very happily. The switching interpreter remains in the highest bank, taking up very little room, just waiting for you to call it back by pressing a special key combination (for which you will need a small modification of the console driver). Then the switching interpreter can load in, and run, some other interpreter, such as Basic.

User Bank 0: "8F" Addressing. The "zero-page anomaly" in Apple III means that every time the 6502 executes a zero-page instruction it actually operates on the designated zero page, found as the value of the zero-page register ($FFD0). The same thing happens when the (sixteen-bit) operand of an instruction has $00 in the high byte, since that also refers to zero page. Similarly, in extended addressing, if the place you are headed has $00 in the high byte of its address, then that is also interpreted as a zero-page location and you are given whatever is the current zero page. But that may not be what you want.

Extended addressing looks at a 64K stretch of memory comprising two consecutive user banks. In extended addressing, $0000 is the bottom of one user bank and $FFFF is the top of the next higher user bank. Whichever pair of user banks is active depends on the Xbyte of the extended address. This works fine except for the lowest page of the lower of the two user banks in the pair. That would have to be addressed as $00xx/8b. But the high byte of that address is $00, so you are given zero page instead.

Normally you get around the zero-page anomaly by decrementing the Xbyte. Then you are looking at a different pair of user banks, with your target bank as the higher member of the pair. The lowest page of that bank can then be addressed as $80xx/8b-1. But what about the lowest page of bank 0? There is no lower user bank to put underneath it in a pair. Hence 8F addressing.

8F is an Xbyte which, when present, causes the extended addressing mechanism to look at 64K of memory constructed as follows:

0000..1FFF	**2000..9FFF**	**A000..FFFF**
lower s-bank	**bank 0**	**upper s-bank**

It is aimost exactly like system addressing (Xbyte $00), with bank 0 switched in. (There is one other interesting feature. It is all RAM, including the RAM beneath the VIA registers $FFD0..FFEF.)

Thus if you are doing a lot of talking to user bank 0, you should use 8F as the Xbyte and address the bank as $2000..9FFF, corresponding to $0000..7FFF in the bank. Then you can get to the lowest page without worrying about the zero-page anomaly.

User Bank 0: Graphics. Which, of course, is why you'll want to be talking to bank 0. That's where graphics are, when graphics are allocated. Pascal and Basic each provide for allocations of $00, $40, or $80 pages of graphics, depending on the graphics mode. Pascal also allows $20 pages, which is enough for one lo-res black-and-white buffer. Hi-res mode appears to interleave two lo-res modes in alternate columns or groups of pixels, much as eighty-column text interleaves two forty-column screens.

The number of pages allocated for graphics is stored in location $1907 in the SOS data area. Presumably this byte is used by the video generator apparatus, as are surrounding bytes in that area.

In black-and-white lo-res mode (BW280), buffer 1 runs from $2000/208F..3FFF/3F8F (which is $0000..1FFF in bank 0). Buffer 2 is found in $4000/408F..5FFF/5F8F. In

buffer 1 the lowest byte ($2000/208F) represents the upper-left corner of the screen. Each bit represents one pixel. Successive bytes (and their contained bits) are in order from the left edge of the screen. One accesses individual bytes by indirect Y-indexed addressing (extended addressing) off the base address, which is the leftmost byte in that horizontal row. The following algorithm (for BW280, buffer 1) relates corresponding bytes in successive rows. It was discovered empirically and is doubtless pathetically slow:

next line up: **subtract $0400**
if < $2000 then add $2000 and subtract $80
if < $3C00 then add $0400 and subtract $28

next line down: **add $0400**
if >= $4000 then subtract $2000 and add $80
if >= $2400 then subtract $0400 and add $28

In hi-res mode (BW560) altemate bytes (groups of eight pixels) come from corresponding bytes of the two lo-res buffers just discussed. Thus the sequence is $2000, $4000, $2001, $4001....Hi-res buffer 2 is the corresponding structure beginning at $6000/608F. In color mode (CP280) the "upper lo-res buffer" presumably contains color information. We are not sure about COL140 mode. And we are not sure if the base address algorithm holds for these modes.

When you are ready for the video generator to display your graphics, it is necessary to fiddle with the soft switches (see table 8). Graphics information is always taken from bank 0, regardless of which user bank is switched in. Presumably this is hard-wired, although it is just conceivable that the source bank is software-selected. If so, we don't know how.

The Text Pages: $0400..07FF and $0800..0BFF. Apple III text memory is very similar to Apple II; possibly identical for forty-column mode. In eighty-column mode the two "text pages" are interleaved: even columns from $0400..07FF, odd columns from $0800..0BFF. The reason for this peculiar arrangement is found in the direct memory access (DMA) apparatus of the video generator. When Apple III was designed for an eighty-column display, the video generator had to call up twice the amount of information as it did for the forty-column display of Apple II. But it did not have twice the time in which to do it. So the memory-fetch path in Apple III was made sixteen bits wide. Every data fetch actually gets two bytes. The video generator uses both. The 6502 chip uses one and ignores the other (except in the case of the Xbyte, which is that extra byte used in extended addressing). The memory fetch does not get

two adjacent bytes. It gets the byte at address and the byte at address: high byte XOR $0C. Thus a fetch to $0400 also gets the byte from $0800, which the video generator puts in the odd column. And this is also the reason why the Xbyte page is related to its zero page by the same relation, high byte XOR $0C.

The Apple II "screen holes" are there, but you aren't supposed to use them for peripheral card scratchpad space. In the Apple III these locations are used as transfer ports when downloading character sets to the video generator. But downloading occurs only at boot time or when programs deliberately change character sets. It is relatively rare. The rest of the time these locations seem to be idle. It may be that a peripheral card could use them for a while. But it's illegal according to the definition of Apple III.

It is possible to write directly to the screen from assembly, bypassing the console driver. Just put ASCII codes in the appropriate memory locations. The high bit should be clear for inverse and set for normal, assuming you are using a standard (not inverted) character set.

The bytes in each horizontal line are accessed by X-indexed addressing off the base address, which is the leftmost byte of that line (see table 2).

If column is odd, add $0400 to the address. Use X index := column DIV 2;

Table 2. Text screen line numbers versus base addresses.

00 0400	**08 0428**	**10 0450**
00 0400	**08 0428**	**10 0450**
01 0480	**09 04A8**	**11 04D0**
02 0500	**0A 0528**	**12 0550**
03 0580	**0B 05A8**	**13 05D0**
04 0600	**0C 0628**	**14 0650**
05 0680	**0D 06A8**	**15 06D0**
06 0700	**0E 0728**	**16 0750**
07 0780	**0F 07A8**	**17 07D0**

In eighty-column mode use X := column DIV 2. If the column number is odd, you must also add $0400 to the base address given in the table. Alternatively, the base address and index may be computed with a modification of the Apple II subroutine Bascalc (table 3).

Entry: Line, column
Exit : Base address (addrL,H), X-index

```
bascalc   lda    line                    asl   A
          pha                            ora   addrL
          lsr    A                       sta   addrL
          and    #03          which      lda   column
          ora    #04                     lsr   A
          sta    addrH                   tax
          pla                            bcc   $2
          and    #18                     lda   addrH
          bcc    $1                      eor   #0C
          adc    #7F                     sta   addrH
$1        sta    addrL         $2        rts
          asl    A
```

Table 3. Subroutine Bascalc.

The Character Set: $0C00..OFFF. At boot time the system character set is loaded from SOS.Driver and stored in these pages. Similarly, if you download another character set from a program by issuing a DControl call #16 or #17 to the console driver, the new set is also placed here. But these pages are not the active character set in current use by the video generator. This is merely a staging area. From these four pages the character set is further transferred to the video generator's storage area, wherever that is. It is not in addressable memory. Presumably the machine contains a 1K RAM chip dedicated for this purpose, analogous to the ROM chip beneath the Apple II keyboard that contains the character set for that machine. In any event, you can change the copy in $0C00..0FFF all you wish, but nothing happens.

The console driver uses a complex mechanism to transfer the.character set into the video generator. It sets up an interrupt-driven background program (spooler) by allocating system internal resources (SIR) numbers $05, $06, and $10. The video-generator mechanism then interacts with the SIR#06 interrupt handler (embedded in the console driver) to transfer the character definitions at its leisure. The computer's attention is returned to the user's programs, and the video generator interrupts when it feels ready for another swallow. There may be simpler ways if you are willing to let the main program wait. For an entire character set the download procedure takes about a second to complete.

The actual transfer involves the E-VIA's peripheral control register ($FFEC), interrupt enable register ($FFEE), a couple of sites in $C000 I/O space ($C0DA and $C0DB) that are probably soft switches, and the notorious screen holes. Apparently the interrupt handler moves the character descriptions piecemeal from the $0C00 area to the screen holes and then alerts the video generator to move them on from there into its own dedicated RAM.

This transfer could probably proceed just as easily from any memory buffer to the screen holes; the $0C00 staging area is merely a convenience. But if you are operating from a background program, and if that program is the interrupt handler itself, then the buffer must be in system bank. If the buffer were in a user bank it would surely go off-line due to bank-switching. Extended addressing is not available for interrupt handlers; it doesn't work on the true zero page. Hence the $0C00 buffer.

Typehead Buffer: $l500..15FF. Page $15 is set aside for use by the console driver as a typeahead buffer. It is nothing more than a first-in-first-out queue. Actually two queues. The first queue ($1500..157F) contains KBD values, which are the ASCII codes generated by the keyboard. For each KBD there is also a KBDFLG byte, the second keyboard byte, which flags the various modifier keys. KBDFLG is stored in the corresponding byte in the second queue ($1580..15FF). The console driver maintains a count of the current number of characters in the queue and keeps index pointers to the current front and rear of the queue.

When a key is pressed, KBD appears at $C000 just as it does in Apple II. At this time KBDFLG also becomes available at $C008. The keyboard interrupt is cleared with the keyboard strobe, $C010, just as it is in Apple II.

KBD and KBDFLG are picked up by the keyboard (SIR#02) interrupt handler, which is embedded in the console driver. If they represent one of the five console control keys, that function is executed immediately. Otherwise, if a standard key was pressed, KBD would be used as an index into the keyboard layout look-up table (page $17). KBDFLG and the modified value of KBD are then stored in the typeahead buffer. The console driver will retrieve them when it feels so inclined.

Before exiting, the keyboard interrupt handler also checks to see if the "any-key" event is armed or if this is the "attention" event character. If so, the handler queues up the appropriate "event." Later, before retuming to the user program, SOS checks the event queue and transfers control to the event handler as a subroutine.

SOS Data and Jump Tables: $1900..19FF. The first few bytes on page $19 contain important status information (table 4). During ordinary business, some (or all) of these bytes control the video generator and/or similar accessory apparatus. But when the monitor is running, they have no perceptible effect. So there must be more than one way to control the video generator.

1900:	**10**	**??**
1901:	**06**	**Highest user memory bank.**
1902:	**00**	**Console control #7 and #9. Setting bit 7 suspends screen output; bit 6 will "flush" screen output. Low nibble: ??**
1903:	**00**	**High bit set indicates NMI pending.**
1904:	**8F**	**??**
1905:	**19**	**??**
1906:	**82**	**Console control #5. Clearing bit 7 turns off video. Bit 6 may be involved in graphics. Low nibble contains text mode [0..2].**
1907:	**00**	**Number of pages allocated for graphics.**

Table 4. SOS status info. Some bytes control video generator.

Page $19 also contains a jump table beginning at $1910. The jumps take the form "1913: 4C CA E2 JMP E2CA". The table provides fixed entry addresses for certain subroutines that apparently will be supported in future versions of SOS. The list is in table 5. Those marked with an asterisk are documented by Apple and are legal to use. The others... well, they do appear in the jump table.

Access	**SOS address**	*** = Legal to use**
1910	**198F**	**Probably debug.**
1913	**E2CA**	*** AllocSlR: Allocate internal resource.**
1916	**E352**	*** DealcSlR: Deallocate internal resource.**
1919	**E3C2**	**Disable reset key (unless NMI pending).**
191C	**E3F3**	**Enable reset key (just sets FFDF bit 4).**
191F	**E41D**	*** Queevent: Queue an event.**
1922	**E3A9**	*** SelC800: Grab $C800 expansion space.**
1925	**EE2A**	**Writes "system failure," the value of A, and hangs.**
1928	**EE17**	*** SysErr: reports errors from drivers to caller.**
192B	**F5C5**	**? error number look-up for internal buffer allocation.**
192E	**F686**	**? error number look-up for internal buffer allocation.**
1931	**F710**	**? error number look-up for internal buffer allocation.**
1934	**19D3**	**Probably debug (AND #20, STA 19D2, RTS).**

1985 1910 Probably debug.

Table 5. Supported SOS subroutines.

Page $19 also contains a copyright notice at $1990, a few other data bytes of mysterious function, and a lot of zeros. The subroutine SysErr stores the error number at $1980 and the return address at $19FD and $19FE. The "system-failure" routine uses the end of this page to store the program counter and all register values for use in debugging. Other SOS routines store various temporaries on this page.

The copyright notice at $1990 is a good spot if you want to store things that can be found from the Monitor. If you want to store a lot of stuff you can also use the character set area.

Those Registers: the Onboard 6522 VIAs. The two VIAs are referred to as D-VIA ($FFD0..FFDF) and E-VIA ($FFE0..FFEF) respectively. They are fully occupied with Apple III hardware manipulations. You cannot, for example, use the VIA timers for your own purposes. The VIAs manage bank-switching, zero-page selection, and much of the other machinery that permits Apple III to accommodate the 64K address space of the 6502 cpu chip.

$FFEF: Bank register (E-VIA IORA). The low nibble selects the currently switched-in bank. The high nibble is generally $F. Attempts to change the high nibble have no effect. Those four bits are flags for interrupt requests from the slots.

$FFDF: Environment register (D-VIA IORA). Table 6 lists the significance of its bits. Apple would be happier if you confined your attention to bit 7 and didn't mess with the others. Table 7 contains a variety of information about the various standard environments.

Value	Bit	Function	Bit=0	Bit=1
01	0	F000..FFFF	RAM	ROM
02	1	ROM#	ROM#2	ROM#1
04	2	stack	alternate	normal (true 0100)
08	3	C000..FFFF	read/write	read only
10	4	reset key	disabled	enabled
20	5	video	disabled	enabled
40	6	C000..CFFF	RAM	I/O
80	7	clockspeed	2MHz	1 MHz

Note: ROM#2 doesn't exist.

Table 6. Environment register ($FFDF).

(Data mostly from unpublished SOS Reference Manual)

	User	Kernel	Driver	IRQ	Monitor
Environment register	$38	$34	$74	$74	$77
Clock speed	2 MHz	2 MHz	2 MHz	2 MHz	2 MHz
I/O space	disabled	disabled	enabled	enabled	enabled
Screen	on	unchanged	unchanged	unchanged	on
Reset key	unlocked	unchanged	unchanged	unchanged	unlocked
Write protect	read only	r/w	r/w	r/w	r/w
Stack	alternate	normal	normal	normal	normal
ROM	disabled	disabled	disabled	disabled	enabled
Zero page	$1A	$18	$18	$00	$03
Xbyte page	$16	$14	$14	none	none
Bank register	unchanged	unchanged	unchanged	handler's	$F0
6502 interrupts	enabled	enabled	enabled	disabled	??
Functions allowed:					
Issue SOS call	yes	no	no	no	no
Be interrupted	yes	yes	with care	with care	n/a
Handle interrupt	no	no	no	yes	n/a
Queue event	yes	no	yes	yes	n/a
Handle event	yes	no	no	no	n/a
Allocate SIR	yes	yes	yes	yes	??
Call SelC800	see text	yes	yes	yes	n/a
Call SysErr	no	yes	yes	no	n/a

Note: Upon entry to an interrupt handler X points to a $20 byte scratchpad area on zero page. These bytes should be addressed $00,X and so on. It the interrupt source is the onboard ACIA then Y contains the ACIA status register.

Table 7. The standard environments.

$FFD0: Zero-page register. Selects the current zero page, which can be assigned to any page in memory. If alternate stack is enabled (bit 2 of $FFDF is clear) then all stack-

using opcodes use the current zero page XOR $01. Extended addressing, on the other hand, functions only for zero pages in the range $18..1F. The user zero page is $1A. That's you and/or the interpreter. Drivers and SOS.Kernel use $18. Interrupt handlers use $00. SOS is supposed to decide these things; you are not. The SOS call handling routine even checks to see if the caller's zero page is $1A. If not, it crashes the system. Somewhere in darkest Cupertino, Apple maintains a coven of witch doctors who will cheerfully do unspeakable things to your image, should you violate this trust.

$FFDD: "Any-slot" interrupt flag. When a peripheral card in one of the slots pulls down the interrupt line, the interrupt handler is entered with 6502 interrupts disabled. The interrupt handler is, of course, responsible for clearing the interrupt condition on the card. If the interrupt handler wishes to enable 6502 interrupts (as it should if it will run longer than 500 microseconds) then it must also clear the "any-slot" interrupt flag by storing $02 in $FFDD. Otherwise the interrupt manager will do it for you when the handler exits.

I/O Space: $C000..CFFF. I/O space is on-line when bit 6 of the environment register ($FFDF) is set. It is actually $C000..C4FF and $C800..CFFF. The intervening bytes $C500..C7FF are always RAM. Table 8 lists those registers of which we have some clue, There are many mysterious others. When in doubt, there is a good chance that a register's function is similar or identical to its role in Apple II.

C000: KBD. ASCII value of the most recent keypress.
C008: KBDFLG. Bits are flags for modifier keys.
C010: Clear keyboard strobe.
C020: Deselect all peripheral slots (CFFF more commonly used).
C030: Clicks speaker (Apple II type).
C040: Beeps speaker (Apple III type).

Soft switches
C050: Black and white on.
C051: Color on.
C052: Forty-column mode, low-res mode.
C053: Eighty-column mode, hi-res mode.
C054: Display buffer 1.
C055: Display buffer 2.
C056: Text on.
C057: Graphics on.

Peripheral card I/O (each slot has $10 bytes)

C090: Slot 1.
C0A0: Slot 2. Normally addressed as C080,X
C0B0: Slot 3. where X = s0 (slot number in high nibble).
C0C0: Slot 4. Note: there is no slot 0.

Onboard 6551 ACIA
C0F0: ACIADR Data register.
C0F1: ACIASR Status register.
C0F2: ACIAMR Command mode register.
C0F3: ACIACR Control register.

Peripheral card PROM space (one page for each slot)
C100: Slot 1.
C200: Slot 2.
C300: Slot 3.
C400: Slot 4.

Table 8. I/O space: $C000..CFFF.

Notice that the $10 bytes beginning $C080, $C0D0, $C0E0, and $C0F0 are not used for slots in the III. In Apple II they would be slots 0, 5, 6, and 7 respectively. There may or may not be a clue to their function in the assignment of various connecting plugs to imaginary slots in emulation mode. For example, $C0F0+ is the ACIA (asynchronous communication interface adapter), as indicated in table 9. The ACIA runs the serial port, which in emulation mode is assigned to an ethereal slot 7. Similarly, emulation mode assigns the floppies to slot 6, and the floppy drivers (buried in SOS.Kernel at $E899) probably access bytes in $C0E0..C0EF, and in $C0D0..C0DF as well. But this may only be speculation.

Entry: A = slot number ($00 deselects all slots)
Entry point: (via JMP table) at $1922

```
E3A9:    C9 05      *  CMP  #05          ; range check
E3AB:    B0 14      *  BCS  ->E3C1       ; error returns carry set
E3AD:    08         *  PHP
E3AE:    78         *  SEI               ; disable 6502 interrupts
E3AF:    8D C0 DF   *  STA  DFC0          ; save slot number
E3B2:    09 C0      *  ORA  #C0
E3B4:    8D BF E3   *  STA  E3BF          ; build instruction at E3BD
E3B7:    2C 20 00   *  BIT  C020          ; deselect strobe
```

```
E3BA:    2C FF CF   *  BIT  CFFF        ;  same
E3BD:    2C FF C0   *  BIT  C0FF        <--- ; becomes CsFF
E300:    28         *  PLP             ; restore 6502 interrupts
E3C1:    60         *  RTS
```

Table 9. SelC800 disassembler listing.

$C800..CFFF is a 2K peripheral card expansion space used "in common" by all the slots. As in Apple II it can be selected by referencing one of the peripheral card I/O locations assigned to that slot. $CFFF does deselect all slots, but $C020 (formerly the cassette output toggle) is the preferred Apple III deselection strobe.

There are some new rules for using $C800 space that are intended to mesh with Apple III's interrupt-driven operating system. You are supposed to allocate the space prior to use by calling the SOS subroutine SelC800. The slot number is passed in the [A] register on a JSR to the entry point at $1922. (See the subroutine listing in table 9.) A value of $00 deselects all slots. Note that SelC800 saves the slot number in $DFC0; this allows the interrupt manager to restore the proper card allocation should an interrupt occur. The interrupt manager routinely deselects all slots on entry and reselects the proper slot on the way out.

The documentation states that SelC800 may be called from any environment including interpreters (except an NMI handler). This turns out not to be entirely true. The subroutine builds an instruction on-the-fly by storing the slot number ORA #C0 as high byte of the operand in bit $C0FF. The bit instruction then physically enables $C800 space for that slot. But this area of SOS is write-protected while running in the user environment, so the STA instruction doesn't work and the subroutine fails without notifying you. If you want to call SelC800 from the user environment you must enable write by clearing bit 3 of the environment register ($FFDF).

There must be another soft switch somewhere. When you enter the Monitor (with control-open-apple-reset), it comes up in forty-column mode. You can change to eighty-column mode with escape-8, and back again with escape-4. From eighty-column mode you might suppose you could also change back to forty columns by fiddling with the soft switches, perhaps by reading $C052 and maybe $C054. Things change, and you can tell it's really trying hard. But no combination quite makes it. We don't know why.

System Internal Resources: SIRs. When an interrupt occurs, the interrupt manager must know which interrupt handler goes with which interrupting device and where

the handler address is located in memory. The SIR allocation scheme provides a look-up table. It also establishes "ownership" of a resource in order to prevent squabbles. Resources should therefore be allocated whether there will be interrupts or not. Somewhere in your code, place the following data table:

```
SIRADDR     .equ  SIRTABLE
SIRTABLE    .byte  00              ;SIR#
            .byte  00              ;ID code (will be assigned by SOS)
            .word  handler         ;interrupt handler address (or $0000)
            .byte  bank            ;interrupt handler bank
```

Allocation is performed by JSR AllocSIR ($1913) and deallocation by JSR DealcSIR ($1916). The 6502 registers must contain: X = SIRADDR; Y = SIRADDR+ 1; A = total number of bytes in SIRTABLE. This will be $05, or some multiple of $05 in the event that several resources are allocated at the same time. AllocSIR returns with carry clear if the resource is successfully allocated.

Table 10 lists the numbers assigned to various resources. Examination of AllocSIR suggests that the range is $00..17. There are a lot of question marks. One wonders about the digital/analog audio converter, the paddle ports, and other mysteries, such as whether the interrupt line of the MM58167A clock chip is wired up.

SIR#	Resource
00	(?)
01	ACIA
02	keyboard
03	(?) clock chip
04	(?)
05	used by console screen code 22. "SYNC"
06	character set downloader interrupts
07..0F	(?)
10	(?) character set downloader
11	slot 1
12	slot 2
13	slot 3
14	slot 4
15..17	(?) pseudo slots 5-7

Table 10. Internal system resource numbers (SIRs).

Boot Sequence: On power-up, or after control-reset, the boot process begins in ROM#1 (ROM#2 doesn't yet exist). Low-level diagnostics are performed. Then block 0 is read from the disk in the built-in drive. This is the SOS boot code and is present on every disk that has been formatted by the System Utilities program. It must be present for a successful boot. It consists of one block of "absolute" code and is loaded into the computer at $A000, where it begins to run.

The boot code begins by locating and switching in the highest bank of RAM. Then it goes back to the disk and loads in five more blocks (blocks 1..5). These are placed in $A200..ABFF. Block 1 currently contains all zeros; blocks 2..5 are the disk directory. The boot code then scans the directory and locates SOS.Kernel, which it loads into memory at $1E00..73FF.

When SOS.Kernel begins running, it promptly relocates bytes $3000..73FF into the area $BC00..FFFF. This is the functional SOS.Kernel. The loader portion is eventually overwritten and discarded. First, however, it locates and loads SOS.Interp and loads the drivers from SOS.Driver. It then initializes SOS.Kernel and each of the drivers. Finally, control is transferred to the first instruction in the interpreter and you are in business.

Data Disks: Your Own Boot Code. If you want to end up in Apple III native mode, the boot process had better find the SOS boot code in block 0 on the disk in the built-in drive. Any disks you ever expect to use as SOS boot disks must have that code. On the other hand, you may wish to create data disks that have the SOS directory structure but cannot be booted. Or you may want the disk to boot, but to end up with some entirely different operating system in the machine, such as an emulator, for example. For either of these alternatives you will want to put your own code in block 0 on your disk.

You start with a single block: the 512 bytes contained in block 0. It will be loaded and begin to run at $A000. You may then use the ROM subroutines to load in more blocks, so you can actually requisition as much space as you require. At the time your code begins to run, you will be in the Monitor or something very similar. The environment register reads $77, zero page is $03, and bank 0 is switched in. You have available all the hardware, including the VIA registers, extended addressing (with proper zero page), and all the internal resources. You do not, of course, have any of the SOS subroutines and facilities.

Your code should be assembled as "absolute" code by the Pascal 6502 assembler and should be ".org'd" on $A000. For data disks it should end in an infinite loop. Word Juggler, for example, creates data disks that, when you try to boot them, print "Can't boot Word Juggler data disk" in the middle of the screen and politely hang the computer.

Formatting Data Disks. After you've assembled your own boot code, it is a relatively simple matter to format data disks from application programs. It can be done entirely from Pascal and almost entirely from Basic. From Basic you will also need an invokable module that will write to a floppy disk by block number, just as Unitwrite will do in Pascal. The assembly source text for such a module is appended at the end of this article.

The floppy format driver (FMTDX) is activated by issuing a DControl call, code number 254 ($FE), to the appropriate driver (.FMTD1 for the built-in drive). In Pascal this is done with Unitstatus procedure. In Basic you use the Request.Inv invokable module that comes on the Apple III Basic boot disk. If you're working in assembly you just issue SOS call $83. When the DControl call is issued, the format process begins immediately. All error checking and confirmation requests must be done by your program before you issue the call.

The DControl call must specify a control list buffer. FMTDX.Driver expects a one-page (256-byte) buffer that will be reproduced on each page of the new disk. Normally this buffer should contain all zeros. The formatter places address code on each track and sector and fills the data fields with zeros, or whatever you put in your buffer. Then it quits.

You now have a formatted disk. It is not yet a SOS disk. It contains neither a directory nor the block 0 boot code. You must store those yourself from program buffers using Unitwrite (if you are working in Pascal). If you ever want to use the disk as a SOS boot disk, just copy block 0 from some other boot disk. Otherwise transfer your own code. Remember to chop off the header block, which the assembler will have placed in front of your code. Start the transfer at block 1 of the codefile.

Next you must install a directory. The minimum requirements for a usable SOS directory are listed in table 11. You must store the indicated byte values on the disk. Just put them in the proper place in the 512-byte buffer and write the whole block onto the disk.

The boot code -- blocks 0..1 (bytes 0000..03FF on the disk)

Your code, or the SOS code from another boot disk
The directory -- blocks 2..5 (bytes 0400..0BFF)
0400: 00 00 03 00
0404: Fx -- where x is the length of the desired volume name in the low nibble. The high nibble should contain F, for root directory
0405: The volume name in ASCII capitals (do not prefix with "/")
0414: 75
0422: C3 27 OC 00 00 06 00 18 01
(These last two words are 0600 = the block number of the bit map
0118 = 280 dec. = blocks on volume)
0600: 02 00 04 00
0800: 03 00 05 00
0A00: 04 00 00 00

The bit map-block 6 (bytes 0C00..0CFF)
0C00: 01 FF FF FE FE...through byte 0C22

Table 11. Minimum requirements for a SOS disk.

Word Juggler manages to write all this information onto the disk by a short segment of elegant and compact code. The utilities program uses the brute-force approach. It simply includes fourteen pages of a standard SOS structure (mostly zeros) and transfers the whole thing to disk in one piece. No wonder the utilities program is 123 blocks long.

Unitread and Unitwrite for Basic: an Invokable Module. The Device.IO.Inv invokable module contains two procedures. In Basic they are external procedures and require the perform statement (see page 162, Apple Business Basic Manual).

The procedures are

unitread (% devnum%, @ buf% (0), % length%, % block%)
unitwrite (% devnum%, @ buf% (0), % length%, % block%)

These procedures read from or write to a specified block number (block%) on the disk in a specified device number (devnum%). They transfer (length%) bytes to or from the buffer in memory. The buffer must contain enough bytes or Unitread will spill data over onto surrounding memory with disastrous results. Normally the buffer should

be dimensioned as an integer array; for example, DIM buf%(512). This buffer will contain more than enough room for two blocks (1,024 bytes).

Unitwrite is a dangerous procedure. There is absolutely no protection from errors. It is easy to write all over a disk directory, destroying it and rendering the entire disk unusable.

The device numbers of the floppy disks are .D1 = 1, .D2 = 2, and so on.

After typing in the text, save it to any pathname, perhaps Devio.Text. Then assemble it to the corresponding codefile, Devio.Code. Finally, change the name to Device.IO.Inv. If the assembler is not allowed to append the suffix .code, the file-type designation will get all screwed up and it won't invoke.

```
        .macro  pop
        pla
        sta     %1
        pla
        sta     %1+1
        .endm
;
        .macro  push
        lda     %1+1
        pha
        lda     %1
        pha
        .endm
;
        .macro SOS
        brk
        .byte   %1
        .if     "%2"<>""
        .word   %2
        .else
        .word param0
        .endc
        .endm
;
DRead   .equ    80
DWrite  .equ    81
;
buffer  .equ    0E8
;
        .proc   unitread,4
```

```
        .def    return,devnum,param0,param1
        .def    param2,length,block
;
        jmp     start
;
return  .word   0000
devnum  .word   0000
;
param0  .byte   00              ;number of parameters
param1  .byte   00              ;device number
param2  .word   buffer          ;pointer to buffer
length  .word   0000            ;bytes to read/write
block   .word   0000            ;block number to begin read/write
param8  .word   0000            ;bytes read -- result
;
start   .equ    *
        pop     return
        pop     block           ;pop procedure parameters
        pop     length
        pop     buffer
        pop     devnum
        lda     #05             ;number of parameters for DRead
        sta     param0
        lda     devnum          ;transfer one byte
        sta     param1
        SOS     DRead           ;issue DRead SOS call
        push    return
        rts
;
        .proc   unitwrite,4
        ref     return,devnum,param0,param1
        .ref    param2,length,block
;
        pop     return
        pop     block           ;pop procedure parameters
        pop     length
        pop     buffer
        pop     devnum
        lda     #04             ;number of parameters for DWrite
        sta     param0
        lda     devnum          ;transfer one byte
        sta     param1
        SOS     DWrite          ;issue DWrite SOS call
        push    return
        rts
;
```

```
.end
```

XFR.Block is a short Basic program intended to illustrate use of these procedures. It transfers whole blocks (512 bytes each) between specified block numbers on (separate) floppy disk drives.

```
10 INVOKE "device.io.inv"
20 DIM buf% (512)
30 HOME: PRINT "Transfer disk blocks utility"
40 PRINT
50 INPUT "Source device number: "; source%
60 INPUT "Destination device number: "; dest%
70 INPUT "Number of blocks to transfer (0..2): "; blks
80 length% = CONV% (blks * 512)
90 INPUT "Block number to begin reading: "; readblk%
100 INPUT "Block number to begin writing: "; writeblk%
110 PRINT: PRINT "Press any key to begin transfer": GET g$
120 PERFORM unitread  (% source%, @ buf% (0), % length%, % readblk%)
130 PERFORM unitwrite (% dest%, @ buf% (0), % length%, % writeblk%)
140 PRINT: PRINT "DONE"
```

Return to Apple /// Library page