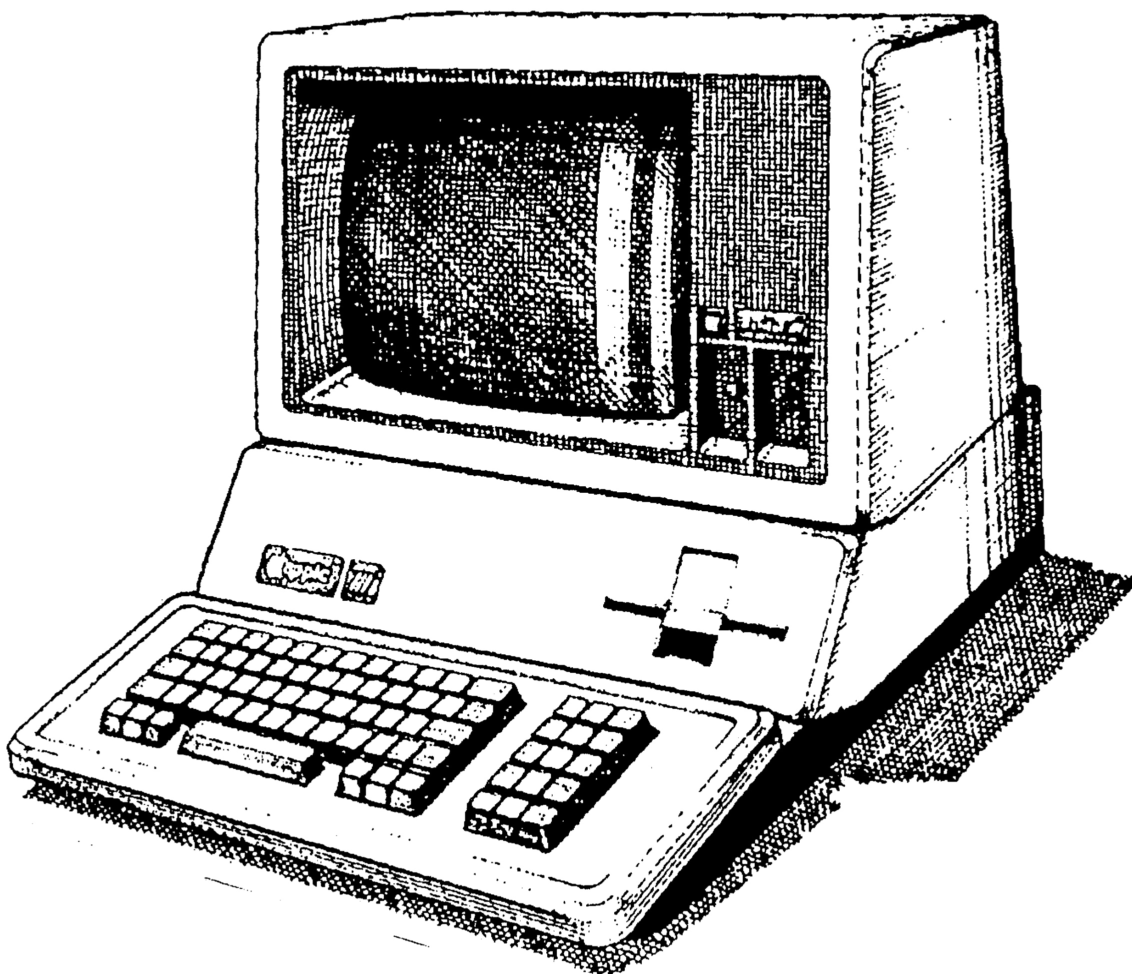
Apple /// Computer Information



DOCUMENT NAME

BANK SWITCH RAZZLE-DAZZLE

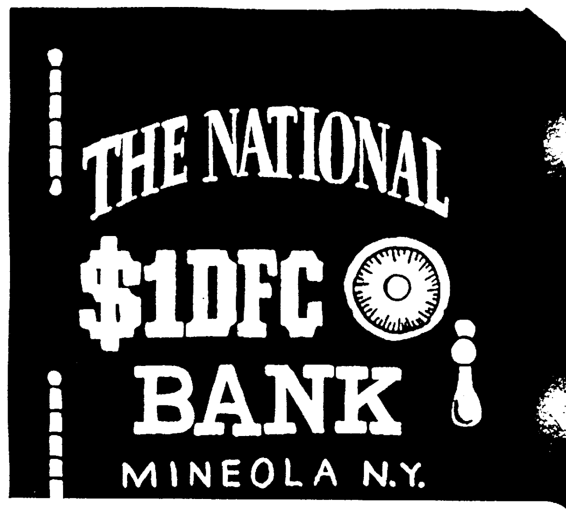
#

17

Ex Libris David T. Craig

BANK SWITCH RAZZLE-DAZZLE

Peeking and Poking The Apple III



EX LIBRIS: David T. Craig
736 Edgewater
[#________] Wichita, Kansas 67230 (USA)

BY JOHN JEPPSON

Picture the Apple II programmer perusing an *Apple III Basic Manual*. Much nodding and smiling. So powerful, so easy . . . so many new built-ins.

But wait. Something's missing. Where are they? Try the contents. Not there. The index? Not in there either. How about he list of reserved words? Here we go: *pdl, perform, pop, prefix$*. Good grief! They've left out *peek* and *poke*!

Doubtless you're in shock. The Apple III's creators left out *peek* and *poke*. They say you don't need them, that the Apple III's operating system takes care of all that. SOS they call it (pronounced "sauce"), the sophisticated operating system. Big Brother in binary.

The weird part is, they're right. You don't really need *peek* and *poke*. The Apple II has a lot of little peeks and pokes that in the Apple III are done directly with Basic statements or by writing control codes to the device drivers. And the big pokes—well, if you're going to insert an assembly language routine, there's a proper way to go about it. You're supposed to fire up Pascal, use the assembler to encode your machine language program, and call up the resulting code as an invokable module from Basic. Is this really possible? Certainly. In fact we're going to do it right here and now. And what assembly language routine shall we write? Why peek and poke, of course. Ha! We'll fix 'em.

What they say is: even if you had peek and poke, it wouldn't do you much good. SOS is constantly moving things around inside. You never know where SOS is going to put something, so how can you peek at it? To a certain extent this objection is valid. SOS loads program segments and places variables wherever it finds room at the moment; only SOS knows where. And while most variables remain at the same address once allocated, some don't even do that. If you make a series of assignments to a Basic string:

```
)Xstr$="abc": Xstr$="cdef": Xstr$="ghijk"
```

each Xstr$ is stored in a new place. So how do you know where to peek? the argument runs.

Peeking Toms. Of course, you may not want to peek just at your own programs. Perhaps what you really want to do is to look at the operating system. Sizable chunks of the operating system do have reasonably predictable addresses that might somehow be exploited. But that is just what those friendly folks at Apple want you not to do. They have provided a great variety of "legal" ways to use the operating system, such as powerful language packages, standard drivers that include very fast graphics, and assembly language modules that may include some thirty-six different SOS calls. But they don't want you messing around in the operating system directly. This policy is not merely to protect trade secrets. While it's true that *SOS.Kernel*, the central part of SOS, is considered proprietary information, Apple Computer has few worries about that. You won't soon unravel the complexity of SOS.Kernel unless you're so bright that you're wasting a national treasure by spending brain time on the task.

There's a more important reason for keeping peek and poke out of applications programs. The Apple III is not intended to be a static, finished product. Rather, it is an evolving computer system. Improvements are expected; indeed, they have already begun. And these improvements will be made to *your* existing machine by simply booting another disk that incorporates the changes. Apple wants your programs to run properly on the advanced Apple IIIs of the future. And they will, if you simply conform to the rules and stick to the tools provided. If your program uses "carnal knowledge" of the operating system and takes shortcuts by poking some magical spot, well, that spot probably won't be there after the next upgrade. And you'll be back to square one.

So why write peek and poke? It's not that we harbor an overwhelming compulsion to pollute the system with "illegal" programs. We'd just like to know what's going on in there.

Congratulations, It's a Chip. Like the Apple II, the Apple III uses the 6502 microprocessor chip. But the 6502 cpu has only a two-byte program counter. That is, it handles memory addresses that are only two bytes, or sixteen bits, long. Now it's an inescapable fact that there are just 64K (2^16 = 65,536) different combinations of sixteen binary bits, so it would appear that the 6502 limits a computer to 64K bytes of memory. How does the Apple III handle four times that much? It turns out there are two distinctly different ways to do this: *bank switching* and *extended addressing*. The Apple III uses both.

Think of the computer as a black box. Imagine that inside the box there is a smaller box. We'll call it a "switch box." Inside that switch box is the 6502. The function of the switch box is to shield the 6502 from the hard realities of life; to delude it into thinking that it lives in a nice, simple 64K machine. In other words, all the 6502 ever sees—all it knows about—is a 64K

stretch of memory. This keeps it very happy. What the 6502 doesn't know is that the 64K bytes of memory it's using are a bit slippery. They aren't the same bytes from one microsecond to the next. The switch box watches the 6502 and when the 6502 isn't looking swaps chunks of the real 128K (or 256K) memory in and out of the active 64K that the 6502 is using. The 6502 just goes on about its business, oblivious to the changing universe around it.

Fetch, 6502. Fetch! When the 6502 wants a bit of data, it performs a memory fetch. What happens is that the desired address, one of the 64K memory locations, appears on the sixteen address lines of the 6502 chip. That is, each of the sixteen address pins on the chip is given either high voltage or low voltage to represent 0 or 1 in the corresponding address bit. Because the chip has only sixteen such pins, the addressable memory is only 64K bytes. To address more memory you'd need extra pins. With seventeen address lines you could access 128K, with eighteen lines you'd get 256K, and so on. In the Apple III, the extra address lines are supplied by the switch box. The 6502 doesn't know it, but the memory chips actually get a twenty-bit address that's sixteen bits from the cpu and four bits from the switch box. The switch box (really some extra circuitry watching the 6502) gets those extra bits from one of the memory registers—hexadecimal location $FFEF, the *bank register*.

Not all 64K bytes that the 6502 uses are swapped. Locations $0000 to $1FFF and locations $A000 to $FFFF, a total of 32K bytes, comprise the *system bank* and are always on line. The other 32K, locations $2000 to $9FFF, constitute one of several different 32K user banks that can be switched in. The bank chosen is indicated by the value of register $FFEF. Thus, if $FFEF contains 2, the 6502 will be dealing with 64K locations made up of:

```
$0000..$1FFF + $2000..$9FFF + $A000..$FFFF = 64K
   Bank S         Bank 2         Bank S
```

The low nibble (bits 0,1,2,3) of $FFEF can contain sixteen different numbers, 0 through 15, or hexadecimal, $0 through $F. One of these numbers, $F, is reserved for a special purpose, but each of the remaining fifteen numbers ($0 through $E or 0 through 14) represents a bank of 32K memory bytes that might be switched in. So this scheme can handle 15 x 32K (480K) plus the 32K S-Bank, or 512K bytes of addressable memory. The present hardware maximum is 256K bytes, but the scheme has room for more in the future.

Don't Bank on It. Bank switching is great if you don't have to do it very often. You can run along in one bank for a while, then switch and run in another. But if you're running in one bank and want to fetch some data from a table in another bank, it's cumbersome. And slow. Several operations are required: you must store a new value in the bank register, fetch the data, and then switch back again by restoring the original contents of the bank register. All for one byte of data. Furthermore, the program code that actually does the switching must be located somewhere in the system bank (the part that's not switched). If the program were running in the switched-in bank at the moment it decided to change the bank register, it would instantly dematerialize itself, a form of suicide reminiscent of killing your own ancestor in a time warp. Suddenly, you never were. The 6502, of course, goes blithely on and fetches the next instruction from the corresponding spot of the newly switched-in bank. The results are generally strange.

Fortunately it is possible in the Apple III to access any byte of memory, in any bank, directly, and with a single operation. This technique is called *extended addressing* and works with any of the 6502 instruction codes that use the zero-page indirect indexed addressing mode. For example, the instruction *LDA ($2B),Y* tells the 6502 to load into the accumulator the contents of that memory byte whose address is found by adding the contents of the Y-register to the address stored in zero-page locations $002B and $002C. That "indirectly" obtained address is the one placed on the sixteen address lines of the 6502 chip.

Take Me to the $1600 Page, and Step on It! But, to access more than 64K memory locations, you need more than sixteen address lines. Once again, the switch box does the trick. Whenever the switch box sees that the 6502 is performing one of those indirect indexed instructions, it quickly adds in the extra address bits. In this case the extra bits are obtained from a memory register called the *Xbyte*. Each zero-page location ($0000 to $00FF) has "associated" with it another memory location at the corresponding spot in the $1600 page ($1600 to $16FF). Thus, if an address is stored as the contents of locations $002B and $002C (a total of sixteen bits), the Xbyte (the extra bits for the extended address) will be the contents of location $162C. So when your program performs a zero-page indirect indexed instruction, the address actually used is twenty

bits wide. And twenty bits is more than enough to get all the memory in the computer.

To the more technically minded reader, a question immediately arises. How is the value that is stored in the Xbyte memory location actually delivered to the circuitry of the switch box? Does the 6502 have to perform extra load and store operations, or what? It sounds like it might be very slow, but it isn't. The mechanism is peculiar, even bizarre. In fact, you should probably skip the next several paragraphs completely. No? Well, the story starts back with the Apple II.

The Apple II, like all respectable computers, transfers information to the video screen by direct memory access(DMA). The screen display must be refreshed and rewritten about sixty times every second, and the information used comes from a pattern of bytes stored in a specified stretch of memory. In the Apple II, the text currently on screen occupies memory locations $0400 to $07FF, a total of four pages, each consisting of 256 bytes (1,024 bytes of memory). So a lot of memory has to be accessed every second just so you can read the screen. Collectively this region of memory is known as Text Page 1. (We'll capitalize Text Page to distinguish it from a page of memory, which is a sequence of 256 bytes whose addresses all have the same high-byte.) The Apple II video Text Page is actually four pages of memory.

A Helpful Vulture. Information is not transferred to the screen by the 6502 cpu chip. That would be very slow and would tie up the cpu with a task unworthy of its time. Instead, there's additional circuitry in the video output section that watches the cpu. Periodically the cpu gets busy churning away inside itself and the address lines fall idle. The video DMA circuits then seize the address lines and make a quick data fetch of their own. Because DMA uses the address lines only when the cpu doesn't need them, the 6502 buzzes right along and doesn't even realize what's happened.

The Apple II also sets aside another 1,024 bytes of memory as an alternate source of video information. This is Text Page 2, which occupies the adjacent region of memory, locations $0800 to $0BFF. These two regions are identical, so any particular spot on the video screen is mapped from corresponding memory locations in each of the two regions. The corresponding spots will always be $0400 memory bytes apart. Thus, $04A3 and $08A3 represent the same screen location as stored in the two Text Pages, respectively. Now it just so happens that in the binary number system the Boolean statements *$4 X-OR $0C=$8* and *$8 X-OR $0C=$4* are both true. (X-OR is the Boolean operator *exclusive OR.*) This means that it's easy for the computer to move from a spot in one Text Page to the corresponding spot in the other. It's just the page number (the high-byte of address) *X-OR*'d with *$C*. And just why this is relevant to Apple III will emerge forthwith.

When it came time to design the Apple III, it was deemed desirable to incorporate an Apple II emulation mode. So, at least in emulation mode these two regions of memory continue to be used for video. Thus the Text Pages were kept around. It was also deemed necessary that the Apple III, in its native mode, have an eighty-column text screen instead of the forty columns of the Apple II.

Double Vision. The change to eighty columns presented a problem. Exactly twice as much data must be moved from memory to screen with every video refresh. DMA must access twice as many memory locations in the Apple III as it did in the Apple II—and it really can't take twice as long to do it. What to do? Well, there are those two separate text screens. Why not use them both simultaneously? So in the Apple III the memory access path was made sixteen bits wide.

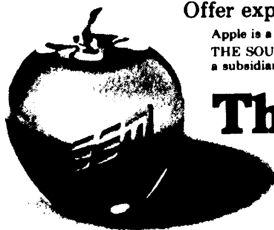
Every time a "fetch" is sent out over the address lines, the fetch returns not one, but two bytes of memory: the byte requested and a corresponding byte from the memory location with page number *X-OR $C*. For DMA, this is great. It retrieves both text pages simultaneously and they are inter-

Apple™ users. THE SOURCE℠ and TRANSEND™ Together for the first time for only $89.

Buy our $89 Transend state-of-the-art data communications software and membership in THE SOURCE, AMERICA'S INFORMATION UTILITY℠, is included. This combination allows you to easily access one of the world's largest information services for up-to-the-minute news and sports, stock prices, travel services and much more.

An optional 260-page Source User's Manual is available for $19.95. Your dealer has complete details. Call 800-227-2400, ext. 912 (in Calif. 800-772-2666, ext. 912) for the name of your local Transend dealer.

Offer expires Oct. 1, 1982.



Apple is a trademark of Apple Computer Inc.
THE SOURCE and AMERICA'S INFORMATION UTILITY are service marks of Source Telecomputing Corp., a subsidiary of The Reader's Digest Association Inc.

The Transformation People.

SSM Microcomputer Products Inc.
2190 Paragon Drive, San Jose, CA 95131

woven on the screen; a byte from one, a byte from the other. You end up with forty-column screens superimposed, with the second display shifted one-half column sideways. And there's your eighty columns. Of course that extra byte is also returned in an ordinary memory fetch by the 6502. But in that case it is simply ignored.

At this point some absolute genius at Apple, whose name we don't know, figured out that the extra byte returned by an ordinary memory fetch could be exploited. This extra byte could provide the extra bits of information needed to extend the address space of the 6502 beyond the 64K (sixteen-bit) limitation. When the 6502 performs a zero-page indirect indexed operation, it must go out to memory to the designated spot on zero-page and get the address it will use for the data fetch. Since the address will be sixteen bits, or two, bytes long, it must go to zero-page twice. First it gets the low-byte of the address, then the high-byte. But the switch box has been watching the 6502 for signs of just this type of operation.

The Prodigal Byte Returns. When the high-byte of the address is returned, the switch box grabs the extra byte that is returned with the address but normally neglected. That is the Xbyte. The information from the Xbyte is quickly placed on the "extra" address lines, and there's the 6502 addressing a full 512K bytes of memory. So the Apple III gets extended addressing more or less for free, as a by-product of the DMA video apparatus. Furthermore, it's very quick, adding a maximum of one clock cycle to the five-clock-cycle memory operation. That Apple engineer, whoever he is, really earned his pay the day he thought that up.

"Aha," you say, "hold on just a minute. That extra memory byte from the DMA business is supposed to come from the location at page number X-OR $C. But when you go after the address you are accessing zero-page, and $0 X-OR $C is not $16. You said the Xbyte was located on page $16. What gives?" (Aren't you sorry you didn't skip all this? You were warned.)

Actually we have another little deception. Zero-page is not really zero-page. It is actually page $1A, and $1A X-OR $C is indeed $16. So the Xbyte is coming from the right place. What's happening is that the poor innocent 6502 thinks it's getting zero-page but it's really being fed something else. It's that switch box again. Every time (really, every time!) the 6502 tries to access any location on zero-page, the switch box yanks the zero-page reference off the address lines and substitutes another "zero-page." In this case it is the page whose number is stored in location $FFD0, the *zero-page register.* So in the Apple III there are a bunch of zero-pages. Languages and user programs are assigned zero-page $1A (locations $1A00 to $1AFF), SOS uses zero-page $18, and interrupt-handling routines use the true zero-page $0000 to $00FF. It's actually possible to designate any page as the zero-page, but the Xbyte extended addressing mechanism works only for zero-pages in the range $18 to $1F.

If you aren't already dazed, you will be overjoyed to learn that the 6502 *stack-page* is also switched when the zero-page is switched. Normally the 6502 considers the stack to reside permanently on page $01, that is, locations $0100 to $01FF. But page $01 can also be thought of as zero-page ($00) X-OR $1. In the Apple III, any instruction that uses the stack (PLA, PHA, and so on) actually uses the current zero-page X-OR $1. So if the zero-page is $1A, then the stack is on $1B. But in this case the reassignment process can be independently disabled by changing one of the bits in yet another special register, ($FFDF) the *environment register.* (Maybe next time.)

A Word from Mad Ave. To clear your mind, try contemplating one of the mysteries of the advertising world. The Apple III, as you may know, does not have the field completely to itself. There's a tiny company in New York, with a little-known three-letter name, that has recently moved into the "small" computer business. You've probably never heard of it. Anyway, the machine they make has a sixteen-bit cpu. A veritable revolution according to the ads. But its memory is only nine bits wide—eight data bits and a parity bit. So every time that machine performs a memory fetch it gets just eight bits of data. Nevertheless, because of its cpu it calls itself a sixteen-bit machine. The Apple III, on the other hand, returns sixteen bits of data with each memory fetch. Does that make it a sixteen-bit machine too? It's hard to say. Maybe it depends on who writes the advertising copy.

Extended addressing is really much easier to use than it is to explain. The key to success is to fix firmly in your mind that the 6502 must, at all times, have 64K bytes of memory to work with. No more, no less. All the complicated swapping around must conform to that principle. Most of the time the cpu sees the upper and lower sections of system bank with one of the user banks switched into the middle:

```
$0000..$1FFF + $2000..$9FFF + $A000..$FFFF = 64K
   S-bank        user bank       S-bank
```

We'll call this *ordinary addressing.* Each user bank is 32K bytes long, which is $8000 in hexadecimal. We can talk about a particular byte in a particular bank by using a number in the range $0000 to $7FFF (which is the size of the bank). Location bank 2/byte $43DE is a spot a little above the middle of bank 2. Now the switch box is going to take this whole bank and place it somewhere in the field of 64K bytes that the 6502 is looking at. In "ordinary addressing" as we have just defined it, that bank will start at what the 6502 is now calling $2000. So far as the 6502 is concerned, that location just discussed will be found at $63DE ($43DE + $2000).

In the lower section of the system bank ($0000 to $1FFF), one page will appear twice. If the 6502 looks at page $1A ($1A00 to $1AFF), it does indeed get page $1A. But if it looks at zero-page ($0000 to $00FF), it also gets page $1A, assuming, of course, that the value of the zero-page register ($FFD0) is $1A—as you will normally find it to be. And yes, the stack-page is also, usually, "duplicated," depending on that bit in the environment register ($FFDF).

Cpu, Meet a New Array. Extended addressing presents the cpu with an entirely different 64K byte array. This occurs only during the data fetch portion of a zero-page indirect indexed operation. The 6502 has already got the address and is now going after the actual data. The Xbyte determines just which chunks of memory are presented to the 6502. For extended addressing, the Xbyte must read $8n where *n* can be any hexadecimal digit. If the Xbyte reads $0n, you'll just get ordinary addressing as defined above.

In extended addressing the cpu sees a pair of user banks, banks n and n+1. For example, if the Xbyte is $80, then the 6502 is looking at:

```
$0000..$7FFF + $8000..$FFFF = 64K
   bank 0         bank 1
```

Alternatively, if the Xbyte is $81, then the 6502 sees:

```
$0000..$7FFF + $8000..$FFFF = 64K
   bank 1         bank 2
```

Notice that a location in bank 1, say $13AB, can be found either as $13AB of bank pair 1,2 (Xbyte = $81) or as $93AB of bank pair 0,1 (Xbyte = $80). It's the same memory byte, but it can have more than one address depending on where it is placed in the 6502's field of vision.

Pascal and Basic each leave a chunk of zero-page for you to use for extended addressing in your own assembly language routines. In Pascal you are given $00E0 through $00EF, and in Basic you get $00E8 through $00F7. These regions overlap, so if you are careful you can use the same assembly language routine with both languages. Suppose you want to load the accumulator with the contents of byte $341D of bank 1. We'll use Xbyte = $81, which looks at bank pair 1,2. The zero-page location where we will store the address (pointer) is $00E8 and $00E9. First, store the desired address pointer on zero-page:

```
LDA  #$1D  ;low byte at lower location
STA  $0E8
LDA  #$34  ;high byte at higher location
STA  $0E9
```

Then store the Xbyte at the corresponding spot in the Xpage:

```
LDA  #$81
STA  $16E9
```

After setting the appropriate value in Y-register ($0 in this case), we fetch the data with:

```
LDA @$0E8,Y ;same as "LDA ($0E8),Y"
```

Note the alternative Apple III indirect notation using @ and omitting all parentheses. The assembler will also accept standard 6502 notation. By simply incrementing the Y register you may step through a whole page of memory without changing the zero-page pointers at all.

There is one problem. Remember that a reference to zero-page *always* results in a swap for the current zero-page whose number is stored in $FFD0. If you want to look at the lowest page of bank 1, say, $0023/bank 1, you can't get there by asking for $0023 of bank pair 1,2 (Xbyte = $81). You will just be given $1A23 from the system bank because you've made a zero-page reference. Instead you must ask for location $8023 of bank pair 0,1 (Xbyte = $80). It's the same place, but you have avoided the zero-page reference problem.

Lower than Low. "Ah," you say, "but what do you do about the bottom page of bank 0? There is no bank number lower than 0, so you can't perform the same trick." That's a good question, especially since bank 0 is where the Apple III puts its graphics, and you may want to meddle with the graphics screen from assembly language. Several areas of the screen are in that lowest page of bank 0. In this case there is a special technique, *$8F addressing.* Use extended addressing with Xbyte = $8F. This produces a 64K block that looks like ordinary addressing with bank 0 switched in. The desired page will now be found as $2000 through $20FF.

$8F addressing has another handy feature. In all other forms of addressing, the area $FFD0 to $FFEF is very special. These locations are not actually in RAM at all. They are on the two VIAs (versatile interface adapters) that the Apple III uses for all sorts of goodies including part of the "switch box" mechanism responsible for the fancy footwork. All the special registers are in this area, and it is always on line. The corresponding locations in RAM are normally not available, but $8F addressing is all RAM, including the thirty-two bytes of RAM "under" the VIAs. And what is squirreled away there? Why, the system clock, of course. That's why the clock is protected when the Apple III is rebooted.

The environment register really deserves a separate article of its own. Table 1 lists the function of its bits without discussing them.

Down to Business. The accompanying assembly language program contains the function peek and the procedure poke. It depends primarily on extended addressing but, regrettably, uses less legal methods as well. After assembly, the resulting code can either be linked to a Pascal program or invoked from Basic as an invokable module. It works the same way in both languages.

Peek is a function and returns an integer value, the contents of the memory location at which you've peeked. The function requires two parameters. You must supply the address (as viewed by the 6502) and the Xbyte. Both are passed as integers. In Pascal you declare peek an external function:

```
function peek (addr, xbyte : integer) : integer;
    external;
```

You may then make an assignment statement to an integer variable:

```
int := peek (addr, xbyte);
```

In Basic the process looks like this:

```
10 INVOKE "peek.poke.code": REM the pathname of the codefile.
100 int = EXFN%.peek(%addr,%xbyte)
```

Poke is similar, but since it doesn't return anything (except, occasionally, disaster), it is a procedure. It has a third parameter, the value to be poked. *Value* must also be an integer.

In Pascal:

```
procedure poke (addr, xbyte, value : integer);
    external;
```

then one can use:

```
value := 128;
poke (addr, xbyte, value);
```

In Basic:

```
10  INVOKE "peek.poke.code"
100 value = 128
110 PERFORM poke(%addr,%xbyte,%value)
```

Don't forget that the variables are all decimal integers. You may want to enter them and display them as hexadecimal strings, but you will have to convert. Basic has handy built-ins: HEX$(integer) and TEN(hexstring). In Pascal you will have to write your own.

The *address* can be any legal, ordinary integer. *Value* and *Xbyte* must be integers in the range 0 to 255. If they are greater, the integer *MOD 256* is used. Only certain Xbyte values have meaning; all the rest are treated as 0. Table 2 has some useful Xbytes and some comments. There are a couple of peculiarities that you should know about:

1. Nothing terrible happens if you give the Xbyte of a bank pair that doesn't exist (yet)—for example, ($8C = 140). Peek will either return $FF, signifying nothing, or some value from one of the existing banks—also of little use.

2. The artificial Xbyte $FF (decimal 255) isn't actually used as an Xbyte. It is merely a signal to the function to do all sorts of illegal things to the environment register, zero-page register, and interrupts in order to get at areas normally inaccessible. With this "Xbyte" you get a block that looks like ordinary (system) addressing but with "true" zero-page and "true" ($01) stack-page. Also, the area $C000 to $CFFF is "I/O space," and $F000 to $FFFF is the read-only memory used in the boot process.

Bit	Value	Function
0	0	$F000..$FFFF = RAM
	1	= ROM
1	0	ROM = ROM #2 (but it doesn't exist)
	1	ROM = ROM #1 (if switched in with bit 0)
2	0	alternate stack (= zp x-or $1)
	1	normal stack (page $01)
3	0	$C000..$FFFF — read/write
	1	— read only (write protected)
4	0	RESET KEY — disabled at keyboard
	1	— enabled
5	0	Video — disabled
	1	— enabled
6	0	$C000..$CFFF — RAM
	1	—I/O space
7	0	Clock speed — 2 mhz.
	1	— 1 mhz.

Table 1. The environment register, $FFDF.

Hex	Decimal	Result
$00	0	"ordinary" system bank. User bank at $2000..$9FFF
$80	128	bank pair 0,1
- -	- -	- -
$82	130	bank pair 2,3 — bank 3 nonexistent in 128K machine
- -	- -	- -
$86	134	bank pair 6,7 — bank 7 nonexistent in 256K machine
$8F	143	like system bank. Bank 0 to $2000..$9FFF. ALL RAM!
$FF	255	"artificial" — gives a system type bank with 1. "true" zero-page and stack-page 2. $C000 to $CFFF = I/O space 3. $F000 to $FFFF = ROM

Table 2. Xbyte values.

Note: There are locations on $C100 page of I/O space that will cause the computer to "hang" just by reading them. It really isn't dangerous, but you'll have to reboot.

A Program by Any Other Name. Boot up Pascal, enter the editor, and type in the program. Capital letters are not required. Neither are the comments, but it would be a shame if you left out all of them. You can save a lot of typing by just typing in peek and duplicating it with the copy buffer. Then go through and make the necessary changes to convert one of them to poke. Save the program on disk. Use a path name of ten characters or less and permit the editor to add the suffix *.TEXT* to your path name (for example, *peek.poke.text)*.

Next, enter the assembler and assemble the program. The assembler will want to add the suffix *.code.* Let it. Otherwise the resulting file will not be type named code file and will not invoke properly. Later you can change the name (for example, *peek.poke.inv*) and the type name won't be affected.

The output of the assembler *is* the invokable module. Move it to your Basic disk and invoke it by its path name. You can then use either peek or poke at will in your program. Details of the required Basic program syntax may be found starting on page 160 of the *Apple III Basic Manual.*

Pascal is even simpler. You just declare peek and poke as *external* and use the linker to add them to your program.

PEEK.POKE.TEXT — Source Code

```
.MACRO   POP
PLA
STA      %1
PLA
STA      %1+1
.ENDM
.MACRO   PUSH
LDA      %1+1
PHA
LDA      %1
PHA
.ENDM
ADDRESS  .EQU    0E8           ;zeropage "pseudo" register
BANKSW   .EQU    0FFEF
ZEROPG   .EQU    0FFD0
ENVRMT   .EQU    0FFDF
         .FUNC   PEEK,2
         JMP     BEGIN
RETURN   .WORD   0
XBYTE    .WORD   0
RESULT   .WORD   0
OLD_XBT  .BYTE   0
OLD_ZPG  .BYTE   0
ENV      .BYTE   0
BEGIN    POP     RETURN
         PLA                   ;"dummy" bytes for function
         PLA
         PLA
         PLA
         POP     XBYTE         ;parameters come off in reverse order
         POP     ADDRESS
         LDA     ADDRESS+1601  ;save original x-byte value
         STA     OLD_XBT
                               ;which bank is desired
         LDA     XBYTE
         CMP     #0FF          ;FF = ROM #1, C0-CF = I/O,
         BEQ     SPECIAL       ;    "true" 00 and 01 pages
         CMP     #80
         BMI     SYSTEM        ;80-8F=extended addressing
         CMP     #90           ;    else system bank (ordinary 6502)
         BMI     EXTEND
                               ;handle system bank
SYSTEM   LDY     #0
         STY     ADDRESS+1601  ;xbyte = 0 so get ordinary 6502
         LDA     @ADDRESS,Y    ;    indirect indexed addressing
         STA     RESULT
         JMP     DONE
                               ;handle extended addressing to a
                               ;    bankpair or $8F
EXTEND   STA     ADDRESS+1601  ;place extend byte
         LDY     #0
         LDA     @ADDRESS,Y    ;"extended" addressing to desired
         STA     RESULT        ;    bank pair
```

Bill Budge's
Raster Blaster
Three animated claws
trap the ball if they are enabled. When three balls become trapped, all are released for exciting multi-ball play.
Three sets of targets
test your aim and timing. Hit all of them to enable the claws.
Plus kickers, thumper-bumpers and an animated spinner help to provide unmatched realism.
Real pinball flippers
make this a game of strategy & skilled shot making.
Animated shields
can shoot a lost ball back into play.
Raster Blaster
for the Apple II and the Apple II Plus may be the first Apple II game that is copied for the arcade machines. It is so technically sophisticated and fun to play that it is sure to attract the big arcade manufacturers. But you can get it right now for your Apple!
Bill Budge's
Dealer inquires invited:
BudgeCo, 428 Pala Ave
Piedmont, CA 94611
(415)658-8141
VIDEO PINBALL FOR THE APPLE II
Requires a 48K Apple II
Apple II is a registered trademark of Apple Computer Inc.

```
         JMP     DONE
                                 ;handle artificial bank 'FF'
SPECIAL  LDA     ADDRESS+1
         BEQ     TRUEPGS         ;true $00, $01 desired?
         CMP     #1
         BEQ     TRUEPGS
                                 ;ROM#1 —> F0000-FFFF,
                                   C000-CFFF—>I/O
         PHP                     ;save status, then disable interrupts
         SEI                     ;(an "illegal" move)
         LDA     ENVRMT          ;save environment
         STA     ENV
         LDA     #73             ;#% 0111 0011 — new environment
                                   reg
         STA     ENVRMT          ;(an "illegal" move)
         LDY     #0
         STY     ADDRESS+1601    ;system bank xbyte = 00
         LDA     @ADDRESS,Y
         STA     RESULT
         LDA     ENV             ;restore ENVRMT
         STA     ENVRMT
         PLP                     ;restore status (including interrupts)
         JMP     DONE
                                 ;desired address on true 00 or
                                   01 page
TRUEPGS  PHP                     ;save status, then disable interrupts
         SEI                     ;(an "illegal" move)
         LDX     ADDRESS         ;load BEFORE leaving old z-page
         LDY     ADDRESS+1
         LDA     ZEROPG          ;save old zpg
         STA     OLD_ZPG
         LDA     #0              ;changes zero-page to 0, stack to 1
         STA     ZEROPG          ;(an "illegal" move)
         TYA                     ;is high byte 00 or 01
         BEQ     $1
         LDA     0100,X          ;indexed addressing (x = addr)
         JMP     $2
$1       LDA     0000,X
$2       STA     RESULT
         LDA     OLD_ZPG         ;restore ZEROPG (and stack page)
         STA     ZEROPG
         PLP                     ;restore interrupts (status)
DONE     LDA     OLD_XBT         ;restore Pascal's xbyte
         STA     ADDRESS+1601
         PUSH    RESULT
         PUSH    RETURN
         RTS
         .PROC   POKE,3
         JMP     BEGIN
RETURN   .WORD   0
XBYTE    .WORD   0
VALUE    .WORD   0
OLD_XBT  .BYTE   0
OLD_ZPG  .BYTE   0
OLD_ENV  .BYTE   0
ENV      .BYTE   0
BEGIN    POP     RETURN          ;parameters come off in reverse order
         POP     VALUE
         POP     XBYTE
         POP     ADDRESS
         LDA     ADDRESS+1601    ;save original x-byte value
         STA     OLD_XBT
         LDA     ENVRMT          ;save ENVRMT
         STA     OLD_ENV
         AND     #0F7            ;for POKE, enable write C000 to FFFF
         STA     ENVRMT
                                 ;which bank is desired
         LDA     XBYTE
         CMP     #80
         BMI     $1              ;80-8F=extended addressing
         CMP     #90             ;    else system bank (ordinary 6502)
         BMI     EXTEND
                                 ;disallow certain addresses
$1       LDA     ADDRESS+1       ;POKE disallowed at (system bank):
         CMP     #0FF            ;    BANKSW = FFEF
         BNE     $2              ;    ENVRMT = FFDF
                                 ;    ZEROPG = FFD0
         LDA     ADDRESS
         CMP     #0D0            ; in this program — suicide certain
         BEQ     DONE            ; in your program — suicide probable
         CMP     #0DF
         BEQ     DONE            ;if you really want to crash, just start
         CMP     #0EF            ;    POKing into SOS
                                 ;      (RAM $8800 = FFFF)
         BEQ     DONE            ;    soon he will get very sick
                                 ;detect artificial bank 'FF'
$2       LDA     XBYTE
         CMP     #0FF            ;FF = ROM #1, C0-CF = I/O
         BEQ     SPECIAL         ;    "true" 00 and 01 pages
                                 ;handle system bank
SYSTEM   LDY     #0
         STY     ADDRESS+1601    ;xbyte = 0 so get ordinary 6502
         LDA     VALUE           ;    indirect indexed addressing
         STA     @ADDRESS,Y
         JMP     DONE
                                 ;handle extended addressing to a
                                   bankpair or $8F
EXTEND   STA     ADDRESS+1601    ;place extend byte
         LDY     #0
         LDA     VALUE
         STA     @ADDRESS,Y      ;"extended" addressing to desired
                                 ;    bank pair
         JMP     DONE
                                 ;handle artificial bank 'FF'
SPECIAL  LDA     ADDRESS+1
         BEQ     TRUEPGS         ;true zp or $01 desired?
         CMP     #1
         BEQ     TRUEPGS
                                 ;ROM#1 —> F000-FFFF,C000-CFFF
                                   —>I/O
         PHP                     ;save status, then disable interrupts
         SEI                     ;(an "illegal" move)
         LDA     ENVRMT          ;save environment
         STA     ENV
         LDA     #73             ;#% 0111 0011 = new environment
                                   reg
         STA     ENVRMT          ;(an "illegal" move)
         LDY     #0
         STY     ADDRESS+1601    ;system bank xbyte = 00
         LDA     VALUE
         STA     @ADDRESS,Y
         LDA     ENV             ;restore ENVRMT
         STA     ENVRMT
         PLP                     ;restore status (including interrupts)
         JMP     DONE
                                 ;desired address on true 00 or 01
                                   page
TRUEPGS  PHP                     ;save status, then disable interrupts
         SEI                     ;(an "illegal" move)
         LDX     ADDRESS         ;load BEFORE leaving old z-page
         LDY     ADDRESS+1
         LDA     ZEROPG          ;save old zpg
         STA     OLD_ZPG
         LDA     #0              ;changes zero page to 0, stack to 1
         STA     ZEROPG          ;(an "illegal" move)
         LDA     VALUE
         CPY     #0              ;is high byte 00 or 01
         BEQ     $1
         STA     0100,X          ;indexed addressing (x = addr)
         JMP     $2
$1       STA     0000,X
$2       LDA     OLD_ZPG         ;restore ZEROPG (and stack page)
         STA     ZEROPG
         PLP                     ;restore interrupts (status)
DONE     LDA     OLD_XBT         ;restore Pascal's xbyte
         STA     ADDRESS+1601
         LDA     OLD_ENV         ;restore C0-CF read/write status
         STA     ENVRMT
         PUSH    RETURN
         RTS
         .END
```

2514 Rio Vista Drive
Bakersfield, CA 93306
(805) 872-1216

John Jeppson is an anesthesiologist who lives in Bakersfield, California. A Harvard and Boston Med School graduate, he has done some aerobatic stunt piloting in his own Citaborea (spell it backward). In 1981, he traded up from a TI-59 programmable calculator to the Apple III, where he now performs loops, rolls, and hammerhead turns that baffle even the folks at Apple.