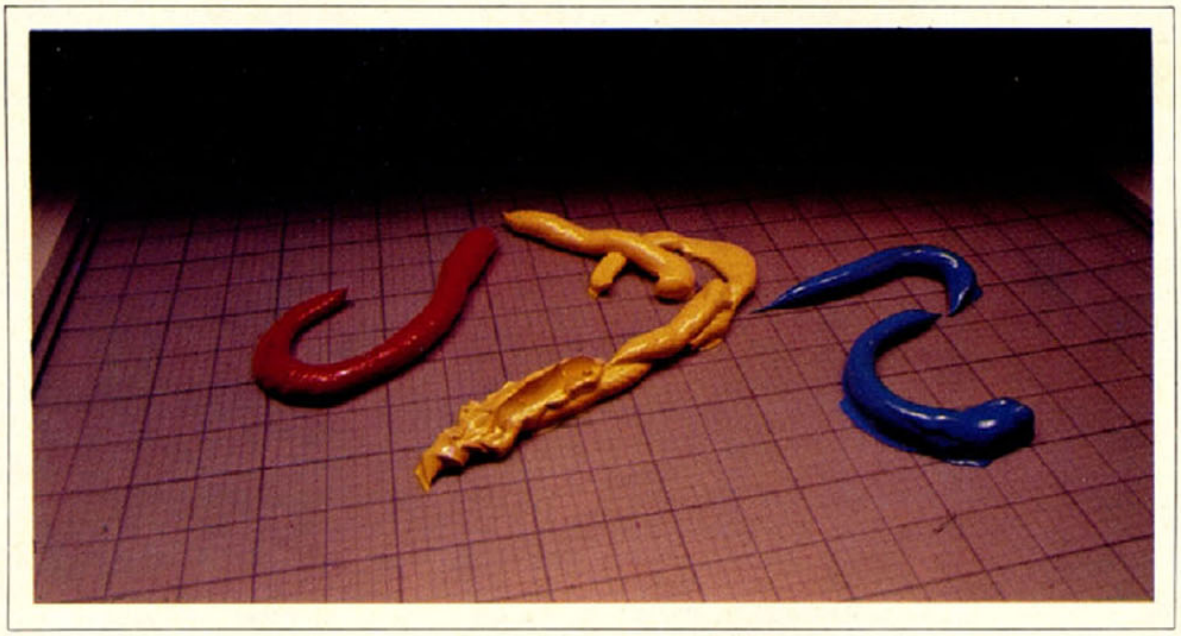
Apple II Graphics: An Inside Look



If you have been impressed by the remarkable graphics displays in some of your favorite game programs, you've probably also become curious about how these graphics were generated on the Apple. You may have read far enough in the *Applesoft Tutorial* or the *Apple II Reference Manual* to recognize the words bit, byte, Peek, Poke, hi-res and lo-res. But it may not be particularly clear yet, despite all the effort you have put in, just what these terms have to do with the colorful, entertaining graphics that dance across your screen when you play your favorite games.

If you are interested in an inside look at how graphics are generated, and in learning how to create graphics to illustrate and enhance your own programs, read on. This ongoing series of articles is definitely for you.

Of Peeks, Pokes, Bytes, and Bits. Learning about graphics means wading through some rather technical (and seemingly irrelevant, but essential) material. However unrelated such material may appear, being able to comprehend and apply what you learn here is crucial to your understanding and being able to make full use of the graphics capability of your Apple later on.

This series will acquaint you with the Apple II's graphics hardware. While knowledge of assembly language is not necessary to an understanding of this material, some experience with Applesoft is recommended.

In this first article, we will explore such topics as the binary and hexadecimal number systems, the System Monitor, ROM, and RAM. You will learn how RAM is laid out, and where and how graphic images are generated and stored in RAM. Once you have this foundation, you can learn various animation techniques, methods for incorporating graphics into programs, and how to construct graphic images. Other future columns in the series will cover such topics as font editors and character set animation.

ROM and RAM Remember. The graphic images that can be generated on your TV screen or video display monitor are possible because of the Apple's memory. As you know, the Apple has two kinds of memory, ROM (read-only memory), which is virtually unalterable, and RAM (random-access memory), the working memory area of your Apple, where program information is stored. The hi-res graphics pages, which will be of great interest to us in this series, are located in RAM.

Another familiar term about which you will learn more in this series is the Monitor. As you have probably surmised from the capital M, we are referring here not to the TV screen or video monitor, which displays the material you type in at the Apple keyboard, but to the System Monitor, a ROM-resident set of routines that enables you to look at the various RAM locations, and to move or alter (patch) them in order to achieve various kinds of results.

Digressing into Digital. In order to understand the memory usage of Apple graphics, you must first understand the numbering system your Apple likes to speak to you in, since this is related to the way information is stored in RAM. This numbering system is called hexadecimal.

But even before discussing hexadecimal, we must digress briefly and talk about digital electronics and the binary number system, both of which are essential to an

understanding how the hexadecimal number system functions.

In relation to computers, digital electronics means, in very loose terms, that information is represented to the computer only by the presence or absence of voltage.

It is much simpler to test for the presence or absence of voltage in a circuit than it is to try to determine how much voltage is present. Imagine, for instance, the degree of complexity and accuracy that would be required to represent correctly a number such as 42,183 in terms of voltage level only.

Digital electronics, through which the precision of today's microcomputers is achieved, concerns itself with only two possible values, on and off. All modern microcomputers, including your Apple, use digital electronics. Digital electronics is based on the binary numbering system. In binary, there are only two possible numbers or values, zero and one.

Making It Happen with Zeros and Ones. Memory within the computer consists of a series of on and off switches. The convention is to call these switches bits. A bit always has a value of on or off, and, once set, a bit retains its value until the computer is turned off (and the values stored in RAM are lost), or the bit is modified. Bits are usually thought of as representing either a zero or a one, with zero indicating that no voltage is present, and one denoting the presence of voltage.

Having only two possible values in a circuit results in one very obvious constraint. It means that only two values can be represented by any given bit. With one bit, then, it would be possible to design a computer capable of counting up to one (0,1).

With two bits, four different values can be represented (twice as many as were possible with only one bit), and with three bits, eight values can be represented. As illustrated in Figure 1, the number of values that can be represented expands exponentially as the number of available bits increases.

The Apple has at its disposal hundreds of thousands of bits. These bits are grouped together by the computer to represent characters. Bit groupings are assigned based on two considerations: first, fewer wires are needed to access a group of bits than to access each individual bit; and second, a bit grouping should be big enough to represent one character (A, B, C, etc.) of the language in use on the computer.

When bits (individual electronic switches) are grouped by the computer as a single unit (a number, letter, or other value), the result is called a byte. Like most other microcomputers, the Apple groups eight bits into a byte. Eight bits is a convenient number to work with, because then any one of 256 possible values can be represented using only a single eight-bit byte. This arrangement works out quite nicely, since it means that any one of the keys on the Apple keyboard, including the upper case, lower case, and control equivalents, can be represented.

The total number of 256 possible combinations that can be achieved by one eight-bit byte was arrived at by taking two to the eighth. (Remember, each time a bit is added, twice as many possible values can be represented.) All of the instructions your Apple can execute are byte oriented. It cannot reference more, and it cannot reference less, than a byte of memory at a time.

In case you've ever wondered why the maximum amount of RAM the Apple II can contain is 64K, the

one bit

on
off

two choices

2^1

two bits

first bit	second bit
on	on
on	off
off	on
off	off

four choices

2^2

three bits

first bit	second bit	third bit
on	on	on
on	on	off
on	off	on
on	off	off
off	on	on
off	on	off
off	off	on
off	off	off

eight choices

2^3

Figure 1

reason is that the Apple's 6502 microprocessor uses two bytes to represent the addresses of items stored in memory. Two Apple bytes equal sixteen bits, and sixteen bits can be arranged in two to the sixteenth unique ways. If you do the math, you will discover that two to the sixteenth is 65,536, the true number of bytes the Apple can address. Many larger computers use three bytes for addressing; the extra eight bits in the third byte multiply by 256 the number of possibilities that can be represented, so with three bytes, a computer can address over sixteen million memory locations.

You may already have noticed that the operand of the Peek and Poke statements in Applesoft is always between 0 and 255. This is so because the thing being Poked is a

byte. Unfortunately, it is hard to know the settings of the bits in a byte just by knowing the decimal value of the byte.

Hex to the Rescue. Here's where hexadecimal comes in. In hexadecimal, a byte is thought of as being composed of two four-bit nibbles. If you do the calculation, you'll find that two to the fourth gives you sixteen.

The main assumption in hexadecimal is that people would never be able to recall 256 separate bit combinations, but that anyone can remember sixteen combinations if really pushed. Also, the inventor of hexadecimal was able to arrive at sixteen symbols to represent a possible sixteen values but would have been hard pressed to come up with 256 symbols.

The sixteen symbols selected were zero through nine and the letters A through F. Zero through nine represent the first ten possible values, A the eleventh possible, F the sixteenth. Thinking too hard about hexadecimal is likely to make your head hurt, so you may want to make a chart for yourself (like the one in Figure 2) of the hexadecimal values and tape it to your wall as a reminder.

The contents of a byte when expressed in hexadecimal is really the value of the byte's two nibbles. For example, FF means two nibbles with all their bits on, and AA means every other bit on, while F0 means four bits on in

Bit 1	Bit 2	Bit 3	Bit 4	Hexadecimal	Binary
off	off	off	off	0	0 0 0 0
off	off	off	on	1	0 0 0 1
off	off	on	off	2	0 0 1 0
off	off	on	on	3	0 0 1 1
off	on	off	off	4	0 1 0 0
off	on	off	on	5	0 1 0 1
off	on	on	off	6	0 1 1 0
off	on	on	on	7	0 1 1 1
on	off	off	off	8	1 0 0 0
on	off	off	on	9	1 0 0 1
on	off	on	off	A	1 0 1 0
on	off	on	on	B	1 0 1 1
on	on	off	off	C	1 1 0 0
on	on	off	on	D	1 1 0 0
on	on	on	off	E	1 1 1 0
on	on	on	on	F	1 1 1 1

Figure 2

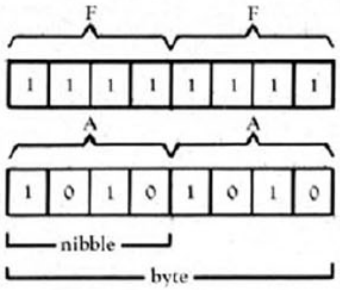


Figure 3

the first nibble and four bits off in the second nibble (see Figure 3).

The convention when writing a hexadecimal value is to precede it by a $. For instance, $DF. The value of a nibble ranges from zero to fifteen, with A equal to ten and F equal to fifteen. This means that the hex value of BC is converted to decimal by the equation eleven times sixteen plus twelve. Remember, $B is eleven and $C is twelve.

Resist the Temptation to Convert. In general, when dealing directly with the memory of your Apple, it is better not to think about converting hexadecimal numbers to decimal. It is not always necessary to convert, and it is often more efficient to try to think in hexadecimal, since that is the numbering system your Apple understands. When you do need to convert a hex number to decimal, such as for use in an Applesoft Poke statement, use the following algorithms:

For one-byte quantities (read the number from left to right)

decimal value equals (value of first nibble times sixteen) plus (value of second nibble)

For two-byte quantities (read the number from left to right)

decimal value equals (value of first nibble times 4096) plus (value of second nibble times 256) plus (value of third nibble times 16) plus (value of fourth nibble).

Memory addresses within the Apple's RAM are represented using two bytes, with values ranging from $0000 to $FFFF. If you apply the second algorithm to $FFFF, it will become clear once again why 65,536 is the biggest number in the Apple world.

To Memory through the Monitor. The ROM-resident system Monitor allows you to inspect, modify, or move the contents of RAM memory. It also contains commonly used subroutines that may be used by your programs to do such things as printing a character or making a beep sound.

Within the context of what you will learn here, the main use for the Apple Monitor is as a means of altering or moving RAM and displaying the contents of various locations. You will also learn to use the Monitor to enter short machine language routines that will be used to manipulate graphics from your programs.

As you have noticed, when you are in Applesoft, the prompt character] appears on your video screen. When you are in the Monitor, you receive the prompt character * instead. To reach the Monitor (assuming you have DOS up), enter CALL -151 from Applesoft. To return to Applesoft from the Monitor, enter 3D0G and press return.

Once you have gotten yourself into the Monitor, you

Memory Map of a 48K Apple II

Function	Address
APPLE MONITOR	$F800-$FFFF
APPLESOFT	$E000-$F7FF
RESERVED	$D000-$DFFF
I/O DECODE	$C000-$CFFF
DOS	$9600-$BFFF
UNUSED	$6000-$95FF
HI-RES PAGE 2	$4000-$5FFF
HI-RES PAGE 1	$2000-$3FFF
UNUSED	$C00-$1FFF
TEXT/LO-RES PAGE 2	$800-$BFF
TEXT/LO-RES PAGE 1	$400-$7FF
DOS VECTORS	$3C0-$3FF
UNUSED	$300-$3BF
TEXT INPUT BUFFER	$200-$2FF
6502 STACK	$100-$1FF
ZERO PAGE	$0-$FF

Figure 4

can begin examining RAM. RAM in your Apple is laid out as shown in Figure 4.

RAM, which can be thought of as the working or program area of the Apple's memory, consists of 256 pages, each of which can hold 256 (sixteen times sixteen) bytes of information.

Now that you are in the Monitor and have an idea of how things are laid out, try displaying some memory. Enter C00 and press return. You should receive the response C00- XX. XX will be some hexadecimal byte, the contents of which depend on the program last run on your Apple.

Now try changing the value of C00. Enter C00:FF (you do not need the leading $ sign, since the Monitor always presumes you are speaking to it in hex). The colon Monitor command (equivalent to a Poke statement in Applesoft) causes memory to be modified to reflect the value that follows the colon.

To prove that memory location (address) C00 now contains the value FF, try displaying the contents of C00 by entering it and pressing return. This time, you should receive the response C00- FF.

Making Monitor to Memory Moves. If you want to alter the contents of eight bytes of memory starting at $C00, enter C00:FF FF FF FF FF FF FF FF. This will store the value FF at locations $C00 through $C07. Then, to prove that you have actually patched memory, enter C00.C07, which asks the Monitor to display the bytes in memory from $C00 through $C07. Remember, when you wish to display a range of memory, just enter its address and press return. When you wish to display a chunk of memory, enter the starting address, a period, and the ending address. (Do not use spaces to separate these elements.)

This method of patching memory works fine if you only want to alter the contents of a few memory locations. But what if you have reason to change a larger range of memory, such as the contents of the entire hi-res page one ($2000.$3FFF). Typing 2000:FF FF FF FF . . . to 3FFF:FF would accomplish your aim, but it would take several hours.

Fortunately, the Apple Monitor contains a routine that will move entire blocks of memory in only seconds. To discover how the routine works, display the contents of $C00 through $C40 by entering C00.C40. Then, pretend you want to move what you have just displayed into the $D00 range of memory. To accomplish this, type D00<C00.C40M. This command tells the Monitor subroutine that $D00 is to receive <, the contents of memory stored in $C00 through $C40. The letter M stands for move. If you now display the contents of $D00.D40, it should contain exactly the same data as the memory range at $C00.C40. (Despite the fact that this command is called a "move" command, it does not mean that data is transported from one set of locations to another, but rather that data is copied and stored in a second location in addition to the first, original one.)

Clearing the Screen for Graphic Action. Now it's time to do something useful with what you've learned so far. The task is to try to clear hi-res page one ($2000–$3FFF) to black. This is similar to, but not exactly the same as, executing the HGR command in Applesoft. But besides clearing the screen to black, the Applesoft command will give you a four-line scrolling text window at the bottom of the screen. What you want to do right now is to clear the hi-res screen to black without putting a four-line text window at the bottom of your video screen.

First, store the value 00 at $2000 by typing 2000:00.This happens to be the bit pattern that causes your Apple to display black, because it means that all the bits in memory location $2000 are in the zero, or off, position. Next, clear the memory locations from $2001 to $3FFF by typing "2001<2000.3FFEM." Then, to verify that hi-res page one has actually been cleared, type "2000.3FFF" and watch all the zeros appear on your screen. Unfortunately, since the hi-res graphics were not "turned on," you cannot see your black screen, but you can ascertain from all of the zero values that the memory move command worked.

You will learn shortly how to control what is displayed on the screen, but first it is important to take a closer look at the move command you just executed. Whenever you use the Monitor move command, it works a byte at a time, going from left to right. When you moved memory to itself as you did earlier, several things happened. First, $2001 was loaded with the value from $2000, which had been set to 0 earlier. Next $2002 was loaded with the value that was just moved out of $2001, $2003 received the value moved out of $2002, and so on, until the last element in the chain was carried out when $3FFF received the value that had been contained in $3FFE. Although this may sound a bit complicated right now, you will soon become comfortable with it, and find ways to make use of it in the future.

How to Tell What's on TV. In the next article in this series, you will examine the hi-res graphics mode in detail and discover how memory is used to create the images

you see on your video display monitor. You will now learn how to turn the different graphic modes on and off from the Monitor.

The Apple's RAM contains a series of switches called soft switches. These are actually memory locations, but they differ from other memory location switches, in that merely referencing their addresses in a program causes something to happen. (This is in contrast to normal switches, which can be altered by plugging new values into them, as we did in the earlier example.)

The graphics soft switches lie in RAM at $C050.C057 (for those of you who have not yet become accustomed to thinking in hex, that's 49232–49239). There are other soft switches in the Apple's RAM, but right now you only need concern yourself with the ones that affect the graphics.

Soft Switch Lives Here. Watch what happens when you mention to the Monitor an address that happens to be the location of a soft switch. Typing C050 will turn on a graphics mode, with a four-line text window at the bottom. As far as the Monitor is concerned, all you did was try to display the contents of memory location $C050. But since all that has to be done to flip a soft switch is to reference its address, you have gone directly into a graphics mode instead.

To switch to hi-res page one with a four-line text window at the bottom, type C057. If you ever want full screen hi-res graphics (sans the text window), type C052. To restore things to normal (that is, to return to text), type C051.

Here is a summary of the procedures to follow in order to turn on the various graphics modes from the Monitor or from Applesoft.

For hi-res page one, full screen graphics, enter C050 from the Monitor or Poke 49232,0 from Applesoft. Then enter C057 from the Monitor or Poke 49239,0 from Applesoft. Then enter C052 from the Monitor or Poke 49234,0 from Applesoft.

To get to hi-res page two (full screen, no text window), enter C055 from the Monitor or Poke 49237,0 from Applesoft.

To get back to hi-res page one, enter C054 from the Monitor or Poke 49236,0 from Applesoft.

For a split screen again (one with a four-line text window), enter C053 from the Monitor or Poke 49235,0 from Applesoft.

To set lo-res graphics, enter C056 from the Monitor or Poke 49238,0, from Applesoft.

To return to text mode once again, enter C051 from the Monitor or Poke 49233,0 from Applesoft.

And Now for a Bit of Fun. Now that you know how to put values into memory and how to turn on graphics, it's time to play with what you've learned.

Go into the Monitor (CALL -151) from Applesoft. Enter C050, then C057, then C053. Doing this will turn on hi-res graphics, with a four-line text window at the bottom of the screen. To clear the screen area to black, enter 2000:0 and then 2001<2000.3FFEM. You should now be looking at a blank screen, with your last four lines of text at the bottom.

What would happen if you were to plug some other value into $2000? Try it. Enter 2000:FF. A little white line should appear in the upper left hand corner of your TV screen. Now try entering the values $01, $02, $04, $08, $08, $10, $20, $40, $80. These values were selected because each of them represents a byte with only one bit on in each of eight possible positions, as illustrated in Figure 5.

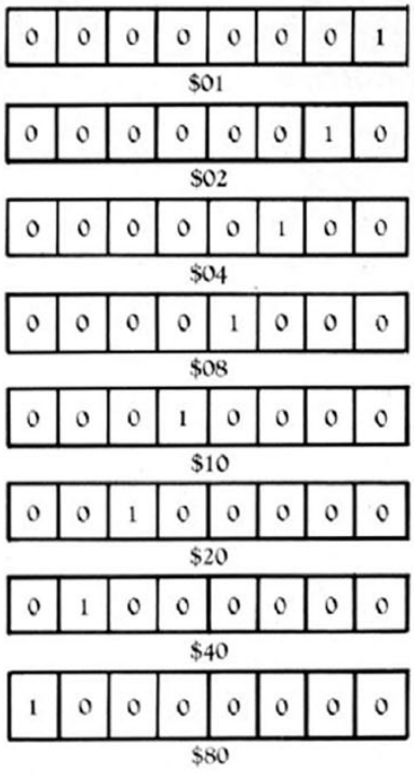


Figure 5

Entering these values should yield some surprising results. If you watched closely on the TV when you entered 2000:02 (after already having entered 2000:01), you saw the dot that was on the TV screen move to the right, even though the on bit within the byte you poked moved to left. This pattern should continue until you enter 2000:80, at which point the screen should once again go to black.

What this tells you is that only the last seven bits in a byte turn on a dot on the television screen, and that the bits turn on dots in reverse of the way you would expect them to. Try entering bit patterns with more than one bit on (for example, 42) and see if you can predict what you'll see on the TV screen.

Don't worry if the last portion of this lesson has you confused. It will be repeated in greater detail in the next issue.

Homework. Read about hi-res graphics and the System Monitor in your *Apple II Reference Manual*. Even though these sections are written in rather technical language, you're likely to find that whatever you do get out of them will enable you to get a little fancier in your applications of the examples we worked with here.

Have fun.

Although we haven't talked much yet about hi-res graphics, the ground we've covered so far is essential to your understanding and being able to take advantage of the graphics capability of your Apple II. If you understood everything you read in this article, you will be well prepared for your next hi-res graphics lesson. If you did not understand this article, chances are you won't be able to make much sense of the next one either.

If you have any questions or comments about information in this article, please write to me care of *Softline*. Then in subsequent articles, I can try to answer your questions and clarify any material that may have been confusing the first time around.

Apple II Graphics:
Mapping the Memory Maze

In this issue, we'll examine the processes by which the Apple displays information on your video monitor or television screen. We'll also take a brief look at the three possible Apple graphics display modes.

But first, let's talk further about the inner workings of the Apple's memory and about how the Apple produces the text and graphic images that appear on your screen.

Memory Mapped Output. All images the Apple displays on your video monitor screen come about through a process known as *memory mapped output.* A separate piece of hardware inside your Apple is constantly looking at and interpreting the contents of memory in order to determine what should be displayed on the video screen.

When the Apple is displaying a certain kind of information on the video screen, it is said to be in that particular mode. A *mode* can be thought of as a condition or set of conditions under which certain rules apply. The Apple has three methods of interpreting the contents of memory: text mode, lo-res graphics mode, and hi-res graphics mode.

In *text mode,* each byte of memory contains the ASCII code for one character. The monitor screen is mapped into a grid comprising forty horizontal and twenty-four vertical character positions. This translates into twenty-four lines of text, each line containing forty character positons.

In *lo-res graphics mode,* each byte of memory can be used to code for two colored blocks. The monitor screen is mapped into forty-eight horizontal rows and forty vertical columns. The grid that results is made up of colored blocks.

In *hi-res graphics mode,* each byte of memory contains the code for producing seven colored dots. The monitor screen is mapped into a grid made up of two hundred eighty vertical columns and one hundred ninety-two horizontal rows. Each of the first seven bits in every byte of the area of memory that controls the hi-res screen can code for one dot in this 192 by 280 grid.

Knowing Where You're Going. As you've probably surmised, the computer needs some way of knowing which one of these three display modes to go into, as well as what memory locations to draw from. Here's where the so-called soft switches we talked about last time come in.

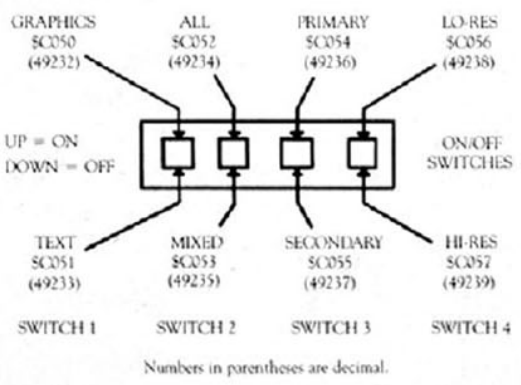


Figure 1

The *soft switches* (four in all) are called soft because they are controlled by the software of the computer. As you'll recall, soft switches can be thought of as on/off switches which activate or deactivate the display modes. Besides the text, lo-res graphics, and hi-res graphics modes, a special mode that mixes graphics with text is provided.

The way to flip a soft switch is to cause the Apple to reference the memory address of its on or off position. This can be done either by entering a memory address from the Monitor or by peeking at the location from Applesoft. For instance, to turn on switch one (select a graphics mode), from Basic you would either enter C050 from the Monitor or PEEK 49232. Remember, the actual value you peek is random gibberish that should be ignored.

Switch one simply serves the function of specifying whether text or a graphics mode is desired. When you first turn on your machine, the Monitor turns off switch one, putting you in text mode. If you flip switch one on, you'll get graphics; which graphics mode is displayed will depend on the settings of the other switches.

If you flip *switch two* to the on position, you're on your way to getting a split screen of graphics and text in which the first three quarters of the screen is in graphics mode, and the bottom quarter is in text mode. But you'll only get this split screen if switch one is on, specifying graphics rather than text mode. This mixed graphics/text mode is very handy and is put to good use in many popular games, including hi-res adventures, in which a player needs to interact with the program in order to play or solve the game.

Switch three permits you to choose between two possible areas of memory through which to display text or graphics.

The System Monitor turns on switch three when you turn on your machine. This switch will become more meaningful to you later on when we explore some animation techniques that use it.

Switch four, which affects the screen display only if switch one is on, specifies whether the computer is to display hi-res or lo-res graphics.

Text mode is the mode in which normal character text is displayed. It's likely that most of the time when you use your Apple, you'll be in text mode, at least at first.

Sometimes, you'll see text faked through the use of the hi-res graphics screen. For instance, some word processing programs rely on this technique in order to be able to represent both the upper and lower-case equivalents of text characters on screen without a lower-case chip.

Translating Text Characters into Numbers. The standard, factory-direct Apple has the capability of displaying only sixty-four different characters. These sixty-four characters are the twenty-six characters of the alphabet (upper case only), twenty-eight special characters (such as parentheses, quotation marks and so on) and ten numerals (zero to nine).

Text coding on the Apple is accomplished by means of the American Standard Code for Information Exchange

(ASCII). Since the computer can only deal with numbers, it needs some way of translating text characters into numerical form. The ASCII code serves this function.

In ASCII, each possible text character is represented by a number from 0–127. This means that seven bits are sufficient to represent one ASCII value in binary, and that one bit of an eight-bit byte is left over for use as a cue to the computer that a key has been pressed.

Depending on the ASCII code used, characters can be displayed three different ways—inverse, flashing, and normal upper case. Normal text characters are made up of white dots on a black background.

When you're working from the Monitor, ASCII values can be expressed in hexadecimal; when in Applesoft, they can be represented in decimal. A text character, the letter A, for instance, can be produced by poking any one of its three ASCII equivalents, 1, 65, or 193 from Basic, or by entering its hexadecimal equivalents from the Monitor. Each of the three ASCII values will give you A, but each in a slightly different form.

There's a chart on page 15 of the *Apple II Reference Manual* that shows the ASCII values for all the characters you can put on the screen—in normal, flashing, and inverse modes. The chart supplies values in both decimal and hex notation. To get, for example, the ASCII equivalent, in hex, of a flashing G, first locate the character on the chart. Look up to the hex number at the top of the column in which you find the character (in this case $40); now look to the left and find the hex number at the beginning of the row ($7). Add these two numbers and you'll get the appropriate ASCII value ($47).

Text mode uses two different areas of memory. The first, called the *primary screen*, occupies memory locations $400 (1024) to $7FF (2044). The second area, known as the *secondary screen*, occupies $800 (2048) to $BFF (3071). Most of what you do in text mode will rely only on the primary screen area. As mentioned last time, the secondary screen overlaps with the area of memory that's used to store the Apple's Basic programs, so it's hard to use it from Applesoft.

Let's get our feet wet by writing a program that demonstrates how memory is used to represent text. Since we know that screen memory for the primary screen runs from 1024 to 2047, it should be possible to poke things into memory and have them appear on the monitor screen without ever using print statements.

For starters, try typing in and running the following Applesoft program:

```
10  HOME
20  FOR I = 1024 TO 2047
30  POKE I, 193 :REM THE LETTER "A"
40  FOR J = 1 to 30 :REM LET'S SLOW THINGS
      DOWN
50  NEXT J
60  NEXT I
70  CALL 65338 :REM BEEP THE SPEAKER
80  GOTO 80
```

When you run this program, you should see the letter A filling the screen, working from left to right and top to bottom. The screen will appear to be broken into three pieces that are being filled with As simultaneously. Lines of As begin to form in three different screen areas, and subsequent lines of As begin underneath each of the first three lines until the screen is filled.

Try replacing the number 30 in line 40 of the program with some other number. The higher your replacement number is, the slower the screen will fill. Modifying the program to poke other values can also be fun. For instance, making A equal to 1 in statement 30, rather than to 193, should give you an inverse A.

Now let's examine a few peculiarities about how the screen fills. Why, for instance, does it appear to be broken into three separate pieces?

The best answer we've received is that this effect is tied somehow to the hardware design of the Apple and has to do with the scan rate on standard television sets. Whatever its cause, this oddity has definitely made life more difficult for programmers. Instead of being a simple matter, locating successive lines in memory requires a complex algorithm or a table.

Did you notice that each time a line is drawn on the bottom third of the screen there's a pause before any more As appear? The pause happens because, for no reason we've been able to discover, eight bytes of memory are wasted after each group of three lines has been displayed on the screen.

Making matters even worse, Apple has caught on to this so-called useless memory and has put it to use within DOS. The eight bytes are used by DOS to remember what disk drive was accessed most recently. You'll notice that your next disk access after running the program we just worked with will cause recalibration to occur.

Page 16 of the *Apple II Reference Manual* contains a chart you can use to reference any byte of memory that's part of the text screen. When experimenting, remember that if the screen should scroll after a poke, you'll lose what you just poked into memory. For instance, if you enter "POKE 1024, 193" from the prompt], you might expect to see an A in the upper lefthand corner of the screen. But if pressing return causes the screen to scroll, you'll lose your A.

Let's not delve any further into text mode at this point, since it's used only occasionally in games. We'll return to text mode in later installments of this series when we learn how to implement a scoreboard using mixed-mode graphics.

Lo-Res Graphics. Because of the blocky looking graphics they produce, lo-res graphics are not as popular as they once were. In general, lo-res graphics are put to best use in situations that have special requirements, such as when you want to use the exta colors they afford, or when you need to take advantage of the lower memory requirement and don't mind working with a forty by forty-eight display. For a list of the colors available in lo-res graphics, see Table 8 on page 17 of the *Apple II Reference Manual.*

Lo-res graphics make use of the same memory area ($400 to $7FF) as text graphics do. We can even use the program we wrote earlier as a jumping off point for leaning about lo-res graphics.

Go back to your program and add a Line 5 which says

"5 GR." Now, run the program again. You'll see each character (where the letter A was before) appear as a magenta box overlying a light green box. The screen should fill in exactly the same manner as the text screen did.

If you get a split screen that has text at the bottom, you'll actually see the As appear. To view this example as a full screen of lo-res graphics (no text window), simply turn on Switch two by adding a Line 7 (X = PEEK 49234) to the Applesoft program.

The decimal value 193 we poked into memory is $C1 in hexadecimal. Remember from last time that every location in memory contains one hexadecimal byte, and that every byte can be split into two nibbles. In lo-res graphics, each nibble in a byte corresponds to one colored block on the monitor screen. The nibble on the left corresponds to the bottom block in a character position, the right nibble to the top block.

One of the main drawbacks of working in lo-res is the confusion that can result from trying to deal with two colored blocks at once. Remember, there's no way to poke or peek just a nibble. The smallest amount of memory an Apple can handle at one time is a byte.

Hi-Res Graphics. A great deal of the commercial software written today relies on hi-res graphics. Games, plotting packages, and even some word processors use the hi-res screen for all video output. Virtually everything that follows in this series of articles will deal with different methods of writing to the hi-res screeen.

As do the text and lo-res graphics modes, hi-res graphics uses memory mapped output. Hi-res graphics memory is also divided into primary and secondary screen pages. Considerably more memory (16K bytes in all) is required to support hi-res graphics than is needed for lo-res or text. Each hi-res screen page contains 8,192 memory locations.

In the hi-res graphics mode, the screen has two hundred eighty dots horizontally and one hundred ninety-two dots vertically. One of the advantages of hi-res graphics over lo-res is that you have complete control over any dot. One limitation of hi-res, as we'll see later on, is that although there are six hi-res colors, a given dot can be only one of four of these colors, depending on the dot's location on the screen.

Let's create a simple program to demonstrate how the 8K of memory for the primary hi-res screen, located between $2000 (8192) and $3FFF (16383), is laid out. (In order to understand fully what this program does, you may wish to refer back to September's column, in which the algorithm used for computing the decimal value of the bits in a byte was explained.)

```
10    HGR :REM TURN ON HI-RES
      GRAPHICS
20    X = PEEK (49234): REM SWITCH TWO
      ON
30    FOR I = 8192 TO 16383: REM START AND
      END OF PAGE ONE OF HI-RES
40    FOR J = 1 TO 8: REM BIT # IN BYTE
50    GOSUB 1000:REM CONVERTS BIT # TO
      BINARY VALUE
60    POKE I,X
70    FOR L = 1 to 30:REM SLOW IT DOWN
80    NEXT L
90    NEXT J
100   NEXT I
110   CALL 65338:REM GO BEEP
120   GOTO 120
1000  REM
1010  REM CONVERT BIT # IN J TO VALUE IN
      X
1020  REM
1030  X = 2^(8-J):REM
1040  RETURN
```

The subroutine from lines 1000–1040 will return the decimal value of a byte with only the bit in J turned on. For instance, if you call it with J = 1, it should return a 128.

This program, when run, will poke all the bytes in the 8K that's used to produce hi-res graphics from the primary screen page. It will poke each byte eight times, once for every bit in the byte. The result is that each bit in a given byte will turn on in sucession.

Now run the program. If it seems to behave in an unpredictable—bordering on bizarre—manner, then it probably worked just right.

Watch the One on the Left. You should have seen a dot appear approximately one-half inch into the screen. The dot should then have moved to the left. Just as the dot bumped into the left border of your video screen, a new dot should have appeared, moved to the left a bit and stopped, and then another new dot should have appeared. This process should have repeated until the screen was full of dots.

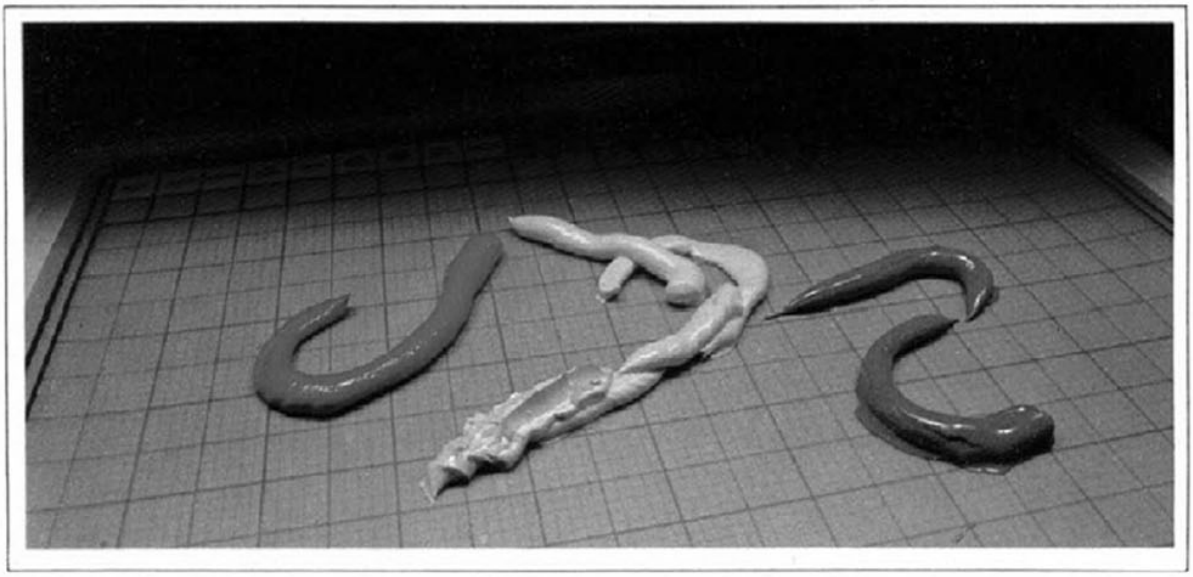
Your first reaction when all this happened may have been to think that the equation in Line 1030 of our program works backward. It doesn't. The Apple actually uses the bits it finds in a byte in reverse of the way you'd expect (remember our brief discussion last time?).

If you think all this is wild, try slowing down the screen fill process by replacing the 30 in Line 70 with a higher numerical value. If you substitute a high enough value, you should be able to count the dots as they are drawn on the screen. If you count only seven dots per byte, you have verified another of the Apple's idiosyncracies. The leftmost bit in each byte is not drawn to the screen. It has another function, which we'll talk about next time.

You must also have observed that once again the screen appeared to have been cut into thirds. Once again, you're seeing that lines are drawn from various positions on the screen, as well as noticing the pause that results from the fact that eight bytes of memory are "wasted" after every third line is drawn on the hi-res screen.

A chart on page 21 of the *Apple II Reference Manual* shows the memory address for every dot on the hi-res screen. In the next issue, we'll learn more about this area of memory. We'll also discuss hi-res color and outline some techniques that will make dealing with the hi-res screen less confusing.

Apple II Graphics: Peculiarities of the Hi-Res Screen



Welcome to part three of our series on Apple II graphics. In previous installments we learned about working from the Apple's Monitor, using the hexadecimal numbering system, and memory mapping. In addition, we discussed what a screen mode is, looked briefly at each of the possible screen modes, and explored how the soft switches work to tell the Apple which screen mode to use.

This time we will begin exploring your Apple's hi-res screen, making use of the knowledge acquired during prior lessons. If you missed the two previous issues of *Softline* you may wish to see about obtaining copies of the earlier articles for reference.

An Experiment. Enter the following program:

```
10  HGR
15  HCOLOR = 2
20  FOR I = 0 TO 100
30  HPLOT I,0
40  NEXT
50  REM
60  REM NOW LET'S TURN THEM OFF
70  REM
75  HCOLOR = 0
80  FOR I = 0 TO 100 STEP 2
90  HPLOT I,0
100  NEXT
```

After examining this program, you're likely to conclude that what it will do when run is draw a blue horizontal line consisting of one hundred dots and then turn off every other dot. Based on the program listing, this is certainly what you'd expect to see. Now run the program.

Surprise! You should have seen the line draw and then erase completely. This occurs because of the way the memory map inside your Apple is organized. Let's digress for a moment to account for this unexpected arrangement.

In hi-res mode your Apple is able to display six colors: black, white, orange, blue, green and violet. In order to allow for any of the 280 by 192 (or 53,760) possible dots on the hi-res screen to be any of the possible six colors, mapping the hi-res screen would require at least 20,160 bytes of memory.

To see how this number was arrived at, recall from the first article in this series our discussions of binary numbers. As you know, the Apple's memory is really just a big collection of on/off switches. If you have three on/off switches, they can be set in two to the third (eight) possible combinations.

For purposes of convenience, your Apple's memory has been broken into 65,536 separate sets of eight on/off switches (bits). Each of these groupings is known as a byte. Since each byte contains eight on/off switches, a byte can have two to the eighth possible settings (two hundred fifty six in all). Representing all six hi-res colors would call for

at least three bits (on/off switches). Multiplying three times 53,760—the number of dots on the hi-res screen—gives us 161,280—the number of bits it would take to represent all dots. If we then divide by eight to convert to bytes we'll get my estimate of 20,160.

If this number were the true amount of memory used by hi-res graphics, programming on the Apple would be limited at best. For you see, only 32,000 of the 65,536 bytes are available after Applesoft, the Monitor, and DOS take their chunks of memory. Subtract another 20,160 and no room would be left for meaningful programs.

Luckily, the Apple has developed a technique for displaying 280 by 192 dots using only 8,192 bytes of memory. This method of looking at memory may seem confusing at first, but it really isn't bad once you get used to it. Certainly, it's preferable to the alternative—having graphics but no memory for programming.

Briefly stated: If any two horizontally adjacent dots are "on," they will appear to be white. If a dot is "on," is surrounded by two "off" dots, and is on an even x-coordinate, it will be either violet or blue. If a dot is "on," is surrounded by two "off" dots, and is on an odd x-coordinate, it will appear green or orange. Any two "off" dots in a row will appear as black.

After studying the statements above, you will be able to deduce the following rules:

1. Any dot falling on an even x-coordinate must be black, violet, blue or white.

2. Any dot falling on an odd x-coordinate must be black, green, orange or white.

We never talked about how the determination is made of whether a dot on an even x-coordinate will be violet or blue (or about how it's decided whether a dot falling on an odd x-coordinate will be green or orange). We'll look at these things now.

Understanding the Memory Map. In your Apple's memory, each byte within the area mapped onto the hi-res screen contains eight bits. Seven of these bits correspond to dots that appear on your monitor or television screen; the bit that remains specifies what color lonely "on" dots are to be. Let's look at another example.

Enter and run the following program:

```
10  HGR
20  HCOLOR = 3
30  HPLOT 0,0 TO 0,100
```

Running this program should have drawn a vertical white line (according to the Applesoft manual hcolor = 3 should be color white1). Instead you were greeted by a vertical green line. Now add the following lines to the program and run it again.

```
35  HCOLOR = 4
40  FOR I = 0 TO 100
50  HPLOT 1,I
60  NEXT
```

Lines 35 through 60 simply set the color to black2 and draw a vertical black line at x-coordinate 1. Since this col-

umn was black already, it would seem that the program should have run the same way with or without lines 35 through 60. But watch closely; you'll see that a vertical green line is drawn and then slowly, starting at the top and proceeding to the bottom, becomes orange.

This occurs because of that extra undisplayed bit in each byte. Notice that column 0 and column 1 of the hi-res screen are in the same byte in memory. When we plot black2 in a byte it turns on the color bit, causing any other displayed bits in the byte to switch colors. If line 35 in the program above had said hcolor = 0 then the original green line would have stayed green; no color change would have occurred.

Decimal	Hex	Binary
		7 6 5 4 3 2 1 0
1	$ 1	0 0 0 0 0 0 0 1
2	2	0 0 0 0 0 0 1 0
4	4	0 0 0 0 0 1 0 0
8	8	0 0 0 0 1 0 0 0
16	10	0 0 0 1 0 0 0 0
32	20	0 0 1 0 0 0 0 0
64	40	0 1 0 0 0 0 0 0
129	81	1 0 0 0 0 0 0 1
130	82	1 0 0 0 0 0 1 0
132	84	1 0 0 0 0 1 0 0
136	88	1 0 0 0 1 0 0 0
144	90	1 0 0 1 0 0 0 0
160	A0	1 0 1 0 0 0 0 0
192	C0	1 1 0 0 0 0 0 0

Backward Bits. The chart shows the decimal, hexadecimal and binary representations of bytes with one bit in each of the right seven bits turned on. Now watch what happens when these values are poked into the hi-res screen. From Applesoft enter the following lines without line numbers:

```
HGR
POKE 8192,1
```

8192 happens to be the machine address of the first byte of the memory mapped area for hi-res graphics. When you enter the above statements, a single dot will appear in the upper left-hand corner of the screen. This dot will be in the same place as if you had entered:

```
HGR
HCOLOR=3
HPLOT 0,0
```

What's peculiar is that you poked a one into memory location 8192 (00000001). One is the rightmost bit in the byte and yet you turned on the leftmost dot. Now let's poke 8192 with a two (00000010):

```
POKE 8192,2
```

Note that even though we poked a byte with the second bit from the right on, it turned on the screen dot that's second from the left. Also note that the dot appears to be green. This is because the leftmost bit in the byte (the color bit) is off and the dot turned on now appears on x-coordinate 1. To turn the dot orange, all we have to do is poke 130 (10000010) into memory, which has the same dot turned on but also has the color bit on.

To summarize: Within each byte are eight bits, the rightmost seven of which are used to light a dot on the screen. The leftmost bit (bit seven, remembering that bits are numbered from zero to seven) controls the color of a solitary bit, with any two bits on in a row representing white.

Left to Right Once Again. Enter the following program:

```
10  HGR
20  HOME
30  VTAB 22
40  PRINT "THIS IS POKING"
50  FOR I = 8192 T0 8231
60  POKE I,1
70  POKE I,2
80  POKE I,4
90  POKE I,8
100  POKE I,16
110  POKE I,32
120  POKE I,64
130  POKE I,0
140  NEXT
150  HOME
160  VTAB 22
170  PRINT "THIS IS HPLOTTING"
180  FOR I = 0 TO 279
190  HCOLOR = 3
200  HPLOT I,0
210  HCOLOR = 0
220  HPLOT I,0
230  NEXT
```

Lines 50 to 140 of this program represent a loop varying I from 8192 (the first byte of the hi-res screen) through 8231 (the end of the first line of the hi-res screen). Within the loop are eight poke statements that poke into memory bytes with bits on in each screen dot position. Lines 180 through 230 simply hplot and then remove a dot from each possible x-coordinate.

When you run this program, you should see a dot move rapidly from the left side of the screen to the right and then begin again on the left—making the same trip as before, but more slowly. What we've begun to experiment with here is *byte move animation*, a technique we'll devote a great deal of time time to later on. As you can see, much greater speed is possible with byte move animation than can be achieved through hplot statements. It's a variation of this technique that makes possible games like *Raster Blaster* and *Threshold*.

Where's the First Line? By now you should all be fairly comfortable with poking things into line 0 of the hi-res screen. If you think back to the November issue, you'll recall that hi-res screen memory is not contiguous. There's no easy way we know of to calculate the addresses of successive lines.

on the hi-res screen. Therefore, most authors we know of use tables to point to the first byte of each screen line. For example:

```
10  HGR
15  DIM A%(23), B%(7), C%(191)
17  HOME: VTAB 22: PRINT "LOAD TABLES"
20  FOR I = 0 TO 23
30  READ A%(I)
40  NEXT
50  FOR I = 0 TO 7
60  READ B%(I)
70  NEXT
80  REM NOW COMPUTE ADDRESSES
90  FOR I = 0 TO 191
100   W = INT(I/8)
110   C%(I) = A%(W) + B%(I - (8*W))
120   NEXT
200   REM
210   REM NOW DRAW A LINE
220   REM
230   FOR I = 0 TO 191
240   POKE C%(I),1
250   NEXT
300   REM
310   REM DO IT THE OLD WAY
320   REM
322   HGR
323   HCOLOR = 3
325   HOME: VTAB 22: PRINT "HPLOTTING"
330   FOR I = 0 TO 191
340   HPLOT 0,I
350   NEXT
1000  DATA 8192,8320,8448,8576,8704,8832,8960,
      9088,8232,8360,8488,8616,8744,8872,9000,9128,
      8272,8400,8528,8656,8784,8912,9040,9168
1010  DATA 0,1024,2048,3072,4096,5120,6144,7168.
```

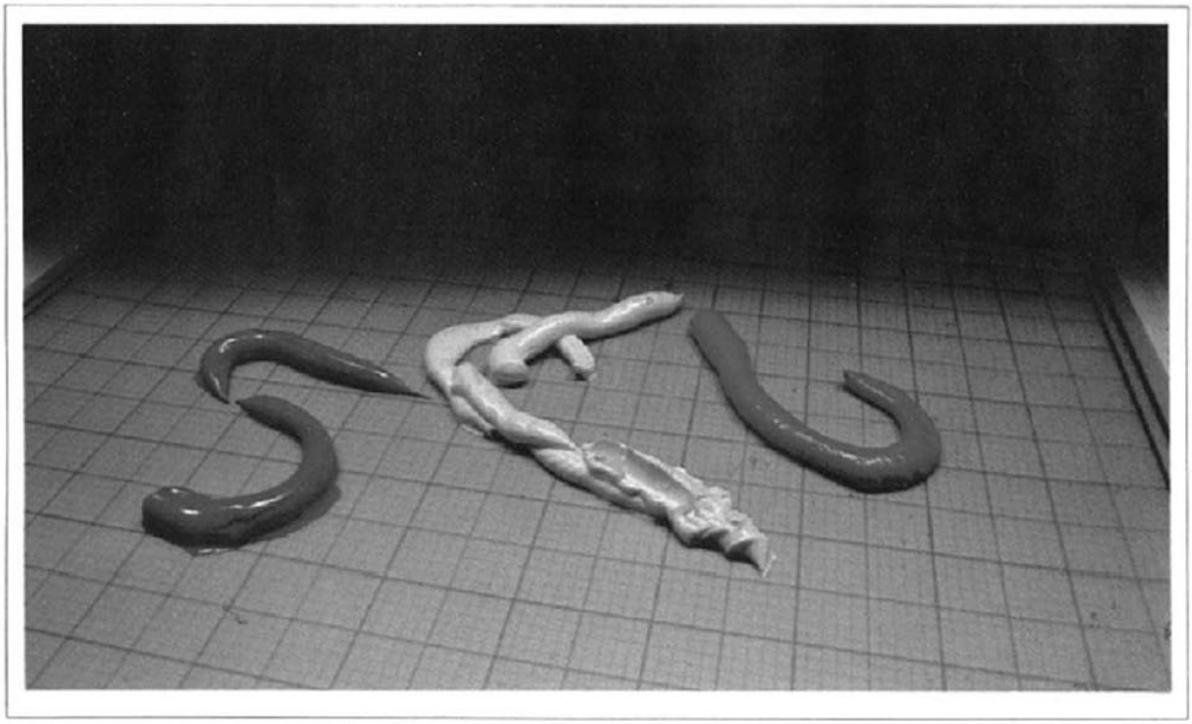
Lines 20 through 70 load the screen line addresses that are found on page 21 of the *Apple II Reference Manual* into the tables A% and B%. Lines 80 through 120 compute the addresses of all 192 lines on the hi-res screen. Lines 230 through 250 poke a dot into the first byte of each screen line and lines 330 through 350 hplot a dot.

When you run this program, you'll notice that poking seems to be no faster than hplotting. In fact, it's substantially slower than hplotting because of the overhead incurred by loading tables. Don't worry; in discussions to come we'll return to this example and show how to use these tables to our advantage.

And in Conclusion. Next time we'll delve even deeper into the wonders of hi-res graphics on the Apple. In particular we'll look more closely at how color is produced, as well as at how products that claim to produce more than the standard six Apple hi-res colors work.

Apple II Graphics:

Color Me Blue . . . or green . . . or purple



Welcome to part four of our series on Apple II graphics. In this installment, we shall concentrate on how color is produced on the hi-res screen. In the process, we hope to shatter some common myths. For instance, there's the one about how many dots exist on a row on the hi-res screen. Did you say 280? Before you finish this article, you may agree that 40, 140, 280, and 560 are all legitimate answers.

We'll be presuming that you have a thorough grasp of the material covered in prior issues, especially hexadecimal numbering and the use of the Monitor. It will also be helpful if you have copies of the prior installments of this series, the *Apple II Reference Manual,* and the *Applesoft Reference Manual,* near at hand.

The overall appearance of an image produced by your Apple is greatly affected by the resolution in which it is drawn. A circle can be drawn nicer, for instance, on a 100 by 100 dot matrix than on one that's 2 by 2. The more dots you have to play with, the nicer the image you can produce.

Along the y axis of your hi-res screen, you always have 192 dots to use. This number does not change whether you are in black and white or in color.

Unfortunately, the x axis is nowhere near so simple. When you're working in black and white, 280 dots can be produced on a row, but only 140 colored dots can be seen. Add to this the limitations imposed by not being able to place an orange dot next to a green dot (in general) and you've opened up a real can of worms.

For convenience in developing animation routines for the Apple, it helps to think of the hi-res screen as being in one of the modes we'll talk about now. These are not hardware-defined modes, nor will you find any soft switches to turn on and off. Rather, they are ways to structure your

thinking that will, we hope, simplify the coding you must do when you program your game.

560 Dots. Let's do an experiment. First, turn off the color on your monitor (or television). Now go into the Monitor (*call −151*); you'll see the asterisk prompt. Turn on hi-res graphics (enter C050 and press return, then C057 and press return).

Now clear the screen by entering 2000:0, then 2001<2000.3FFEM. (If all this seems confusing to you, rereading the first two installments of this series should help.)

Next, turn to page 21 of the *Apple II Reference Manual*. Compute the memory address of the first byte of each screen line for the first fourteen screen lines. The results of your figuring should give you the following list:

$2000
$2400
$2800
$2C00
$3000
$3400
$3800
$3C00
$2080
$2480
$2880
$2C80
$3080
$3480

We want to *poke* the hex values 01, 81, 02, 82, 04, 84, 08, 88, 10, 90, 20, A0, 40, C0 into the bytes we have just identified. To do this, enter 2000:01, then 2400:81, then 2800:02, and so on into the Monitor until you have entered 3480:C0. As you *poke* values into memory, you should see a diagonal line appear.

Wait a second. We just *poked* a green dot (01) into line 0 of the hi-res screen and a blue dot (81) into line 1 of the hi-res screen with the color turned off. Isn't it reasonable to expect that the result of our activity would be two dots exactly on top of each other? Why isn't this what happens?

Let's answer these questions with actions rather than words. Try *poking* the following values into the same addresses: 01, 01, 02, 02, 04, 04, 10, 10, 20, 20, 40, 40. These values are the same as the ones we entered a moment ago, except that every other value does not have the high bit turned on.

Now we get what we expected—seven sets of two dots on top of each other. Notice that the diagonal line that results this time is nowhere near as smooth as the one we got before; this is because we are dealing now in only 280-dot resolution rather than in 560.

We've just demonstrated that the Apple has the potential to turn on 560 dots on any screen line. Unfortunately, Apple's hardware designers used only forty bytes to represent these 560 dots. Each byte can turn on or off seven dots; which set of seven out of the possible fourteen from each byte is determined by whether or not the high bit is set.

You're probably asking yourself what good it is to have 560 dots, only half of which can be lit from any one byte. In a later article we'll be exploring some tricks to take advantage of this peculiarity. It's possible to create eighty-column screens with beautiful character sets using this method, with half the dots on hi-res screen 1 and the other half on hi-res screen 2. This method also makes possible the production of very smooth graphs.

280 Dots. The vast majority of the game programs written today for the Apple work under the presumption that the hi-res screen has 280 dots on any given horizontal line. With rare exceptions, this is not the best possible mode to think in. Consider, for instance, the fate of a green monster drawn on an odd x coordinate.

Recall from last time that a green dot can never be drawn on an odd x coordinate. Therefore, if we were to draw a monster along these coordinates and then specify green as he color we want, no monster would appear. There are only 140 possible x coordinates that a green monster could se drawn on: the even ones.

Let's try another example, this time in Applesoft.

From the Applesoft prompt, enter *hgr*. If the prompt haracter goes away, press return until it reappears at the ottom of the screen. Enter *hcolor=1*. This sets the current olor to green. Now draw a vertical green line by entering *plot 0,0 to 0,20*.

Nothing happens. There is absolutely no change. How can this be? How can Apple claim 280-dot resolution when only half of the dots can be used?

The simple truth is that you have 280 dots to play with only if you are working in black and white. What we've just discovered in attempting to draw a vertical green line by *hplotting* 0,0 to 0,20 is that if we are working in green we can draw on only half of the x coordinates. The same is true for violet, orange, and blue; we can draw on only half of the x coordinates. It can also be said that no more than 140 white dots can exist, for it takes at least two dots in a row to form white.

Most gamemakers either think of the hi-res screen as having 280 x coordinates animate either in black and white or, working in color, they always add two to the x coordinate when moving an object (thus simulating 140 mode). These methods come up short for many applications. When printing text on the screen, such as for a scoreboard, for example, you'll get much better resolution by using the 280 or 560 dot modes.

140 Mode. Generating a colored dot on your Apple requires the use of two memory bits. Since forty bytes are allocated to represent each line and seven bits per byte are used to represent screen dots, we compute that 280 bits are in use for screen mapping (40 × 7). Given that two bits are used to represent a colored dot (one on and one off for blue, orange, green, and violet, both on for white, and both off for black), it becomes convenient to think of each line of the hi-res screen as containing 140 colored dots.

Unfortunately, this method of looking at the hi-res screen breaks down in most cases where two different colored dots are placed side by side. For instance, when an orange dot is followed by a blue dot, you get a white dot. There's no way of avoiding this; it's a consequence of the general rule governing Apple graphics that says any two dots on in a row will always appear white.

Here, for your your convenience, is a summary of the effect of contiguous dots on the hi-res screen:

if a dot is	and is followed by	then you will get
violet	violet	violet
violet	green	violet and green
violet	white	violet and white
green	violet	white
green	green	green
green	white	white
white	violet	white
white	green	white and green
white	white	white
blue	blue	blue
blue	orange	blue and orange
blue	white	blue and white
orange	blue	white
orange	orange	orange
orange	white	white
white	blue	white
white	orange	white and orange
white	white	white

What should be reinforced by this is that when you're dealing with hi-res graphics on the Apple II, there's just no way to win except to spend literally weeks thinking about and planning for the graphic effect you wish to achieve before you jump into code.

Probably the best argument in favor of using 140-dot mode holds true only for machine language programmers: in 140, your x and y coordinates fit in one byte. Even Basic programmers stand to benefit from this; some of the machine language routines we'll be using later on in this series depend on 140 mode for their speed.

40 Mode. Perhaps it seems to you that the safest way to think of the hi-res screen is to pretend it has only forty dots, each of them one byte wide. To explore this idea further, let's try another experiment.

Enter the Monitor with the command *call −151*. Turn on hi-res graphics with the commands C050 and C057. Enter 200:AA. Then enter 2001<2000.3FFEM. This will set the enter screen to a byte of orange followed by a byte of blue. You'll probably notice that a black line separates the orange and blue columns. Once again, this happens because we have turned off two bits in a row.

The point here is that even with one byte per dot there just isn't an easy way to predict what you will get when plotting on the hi-res screen.

How Many Colors? Now that we've all become totally confused about the number of columns on the hi-res screen, let's take a quick stab at determining the number of colors the hi-res screen can display.

According to page 89 of the *Applesoft Reference Manual*, the Apple is capable of displaying eight colors (black1, green, blue, white1, black2, violet, orange, and white2). Since the two blacks and the two whites are indistinguishable, we are left with only six colors.

But what about all those games and graphics packages that claim to produce twenty-one or even one hundred colors? Let's do another experiment.

Enter the following Applesoft program:

```
10    HGR
15    SZ = 15
20    FOR C1 = 0 TO 7
30    FOR C2 = 0 TO 7
40    GOSUB 1000
50    NEXT, C2,C1
99    END
1000  FOR Y = 0 TO SZ STEP 2
1010  FOR X = 0 TO SZ
1020  HCOLOR = C1
1030  HPLOT X,Y
1040  HCOLOR = C2
1050  HPLOT X, Y+1
1055  NEXT X,Y
1099  RETURN
```

When you run this program, you should see dozens of colored squares drawn in the upper left-hand corner of the screen. To increase the size of the squares, change line 15 from SZ = 15 to SZ = 30. You'll find that the larger the value for SZ, the larger the square.

All we're doing here is displaying squares comprised of alternating colored lines. However, you should see, for instance, that when green and white are alternated, the color of the resulting square could be termed lime-green. This blurring of colors when they are placed next to each other is what makes it possible for companies to produce more than the standard six colors.

A variation on this theme is a checkerboarding effect used in several games that are on the market now. For example, dark orange may be produced by checkerboarding (in 140 mode) orange dots and black dots. Other more complex schemes may mix alternate horizontal lines of green with lines of alternating green and black. The combinations are endless.

In general, no matter how you look at it, the Apple II can generate only six colors. When different colors are situated close to each other, they will seem to blend and produce a new color. This approach has been used successfully in many games; we'll come back to it later in this series.

What's Ahead. By now you probably have an idea of the work and thought that's involved in planning for color before you can begin programming a game. It's not uncommon to spend weeks working on color design and coordination before beginning to program.

In the lessons ahead, we'll be using routines that require you to think in 140, 280, and 560 modes. Next time, we'll look at shape tables and begin to look at animation. Specifically, we'll be talking about such things as flicker and speed.

Apple II Graphics:

Mooving into Animation

Welcome to part five of our series on graphics for the Apple II computer. In this article, we'll examine animation techniques. To do this, we'll use Applesoft and shape tables.

We won't get into much detail here about how to create shape tables since many people already own shape table generation programs and because using shape tables is an extremely poor method of animation. So before we get started, read the section on shape tables (Chapter 9, especially pages 92-96) in your *Applesoft Reference Manual*. In future articles, we'll learn about much faster methods.

Before you can begin animating, you need something to animate. From Applesoft enter the following sequence of commands:

```
CALL -151
4000:04 01 0A 00 00 00 00 00
4008:00 00 92 92 92 09 0D 4D
4010:49 09 DD 4D 49 D8 DB DB
4018:DF DB DB FB 48 0D 0D 0D
4020:0D 0D 4D 49 09 D8 DB DB
4028:1F 1F 1F 1F 1F 1F 48 0D
4030:0D 0D 0D 0D 4D 49 09 D8
4038:1B 1F 1F 1F 1F 1F 1F 1F
4040:1F 68 0D 0D 0D 0D 0D 0D
4048:0D 0D 0D 18 FF 1F FF DB
4050:DB DB 1B 48 49 49 49 49
4058:0D 0D 0D D8 DF FB DB DB
4060:DB DB 00
3D0G
BSAVE HIRES COW,A$4000,L$63
```

In order to prove that we successfully created our shape let's try to draw it.

To do this enter the following commands into Applesoft from the] prompt:

```
BLOAD HIRES COW
POKE 232,0
POKE 233,64
HGR (If you don't see your cursor, press return
  until it appears)
SCALE = 1
HCOLOR = 7
DRAW 1 AT 9,10
```

A hi-res cow should have appeared in the upper left corner of the screen. This particular cow was borrowed from the popular arcade game *Crop Duster* (Slipshod Software). We'll use this cow as our object for practicing animation. In future articles, we'll also look at how the tremendous 3-D effect given to the female chickens in this game was produced.

Before continuing, let's look at how we were able to draw our cow on the hi-res screen. Locations 232 and 233 ($E8 and $E9) contain the address of the shape table used by Applesoft's *draw* and *xdraw* commands. We *poked* the shape for our hi-res cow into memory starting at $4000.

The Apple typically stores addresses in memory with the bytes reversed, such that $4000 would be stored as $00 followed by $40. *Poke 232,0* stores the low byte and *poke 232,64* stores the high byte (remember that hexadecimal $40 is a decimal 64). We then turn on hi-res graphics *(hgr)*. Now we tell Applesoft to print our shape at size 1, meaning the same size at which it was created *(scale = 1)*. Next we must tell Applesoft what color to draw our shape in *(hcolor = 7)*. Lastly, we actually draw the shape *(draw 1 at 9,10)*.

When Does White = Orange? According to page 89 of your Applesoft manual our statement *(hcolor = 7)* sets the color of things drawn to white2. However, if you review the last article in this series, you'll recall that white can only occur when two dots in a row are turned on. A shape table definition simply identifies which dots are to be set to the current color and what the high bit of each byte in which a dot is affected should be set to. For instance, as long as our hi-res cow is still on the screen let's try removing it:

```
HCOLOR = 0
DRAW 1 AT 9,10
```

should erase the cow. To draw it in blue rather than orange we enter:

```
HCOLOR = 7
DRAW 1 AT 10,10
```

Making the Cow Move—Slowly. To do some animation, enter the following commands. These commands presume you still have a blue hi-res cow on the screen.

```
HCOLOR = 0
DRAW 1 AT 10,10
HCOLOR = 7
DRAW 1 AT 12,10
HCOLOR = 0
DRAW 1 AT 12,10
HCOLOR = 7
DRAW 1 AT 14,10
HCOLOR = 0
DRAW 1 AT 14,10
HCOLOR = 7
DRAW 1 AT 16,10
```

There's Gotta Be a Better Way. In case you didn't notice, this form of animation is a bit cumbersome. Did you wonder why the x coordinate was incremented by two each time we drew the cow rather than by 1? The reason is that it had to be done this way in order to stick to a blue cow. Any cow drawn on an odd x coordinate would have been orange.

Now let's try it like this:

```
NEW
10 HGR
20 POKE 232,0
30 POKE 233,64
35 SCALE = 1
40 X=0
45 HCOLOR = 7
50 DRAW 1 AT X,10
60 HCOLOR = 0
70 DRAW 1 AT X,10
80 X=X+2
90 IF X > 255 THEN GOTO 40
100 GOTO 45
```

Now run the program (you'll have to press reset to stop it).

If you'd prefer an orange cow, simply *init* x to 1 on line 40 and run the program again. Or, to see a blue and orange cow, change line 80 to read X= X + 1. The cow appears to be flickering.

A large portion of the time involved in writing any animated program is spent trying to decrease flicker. One way to decrease flicker in this program is to lengthen the time your cow is on the screen in relation to the time it is gone. For instance run the program with a delay loop at line 55:

```
55 (FOR J = 0 TO 40:NEXT J).
```

The longer the delay, the smoother the motion. In an actual game, rather than using a delay loop, this is when you'd do your calculations of where to next move your cow.

Getting Exclusive. Let's say you're working on a game that requires your hi-res cow to pass in front of a tree. Think for a moment about what would happen if you did your animation the way we did it earlier.

First you would draw a cow in front of the tree. Then you would change the color to black and try to undraw the cow. Oops! You now have a black cow standing in front of your tree. What you really wanted to have happen was for the background to be restored to what it looked like prior to the drawing of the cow.

Unfortunately, we're not aware of any set of animation routines currently on the market that recalls what was on the screen prior to drawing an object.

The challenge, then, becomes finding an acceptable alternative that allows you to draw and then undraw things regardless of what's already on the screen. A trick of the hardware allows us to do just this—reliably. It's called *exclusive or*.

A Little Boolean Math. Sounds scary huh? Don't worry; it's not so bad. Study the chart below:

		1	0
1	:	0	1
0	:	1	0

This chart tells us the result of using exclusive or (EOR) on any binary number with another binary number. For instance it says 1 EORed with 1 results in 0. 1 EOR 0 = 1. 0 EOR 1 = 1. And 0 EOR 0 = 0.

You're probably wondering what this has to do with computer graphics. It turns out that this technique allows us to draw our hi-res cow in front of trees, clouds, other cows, and anything else we choose. A simpler way of looking at the table is to notice that if we think of the 0 and 1 on the left of the chart as being dots to be plotted from our shape, and of the 0 and 1 on the top of the chart as dots on the screen, an amazing thing happens. For any 1 dot in our shape, the screen dot is inverted. For any 0 dot, the screen dot is left alone.

Using exclusive or in putting a shape onto the hi-res screen causes any dot that would normally be drawn to be inverted instead. Dots that would not normally be affected are left untampered with. It turns out that Apple caught on to this effect and provided us a simple command to exclusive or with. It's called *xdraw*. Let's experiment with the following (presuming you still have your hi-res cow in memory):

```
HGR (press return until you see your cursor again)
HCOLOR = 7
XDRAW 1 AT 9,10
XDRAW 1 AT 9,10
```

You should have seen the cow appear, then disappear. This is because the first *xdraw* caused the dots in your black screen to invert producing the cow. The second *xdraw* also caused inversion, but only of the dots comprising our cow. Of course, inverting an on dot produces an off dot.

Let's see what would happen if our hi-res cow were to stand in front of a cloud.

```
HGR
HCOLOR = 7
FOR X = 0 TO 100:FOR Y =0 TO 100:HPLOT X,Y:
  NEXT:NEXT
XDRAW 1 AT 9,10
```

Aha! We have a hi-res cow on a white background, just as we expected. A cow half on and off a cloud will appear to be two different colors (try it!). However, most games are set up so this is the exception rather than the rule. In general, if your cow is running and happens to pass in front of a cloud or tree, the slight color distortion that results is highly preferable to losing your background or your cow.

Now that we have our cow moving, let's put him under the control of a joystick (or paddles if that's what you have).

```
NEW
10 HGR
20 HCOLOR =7
30 X=PDL(0):Y=PDL(1)
40 IF Y > 140 THEN Y=140 :REM HANDLE OFF BOTTOM OF
    SCREEN
50 XDRAW 1 AT X,Y
60 NX=PDL(0):NY=PDL(1) :REM READING PADDLES
    PROVIDES OUR DELAY LOOP
70 XDRAW 1 AT X,Y
80 X=NX:Y=NY
90 GOTO 40
```

Now run the program.

Two things would greatly improve this program: first, only drawing the shape on even or odd x coordinates (to make the color of our cow consistent); and second, not erasing the cow if its position is unchanged. Experiment with this program. It's a good place to start trying to cure flicker.

Wrapping Up. Our next lesson will begin to address *byte-move animation* and how to create byte shapes. This is the technique typically used by professional programmers for high-speed animation. In lessons ahead, we'll look at new techniques for decreasing flicker and discuss such topics as collision detection.

Apple II Graphics

We've talked about how your Apple's memory is laid out, how to poke stuff into memory to effect the display, how to work with binary, hex, and decimal numbers, and how to animate using shapes.

This month, we'll explore the world of *byte-move animation*. This technique is very different from animating with shape tables and is used in many of today's computer games.

When you create a shape in hi-res, what you are really doing is giving the computer a set of directions to follow when it draws the shape on the screen. (For an explanation of defining shapes, refer to chapter 9 in your Applesoft manual.) Every time the shape is drawn or Xdrawn, your computer follows those directions (for instance, plot the first point, then move up, then move to the left twice without plotting, then plot that point and move down one . . .) to recreate the shape on the screen.

That procedure is fine for some applications, but each component instruction must be processed *every time the shape is drawn*, and that is relatively slow since even a simple shape can easily contain a hundred instructions. Speed is one of the primary requirements for smooth, flicker-free animation in which the figures seem to appear on the screen instantaneously.

Byte-Size Pieces. The idea behind byte-move graphics is to translate any desired figure into the corresponding data values and then poke those values directly onto the screen instead of using shapes to draw the figure. Type in the following program and run it. If you want to spare your fingers, you may omit the *rem* statements.

```
10  REM INITIALIZE Y
20  REM COORDINATES
30  REM
40  Y1% = 1:Y2% = 2:Y3% = 3:Y4% = 4:Y5% = 5:Y6% =
    6:Y7% = 7
50  REM
60  REM READ DATA FOR FIGURE
70  REM
80  FOR I = 1 TO 4: REM 4 FRAMES
90  FOR J = 1 TO 7: REM 7 BYTES PER FRAME
100 READ V%(I,J)
110 NEXT J,I
120 REM
130 REM INITIALIZE ADDRESSES
140 REM OF Y COORDINATES
150 REM
160 Y%(1) = 8192:Y%(2) = 9216:Y%(3) = 10240:Y%(4) =
    11264:Y%(5) = 12288:Y%(6) = 13312:Y%(7) = 14336
170 HGR
180 REM
190 REM POKE THE FOUR FRAMES
200 REM
210 FOR I = 1 TO 4
220 POKE Y%(Y1%),V%(I,1):
    POKE Y%(Y7%),V%(I,7)
230 POKE Y%(Y2%),V%(I,2):
    POKE Y%(Y6%),V%(I,6)
240 POKE Y%(Y3%),V%(I,3):
    POKE Y%(Y5%),V%(I,5):
    POKE Y%(Y4%),V%(I,4)
250 NEXT
260 GOTO 210: REM START AGAIN
270 REM
280 REM DATA FOR THE FOUR FRAMES
290 REM
300 DATA 1,2,4,8,16,32,64
310 DATA 8,8,8,8,8,8,8
320 DATA 64,32,16,8,4,2,1
330 DATA 0,0,0,127,0,0,0
```

If you managed to type everything in correctly, you'll see what passes for an airplane propeller spinning in the corner of your monitor screen. You must have noticed the delay in running the program before the animation began. That is a characteristic of byte-move graphics, even in the professional games, and it is caused by the need to initialize several tables before the animation can take place.

In our listing, line 40 sets the seven Y coordinates used in the figure and line 160 assigns the seven corresponding addresses. Lines 80 through 110 set up a table that contains four versions of the prop, each in a different rotation and each using seven screen lines.

To understand this, let's look at a diagram of each of the four frames, where X indicates a screen dot turned on, and – represents an off dot.

Although the figure appears to rotate like a propeller, the program is actually flashing the four frames onto the screen sequentially. It happens quickly enough that your eyes and brain are fooled into thinking the rotation is continuous—the poke is quicker than the eye!

Frame #1 in figure 1 shows the propeller running diagonally, and it also gives the binary bit pattern used to produce each dot pattern (remember, the dots are displayed as the reverse of the bits in each byte) and the equivalent decimal value. If the translation from dot pattern to binary and decimal value overloads your brain, go get something to drink, and then reread the third article in the series where we discuss the (many) peculiarities of hi-res graphics. The decimal values calculated in figure 1 correspond with the data values you see in lines 300 through 310.

The true heart of the program is the loop from lines 210 to 250 where each frame in turn is poked into hi-res screen memory. Using variables in the poke statements obscures the mechanics of what we're doing, but it also enhances the execution, as it takes more time for the computer to generate a number such as 12288 than it does to look that value up in an array.

The other reason for all the variables is that we are going to alter the routine to allow the propeller to be placed at any Y coordinate on the screen. The array Y% will contain the starting addresses for

SCREEN PATTERN	BINARY VALUE	DECIMAL
X------	0000 0001	1
-X-----	0000 0010	2
--X----	0000 0100	4
---X---	0000 1000	8
----X--	0001 0000	16
-----X-	0010 0000	32
------X	0100 0000	64
	FRAME #1	

SCREEN PATTERN	BINARY VALUE	DECIMAL
---X---	0000 1000	8
---X---	0000 1000	8
---X---	0000 1000	8
---X---	0000 1000	8
---X---	0000 1000	8
---X---	0000 1000	8
---X---	0000 1000	8
	FRAME #2	

SCREEN PATTERN	BINARY VALUE	DECIMAL
------X	0100 0000	64
-----X-	0010 0000	32
----X--	0001 0000	16
---X---	0000 1000	8
--X----	0000 0100	4
-X-----	0000 0010	2
X------	0000 0001	1
	FRAME #3	

SCREEN PATTERN	BINARY VALUE	DECIMAL
-------	0000 0000	0
-------	0000 0000	0
-------	0000 0000	0
XXXXXXX	0111 1111	127
-------	0000 0000	0
-------	0000 0000	0
-------	0000 0000	0
	FRAME #4	

Figure 1.

each line of the screen, and Y1% through Y7% will contain the seven Y coordinates used in the figure.

But let's get back to those poke statements. Line 220 pokes the first and seventh bytes, V%(I,1) and V%(I,7), line 230 pokes the second and sixth, and line 240 pokes the third, fifth, and fourth bytes. The bytes are poked in that peculiar order to improve the image, but you might like the effect obtained by poking the seven bytes in numerical order instead. Try it!

The propeller is an example of stationary animation; that is to say that though the prop moves, it always stays in the same position on the screen as it does so. Most figures you use in a game need to move around the screen, so we'll alter our program shortly to allow that. There are, however, many applications for stationary animation; the Applevision demo on your DOS 3.3 System Master is an example, as is putting a scoreboard on the hi-res screen. In the instance of the scoreboard, the frames would not be pictures of a moving object; instead you would use successive digits.

Drop the Prop. With the previous program still in memory, type in the following lines:

```
160   GOSUB 1000: REM CALC ADDRESSES
245   Y1% = Y1% + 1: Y2% = Y2% + 1:
      Y3% = Y3% + 1: Y4% = Y4% + 1:
      Y5% = Y5% + 1: Y6% = Y6% + 1:
      Y7% = Y7% + 1
1000  REM
1010  REM CALCULATE Y
1020  REM COORDINATES
1030  REM
1040  DIM Y%(192)
1050  FOR I = 1 TO 185 STEP 8: READ SA%
1060  FOR J = 0 TO 7:Y%(I + J) = SA% + J * 1024
1070  NEXT J,I
1080  DATA 8192,8320,8448,8576,8704,8832,8960,9088
1090  DATA 8232,8360,8488,8616,8744,8872,9000,9128
1100  DATA 8272,8400,8528,8656,8784,8912,9040,9168
1110  RETURN
```

Lines 1000 through 1110 calculate the starting addresses for each of the 192 lines on the screen in the same way you would find them if you were to use the method described (albeit sketchily) on page 21 of your *Apple II Reference Manual*. After each frame of the prop is poked onto the screen, line 245 increments each of the seven Y coordinates so that the next frame will appear one line below the last.

Run the modified program, and you'll see the propeller spinning as it drops down the left side of the screen. You will also see a trail of garbage left behind as the figure progresses.

Oh well, you couldn't have a program work right the first time, could you? Most of the figure is erased when the next frame is drawn over it, but since each frame is lowered one line, the top line of each frame remains to haunt you.

The problem is easily remedied by inserting:

```
244  Y0% = Y1%
215  POKE Y%(Y0%),0
```

Line 244 sets Y0% to the coordinates of the top line, and line 215 pokes a zero into that address in order to erase the old top line. Now, when you run the program, the picture moves down the screen without leaving a trail.

So Much for the Easy Stuff. So far you have done a stationary animation and a vertical animation using byte-move, but we have left horizontal animation for last. Type in and run the following routine:

```
10  HGR
20  FOR L = 8192 TO 8231
30  POKE L - 1,0: REM ERASE PREVIOUS BYTE
40  POKE L,127
50  FOR I = 1 TO 50: NEXT I
60  NEXT L
```

The program is short and simple, and it moves a line across the screen quickly, even with the delay loop. But it has one drawback fatal to any game: the animation is jerky instead of being nice and smooth. Poking the value 127 turns on all seven dots of a byte, and if you increase the delay, you'll see that the line moves in one-byte increments, which explains the uneven movement. The answer is elegant, though not without problems: move the figure along one dot at a time.

Imagine that you are looking out a window that is seven dots wide, and that the line crawls across your field of vision. At first you see only the leading dot, then the first two dots, then three, four, and so on until all seven dots are visible through the window. Then, as the line continues, the leading dot moves out of range, then the second dot follows; that continues until the window is empty.

In computer terms, the window is one byte of memory, and when just the lead dot of the line is showing in a byte, the other six dots are showing in the previous byte. In that case, you need one byte with just the left-most dot on, and another with the six right-hand dots on; the values 1 (0000 0001) and 126 (0111 1110) will do the trick. Again, remember that the bit pattern is the reverse of the desired dot pattern. (Curses!) From Basic type:

```
HGR
POKE 8192,126: POKE 8193,1
POKE 8192,124: POKE 8193,3
POKE 8192,120: POKE 8193,7
POKE 8192,112: POKE 8193,15
POKE 8192,96: POKE 8193,31
POKE 8192,64: POKE 8193,63
POKE 8192,0: POKE 8193,127
```

It is another characteristic of byte-move graphics that each figure requires seven shifted copies, or *separations*. That means that a figure one byte wide actually requires two bytes, a two byte figure requires three bytes, and so on.

Entering each pair of pokes shifts the line one dot to the right, so the cumulative effect is to move the line slowly across the screen. You could continue the process by poking the same sequence of values into locations 8193 and 8194, but at that rate it would take you several hours to go all the way across. The following program does essentially that, but faster.

```
10   DIM A%(280) : REM 280 X COORDINATES
20   REM
30   REM READ THE VALUES FOR
40   REM THE 7 PAIRS OF FRAMES
50   REM
60   FOR I = 0 TO 6
70   READ T%(I),H%(I)
80   NEXT I
90   REM
100  HGR
110  REM
120  REM INITIALIZE THE TABLE
130  REM OF ADDRESSES
140  REM
150  J = 0
160  FOR I = 8192 TO 8231
170  A%(J) = I:J = J + 1
180  NEXT
190  REM
200  REM PLOT THE LINE AT
210  REM EACH X COORDINATE
220  REM
230  FOR X = 1 TO 280
240  Q% = INT (X / 7)
250  R% = X - (7 * Q%)
260  C% = Q% + 1
270  POKE A%(Q%),T%(R%): POKE A%(C%),H%(R%)
280  NEXT X
290  END
294  REM
295  REM DATA TABLE
296  REM
300  DATA 126,1,124,3,120,7,112,15
310  DATA 96,31,64,63,0,127
```

Again, we use variables to speed up the program and confuse the reader. The values for the line are read into arrays T% (for tail) and H% (head) in lines 30 to 80, and array A% contains the addresses for each of the forty bytes across the top of the screen and is initialized in lines 120 through 180. The loop from 230 to 280 plots the line at every X coordinate across the screen, but lines 240, 250, and 260 merit more study.

The purpose of these lines is to determine which pair of bytes is being used and which of the seven pairs of values need to be poked. They do that by dividing the X coordinate by 7, and calculating the quotient (Q%) and the remainder (R%). For example, when the X coordinate is 73, seven g'zinta 73 (do you remember your g'zintas?) ten times, with three left over. So you need to poke the tenth and eleventh bytes with the third pair of values.

As you can see, even as simple a figure as the line requires seven different versions and significant preparation to animate horizontally, but the animation that results is as smooth as you could wish for, even if it is a bit slow. Most games are written in machine language to take advantage of the better speed of execution, but these examples in Basic serve to give you the idea.

Next time, we'll talk about ways you can streamline animation by doing partial modifications, preshifting, and precomputing. After that, we'll talk about some of the methods used to detect collisions between objects on the screen.

But for now, you have enough stuff to make your head hurt until the next issue arrives.

Apple II Graphics:

A High-Speed Triple Play

It's graphics time again! By now you've probably digested what we talked about last time—byte-move animation can get pretty involved. The big reason for using byte-move is speed, but with all the tables required, Basic is hard pressed to process a byte-move shape any faster than it processes a regular shape. It's really only when you use byte-move animation with machine-level routines that the advantages begin to shine through.

This month we'll look at three ways to increase the speed and efficiency of your graphics. Though the ideas may be applied either in Basic or machine code, the examples are presented in Basic for simplicity. The amount of benefit derived from each technique depends on the particular application and may vary from a lot to none at all (or worse). But since a great amount of the time spent programming any game is devoted to cleaning up the graphics, you sometimes have to be satisfied with achieving several small improvements.

We'll start with the idea of partial modification, where instead of redrawing the entire figure each time, you plot only those bytes that have changed from what they were in the previous figure. A scorecard in hi-res is a good illustration of that idea; the digits keep changing in a predictable manner, and you can use that fact to shorten your code.

Enter and run the following program (you may omit the rem statements if you wish):

```
10  REM PARTIAL MOD
20  REM
30  HGR
40  REM
50  REM POKE EIGHT
60  REM
70  POKE 8192,60: POKE 9216,66:
    POKE 10240,66: POKE 11264,60
80  POKE 12288,66: POKE 13312,66:
    POKE 14336,60: POKE 15360,0
90  VTAB 24 : PRINT "PRESS A KEY ":
    GET R$
100 REM
110 REM POKE NINE
120 REM
130 POKE 8192,60: POKE 9216,66:
    POKE 10240,66: POKE 11264,60
140 POKE 12288,64: POKE 13312,64:
    POKE 14336,60: POKE 15360,0
150 VTAB 24 : PRINT "PRESS A KEY ":
    GET R$
160 REM
170 REM POKE ZERO
180 REM
190 POKE 8192,60: POKE 9216,66:
    POKE 10240,66: POKE 11264,66
200 POKE 12288,66: POKE 13312,66:
    POKE 14336,60: POKE 15360,0
210 VTAB 24 : PRINT "PRINT A KEY":
    GET R$
220 GOTO 70
```

When you have everything keyed in correctly, you'll see the digits eight, nine, and zero cycle on the hi-res screen. That was done easily by poking the appropriate dot patterns into screen memory.

The numerals eight and nine are composed of eight rows of dots that correspond to byte values (see table 1).

DOT PATTERN	BIT PATTERN	DECIMAL VALUE	DOT PATTERN	BIT PATTERN	DECIMAL VALUE
--XXXX-	0011 1100	60	--XXXX-	0011 1100	60
-X----X	0100 0010	66	-X----X	0100 0010	66
-X----X	0100 0010	66	-X----X	0100 0010	66
--XXXX-	0011 1100	60	--XXXX-	0011 1100	60
-X----X	0100 0010	66	------X	0100 0010	66
-X----X	0100 0010	66	------X	0100 0010	66
--XXXX-	0011 1100	60	--XXXX-	0011 1100	60
-------	0000 0000	0	-------	0000 0000	0

(- indicates a dot off)
(X indicates a dot on)

Table 1.

For a refresher on how to translate the dot patterns into the values, you might look at *Softline* volume 1, number 3, where we talked about how the hi-res screen is laid out.

Now if you take a minute to compare the values used for eight with those for nine, you should notice that all except two of the values are the same. So the question arises, "Since nine always follows eight, why should I poke all the values for the nine, when six of them are the same as before?" Glad you asked. . . .

That's the idea behind partial modification—alter the existing figure instead of replacing it with a new one. To change the previous listing, type these lines:

```
130  REM
140  POKE 12288,64: POKE 13312,64
190  POKE 11264,66
200  POKE 12288,66: POKE 13312,66
```

Now lines 130 and 140 poke only the changes that are needed to turn the eight into the nine. Similarly, lines 190 and 200 poke the changes required to turn nine into zero. When you run the program you'll see the same results as before, but you'll have the satisfaction of knowing that your code is more efficient now than it used to be.

Granted, the time you save is insignificant in this example, but when you're trying to animate 150 bytes' worth of Zylon spaceship (or what have you), partial modification can potentially save a great deal of time.

The next topic to look at is precalculation. When a figure is moving around the screen, there are a lot of calculations to be made. These include the shape's X and Y coordinates, perhaps the address corresponding to those coordinates, and, if you're using byte-move, which of the seven versions of the shape is to be used at each coordinate. Calculating all of that "on the fly" can cause problems because arithmetic operations tend to require relatively large amounts of time.

Sometimes the figuring can be done after the shape is drawn on the screen and before it is erased, thus increasing the time the object is on the screen. That has the effect of increasing the ratio of the display time to the erase time, which in turn reduces flicker.

It is also possible, however, to compute the path of an object before it starts, and to store each of the coordinates in a table (Basic calls these groupings arrays). It is usually faster to look a number up in the table (especially in machine code) than it is to compute it on the spot.

In the September 1982 article, we used byte-move techniques to move a line across the screen. If you still have that program lying around on a disk somewhere, go get it—we're about to modify it. The complete listing is given below.

```
10  DIM A%(280): REM 280 X COORDINATES
20  REM
30  REM READ THE VALUES FOR
40  REM THE 7 PAIRS OF FRAMES
50  REM
60  FOR I = 0 TO 6
70  READ T%(I),H%(I)
80  NEXT I
90  REM
100 REM
110 REM
120 REM INITIALIZE THE TABLE
130 REM OF ADDRESSES
140 REM
150 J =0
160 FOR I = 14336 TO 14374
170 A%(J) = I:J = J + 1
180 NEXT
190 REM
200 REM PRE CALC
205 REM
210 DIM Q%(280),R%(280)
220 FOR X = 1 TO 280
230 Q%(X) = X / 7:R%(X) = X — Q%(X) * 7
240 NEXT X
250 HGR : REM SET GRAPHICS
254 REM
255 REM HERE GOES!!
256 REM
260 FOR X = 1 TO 280
270 POKE A%(Q%(X)),T%(R%(X)):
    POKE A%(Q%(X) + 1),H%(R%(X))
280 NEXT X
290 END
294 REM
295 REM DATA TABLE
296 REM
300 DATA 126,1,124,3,120,7,112,15
310 DATA 96,31,64,63,0,127
```

Line 100 and lines 200 through 270 contain the only modifications to the listing from last time.

The arrays Q% and R% hold the quotients and remainders for each of the 280 X coordinates. Dividing the X value by seven tells you which byte across the screen to address (0 to 39), and which of the seven versions of the figure should be used. (Remember from last month?)

For example, if you wish to start the figure at the thirty-first coordinate, divide 31 by 7 (4 with remainder 3). This tells you to poke the third version of the figure into byte number 4. Since the starting address for our screen line is 14336, we poke the value into 14340 (14336 + 4).

Lines 160 through 180 fill array A% with the addresses for each of the forty bytes across the screen line, and lines 220 through 240 calculate the 280 quotients and remainders. Finally, line 250 turns on hi-res and line 270 does the actual poking.

Sorry about the compound indexing—A%(Q%(X))—but it couldn't be helped. X is the coordinate number so Q%(X) is the quotient belonging to that coordinate, and A%(Q%(X)) is the address determined by that quotient.

This new version of the program runs the line across the screen in seven seconds, as opposed to the nine seconds the earlier one required. Now seven seconds is still pretty slow (blame Basic), but precalculating did result in a significant improvement (22 percent, since you asked).

And now, on to what is perhaps the most elegant of the techniques: preshifting. We're going to animate using a shape table, so from Basic type:

```
CALL —151
300:  02 00 06 00 45 00 3F 3F
308:  3F 3F 3F 3F 08 2D 2D 2D
310:  2D 2D 2D 18 3F 3F 3F 3F
318:  3F 3F 08 2D 2D 2D 2D 2D
320:  2D 18 3F 3F 3F 3F 3F 3F
328:  08 2D 2D 2D 2D 2D 2D 18
330:  3F 3F 3F 3F 3F 3F 08 2D
338:  2D 2D 2D 2D 2D 18 3F 3F
340:  3F 3F 3F 3F 00 24 24 24
348:  24 DF DB DB DB 06 36 36
350:  36 36 00 00 00 00 00 00
3D0G
BSAVE SQUARE,A$300,L$53
```

With the table still in memory, let's find out what we have there. Type:

```
POKE 232,0: POKE 233,3
HCOLOR = 3:ROT = 0:SCALE = 1:HGR
DRAW 1 AT 50,50
```

The first line tells Applesoft where the table is stored ($0300) in lo-byte/hi-byte from (00 and 03). The second line sets all the parameters, and the third draws the first shape, a rectangle, at 50,50 on the hi-res screen. For more information on those commands, refer to the Applesoft manual, pages 98 and 99.

The second shape in the table happens to be the horizontal preshift (the what!??) of the original rectangle. Imagine that you are about to shift the rectangle one place to the right. The bulk of the figure is unchanged; there is a single line added to the right side and one deleted from the left side. Preshifting, like partial modification, is a way to process only that portion of the figure that changes, while leaving the rest of it alone.

Now type:

```
DRAW 2 AT 51,60
```

to draw the preshift below the rectangle. Notice that the preshift has a single line to the right of the rectangle and another that lines up with the left side of the rectangle. To effect the modification, we will xdraw the preshift on top of the rectangle.

Let us digress for a moment. When used to superimpose one shape on another, xdraw has the effect of comparing corresponding dots of the two shapes and forming a resultant figure from them. The resultant dot is on if either one of the original dots was on, but not if both were. Table 2 summarizes the results from the four possibilities.

Dot#1	ON	ON	OFF	OFF
#2	ON	OFF	ON	OFF
Result	OFF	ON	ON	OFF

Table 2.

So when the preshift is xdrawn over the rectangle, the dots on the left side of both figures are on. This has the effect of turning off that whole row. But the right side of the preshift is one row beyond the rectangle, and since only the preshift dots are on, the result is that that row of dots will be turned on. But enough of words; let's try it! From Basic type:

```
XDRAW 2 AT 51,50        XDRAW 2 AT 54,50
XDRAW 2 AT 52,50        XDRAW 2 AT 54,50
XDRAW 2 AT 53,50        XDRAW 2 AT 53,50
```

The rectangle will move to the right and then back to the left. You may be surprised at first by the two xdraws at 54, but remember, two consecutive xdraws always cancel each other out. In this instance, the first one moves the rectangle to the right, and then the second cancels out the effect of the first and begins to move the rectangle back to the left. Play with xdrawing this figure until you can move it around comfortably.

The following program uses this idea to move the square across the screen.

```
10  REM PRESHIFT
20  REM
30  D$ = CHR$ (4)
40  PRINT D$"BLOAD SQUARE"
50  POKE 232,0: POKE 233,3
60  HCOLOR= 3: ROT= 0: SCALE= 1
70  HGR
80  DRAW 1 AT 20,100
90  FOR I = 21 TO 275
100 XDRAW 2 AT I,100
110 NEXT I
```

There are two very pleasant surprises with this program. The first is that it is short and simple, and the second is that it moves the figure quickly and with very little flicker. Ta-daa!

Apple II Graphics

A vital element of most every arcade game in existence is the ability to detect a collision between two objects on the screen. For you, seeing when two objects meet is trivial. All you have to do is look at the monitor! But your Apple has no eyes, so it's in the same position as a blind man who uses a cane to discover objects in his path. Actually, that analogy contains more truth than it has a right to.

Imagine what's involved in dealing with two objects on the screen. Theoretically it's possible to keep a list of all the X,Y coordinates used in each shape and check for collision by seeing if there is a coordinate pair that lies in both shapes.

Suppose that one shape is plotted on the points (1,1), (1,2), and (1,3) and the second uses coordinates (1,2), (2,2), and (3,2). In this case there must be a collision, since the point (1,2) is a part of both shapes.

But that method quickly becomes unworkable. In animation you are continually changing the points used in each shape, so you'd have to keep updating all the lists. More important, the task of crosschecking each point gets out of hand since even a trivial figure will contain perhaps ten points. Crosschecking two such figures would call for 100 checks, and checking three would require 1,000! So much for that idea.

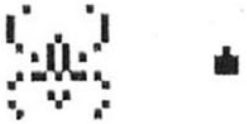
The Blind Man and the Elephant. Let's look now at two more methods, both of which use the blind man's cane approach. We will fire a missile at a shape borrowed from *Crossfire* (by J. Sullivan), and, as the missile moves, we'll continually check its path. So take a few minutes now to enter and save the shape table that follows. From Basic type *call −151* to enter the Monitor, then enter these lines.

```
300:  02 00 06 00 4A 00 4D 49
308:  49 69 18 DF DB DF DB 07
310:  48 49 69 4D 49 18 DF DB
318:  DB DB 07 48 0D 6D 0D 6D
320:  0D D8 DB FB DF DB 48 09
328:  0D 0D 0D 4D 01 D8 DF FB
330:  DF FB 08 4D 49 4D 49 05
338:  18 DF DB DB DB 07 08 4D
340:  49 49 49 05 D8 FB DB DB
348:  DF 00 36 27 0D 36 00 00
```

Now type *3D0G* to get back to Basic, and enter these commands:

```
BSAVE XFSHAPE, A$300,L$50
POKE 232,0:POKE 233,3
HGR:HCOLOR=3:SCALE=1:ROT=0
DRAW 1 AT 20,20
DRAW 2 AT 20,50
```

The first shape, our target, should look a bit like a spider. The second shape is our missile. The shapes should look like this:



If they don't look right, check your work by typing:

```
CALL −151
300.34F
```

When you get the table saved correctly, you are ready to animate. We'll leave the spider alone and move the missile back and forth across the bottom of the screen using the paddles.

When we press the paddle button, the missile moves up the screen until it bumps into the spider or hits the top of the screen. To create that movement, we will repeatedly plot and erase the missile, moving it up one point each time. Before the missile is moved up each time we'll check the space it is moving to to see if the spider is there.

We'll develop the program in two stages, so key in the following to get the basic movement going. You may omit any rem statement except for those in lines 150 and 500.

```
10  D$ = CHR$ (4)
20  PRINT D$"BLOAD XFSHAPE"
30  M0 =  5:M = 5
80  REM
85  POKE 232,0: POKE 233,3
90  HGR : HCOLOR= 3: ROT= 0: SCALE= 1
100 DRAW 1 AT 140,50
110 XDRAW 2 AT M,150
120 REM
130 REM CHECK PADDLE
140 REM BUTTON
150 REM
160 IF PEEK ( − 16287) > 127 THEN GOSUB 500
170 REM
180 REM MOVE MISSILE
190 REM
200 IF PDL (0) < 90 THEN M = M − 2: GOTO 220
210 IF PDL (0) >  150 THEN IF M < 277 THEN M = M + 2
220 IF M < 1 THEN M = 1
230 IF M = M0 GOTO 250
240 XDRAW 2 at M0,150: XDRAW 2 at M,150: M0= M
250 GOTO 150
480 REM
490 REM COLLISION DETECT
500 REM
510 REM WE'LL PUT THIS IN SHORTLY!
520 PRINT CHR$(7):RETURN
```

This much will get the spider onto the screen and the missile moving and firing under the control of paddle 0. When you press the paddle 0 button, the computer will beep at you.

Whenever the missile moves, M0 holds the old X coordinate of the missile and M holds the new one. The missile is moved by decrementing or incrementing M (lines 200 and 210) and then xdrawing to erase the old missile and xdrawing again to plot the new one (line 240). If you wonder about the function of any line, try deleting it and see what doesn't happen.

Arachnid Annihilation. After you've got the missile moving across the bottom of the screen, we'll talk about the collision routine. Run the program as you have it so far and press control-C to recapture control, leaving the spider on the screen. Imagine that there is a little box drawn around the spider. In fact, put the box there by typing these four lines from Basic:

```
HPLOT 137,38 TO 153,38
HPLOT TO 153,51
HPLOT TO 137,51
HPLOT TO 137,38
```

As we move the missile up the screen, we will check to see if it moves inside that box, and if it does . . . *boom!*

Add these lines to your program:

```
40  REM
50  REM SET BOUNDARIES
60  REM
70  YMAX = 51:MINX = 137:MAXX = 153
510 PRINT CHR$ (7): REM BELL
520 C = 0: REM COLLISION FLAG
530 FOR Y = 149 TO 0 STEP - 1
540 IF Y < YMAX THEN IF M > MINX AND M < MAXX THEN C =
    1: REM COLLISION!!
550 IF C THEN PRINT CHR$ (7); CHR$ (7); CHR$ (7): XDRAW 2
    AT M,Y + 1: GOTO 620
560 XDRAW 2 AT M,Y + 1: XDRAW 2 AT M,Y
570 NEXT Y
580 REM ERASE MISSILE AT TOP
590 Y = Y + 1
600 XDRAW 2 AT M,Y
610 REM DRAW NEXT MISSILE
620 XDRAW 2 AT M,150
630 RETURN
```

Line 70 sets the boundaries for the square. The top boundary is ignored in this example because the missiles always come from below.

Line 540 is the real detection line. It first checks the Y coordinate of the tip of the missile to see if it is high enough to strike the box. If it is, the X position is checked to see if it is in the correct horizontal range. If all of these conditions are true, we have a collision and the flag (C) is set to one for later reference.

Line 550 is where you ordinarily would jump to your nifty explosion routine, but since this is only an example all we do is set off the bells and whistles, erase the missile from the bottom of the box, and then jump to where the new missile is drawn at the bottom—big deal! As before, the reason for obscure statements like the one in line 590 can be found by deleting them and watching for what messes up.

You may think it would be more efficient to check the X coordinate of the missile before checking the Y, thereby determining whether or not there would be a collision before the missile even started to move up the screen. Before you sharpen your pencils to write Directline and tell us that . . . okay, you're right; you win. But that only works when the target is stationary, and you probably won't feel much like rewriting the routine after getting it all checked, typed up, and checked again.

On the other hand, maybe you will. . . .

The box method is fairly simple to use, but since it declares a collision based on some imaginary square instead of actual contact with the target it's also a little on the sloppy side. Furthermore, we only checked for the center of the missile, which may actually miss the spider even when one of the edges hits it. Unfair! If you've ever played a game that has sloppy collision detection (and there are a bunch of them) you know how frustrating it can be.

Checking across the full width of the missile is pretty easy; try it yourself. The problem of detecting an actual contact is more interesting.

Connect the Dots. Our collision detection routine involves taking a peek at hi-res memory to see if you are about to move onto a dot that is already turned on. (In this example, the only dots turned on are in the target.)

Dealing directly with memory is always complex from Basic, and it gets worse in this instance because we want to look at individual bits. To set this process in motion, type in the following changes to the existing program:

```
DEL  40,70
5    GOSUB 1000: REM CALC Y ADDR.
515  Q% =  M / 7:R% = M - 7 * Q%:R% = R% + 1
536  V% = PEEK (Y%(Y) + Q%)
538  FOR I = 7 TO R% STEP - 1
540  P% 2 ^ I
542  IF V% > = P% THEN V% = V% - P%
544  NEXT I
546  IF V% > = 2 ^ I THEN C = 1
1000 REM
1010 REM CALCULATE Y
1020 REM COORDINATES
1030 REM
1040 DIM Y%(192)
1050 FOR I = 0 TO 184 STEP 8: READ SA%
1060 FOR J = 0 TO 7:Y%(I + J) = SA% + J * 1024
1070 NEXT J,I
1080 DATA 8192,8320,8448,8576,
     8704,8832,8960,9088
1090 DATA 8232,8360,8488,8616,
     8744,8872,9000,9128
1100 DATA 8272,8400,8528,8656,
     8784,8912,9040,9168
1110 RETURN
```

Line 5 and lines 1000 through 1110 should be familiar from past articles in this series; they set up a table that assigns to each line on the screen (0 through 191) its starting address.

Line 515 divides the X coordinate by seven to determine which of the forty bytes across the screen the missile is in (Q%) and which dot of that byte (R%) it's on. Let's suppose that the missile is fired along X coordinate 45, so that the tip is in byte 6 and dot 3 (counting the first dot in each byte as 0). Then we need to check dot 3 in the byte above the missile before moving the missile up to each new Y coordinate. As long as that dot is off, there's nothing to hit. If that dot is on, we've run into the target and need to set the collision flag.

The process of isolating a particular dot is clumsy in Basic, but lines 538 through 546 do the job. Line 536 sets V% to the value of the byte you need to check. To understand how those lines operate, it would be instructive (though insufferably tedious) to set V% to 60 (which would be 0011 1100 in binary) and run through those lines by hand.

Like the previous routine, this routine only checks the center of the missile, so it's possible to contact the target with the left or right sides and still not detect a collision. Two trivial modifications to the routine will cause it to check across the entire width of the missile, but you can discover them for yourself.

Faster than a Speeding Snail. When you run this version of the program, you'll notice that a collision is detected only when the missile actually contacts the target. You'll also notice that the animation is slower as a result of the increased processing involved in this scheme—you can't have everything.

With a little effort, the Basic code can be streamlined to enhance the execution, but the real gain is made by writing this collision detector in machine code, something we won't cover here.

The contact collision detection is further complicated if you have other objects on the screen, such as clouds, which the missile is required to ignore. There are two common ways of handling that.

In some games, a table is kept of the current position for each target. When any collision is flagged, that table is scanned to see if there is a target close by. If so, the target is destroyed; if not, the collison is ignored. One interesting side effect of this technique is that when two targets are close together the scanning routine will sometimes decide that the wrong one has been shot.

Hi-Res Sleight of Hand. The other method of dealing with extraneous objects is to use both hi-res screens. One is displayed with all the junk on it, while the other has only the missiles and targets on it and isn't displayed. Then the collision detection routine can check the second screen and not even worry about the extra figures! In some applications the extra time needed to process two screens is offset by not having to spend time deciding what was run into.

That about wraps it up. We hope this series has shed some light on ways to make Apple graphics do what you want them to and how a lot of the games achieve their effects. We're interested in your comments on these articles and your suggestions for future ones. Send them to Ken's Graphics, Softline, Box 60, North Hollywood, CA 91603.