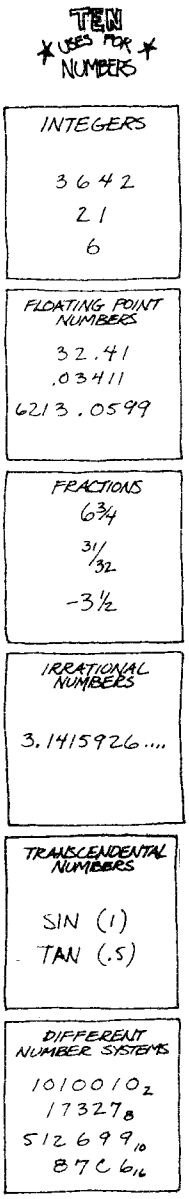
20.



What is a number? I've been wanting to ask you that question at just about every session, but I think now's the time for it. What is a number?

No matter what comes to mind in response to the question, it's probably right, and that means the computer has to deal with it. Somehow, the binary data has got to be arranged to handle all those conceptions. Numbers might include . . .

- integers both positive and negative.
- floating-point decimal numbers
- fractions
- irrational numbers and transcendental functions
- different number systems
- identification or code numbers
- scales or scientific ranges
- money
- very large or very small numbers

Some of these — like floating-point numbers and money — are just slight conceptual variations. Others — like transcendental numbers and different number base systems — are strikingly dissimilar.

Learning the details of handling all these different numbers in assembly language would require a separate course, so I'm going to limit the discussion to simple numbers. Once you've got this session down, you'll be ready for all the rest. You already understand positive and negative integers, so I mean to go one step further — to floating point numbers and how they are represented in binary notation.

Signed and unsigned integers have been the limit for the calculation and conversion examples so far. Numbers are mind-bogglingly more than that, and binary format has to be forced to handle them all. Floating-point notation -- that is, representation of decimal numbers -- is far and away the most obscure topic in assembly language. Even to the experienced, it comes only with irritation.

* What is an integer?

A whole number with no fractional part.

* How does the 6809 represent signed integers?

By using the most significant bit (the leftmost bit) of a number.

* What is the sign for positive and negative?

A zero in the most significant bit is positive; a one in the most significant bit is negative.

Accuracy and range

* Show $8FC2 in binary; is it positive or negative? Why?

$8FC2 is 1000 1111 1100 0010, and it is negative because the most significant bit is a one.

* What is a floating-point number?

A number with a fractional part.

* Can 326 be a floating-point number? Why?

Yes, because the fractional part is zero (.0000000....)

* What is the accuracy of the Color Computer?

9 significant digits.

* What is a significant digit?

The part of the actual number used in storage or computation.

* What are the signficant digits of 123,456,789,876,543,210 on the Color Computer?

The most significant digits are 123456789.

* How would 123,456,789,876,-543,210 be displayed on the Color Computer?

It would be displayed 1.2345679E+17.

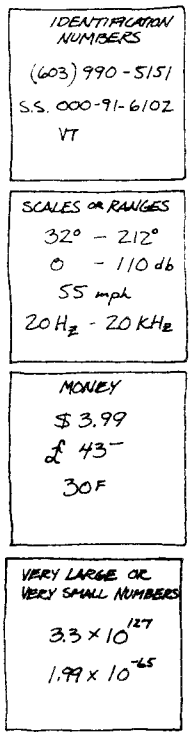
* What does the E mean in 1.2345679E+17?

E means exponent, that is, the power of 10 by which the number is multiplied; in other words, 1.2345679 times 10 to the 17th.

* What is 10 to the 17th (10+17)?

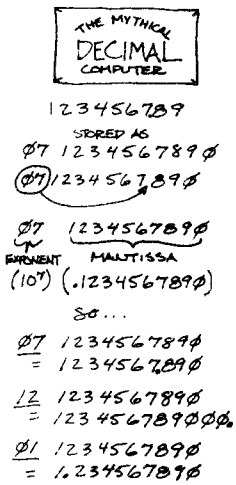
100,000,000,000,000,000

"Floating point" is jargon for numbers in complete form — positive or negative numbers, with integer and fractional portions. All numbers in the Color Computer's BASIC are stored as floating point numbers, whether they look like integers or not. The number 10, for example, is actually thought of as 10.0000000 with the computer's internal hexadecimal representation **$84 20 00 00 00**. One million is thought of as 1000000.00, with the internal hexadecimal representation of **$94 74 24 00 00**. 0.1 becomes 0.100000000 and is represented by hexadecimal **$7D 4C CC CC CD**, and one-millionth is 0.0000010 and is stored as hexadecimal **$6D 06 37 BD 06**.



Don't expect these hexadecimal patterns to make any sense as I read them to you. They are, in fact, five-byte groupings capable of representing any number from –170,141,173,000,000,000,000,000,000,000,000,000,000 (negative 170 trillion, 141 billion, 173 million billion billion billion) to +170,141,173,000,000,000,000,000,000,000,000,000,000. The Color BASIC language can handle these with nine significant digits of accuracy — that is, only the first nine digits are used for the actual computations. This is excellent accuracy (far better than my old 4-digit slide rule), but not always enough for the modern age of high technology, with its measurements of astronomical vastness or molecular smallness. By understanding how floating point numbers are represented, it is possible to extend the accuracy of numbers to as many digits as you need. No matter how fast the machine's speed, handling such large numbers will take time; but handling large and small numbers will be possible — even via BASIC.

Now to what those numbers mean. The principle is, once again, disarmingly simple. Let me start the explanation as if you were using a decimal computer instead of a binary one. Take the decimal number 1234567.89. Now say this decimal computer you own has a precision of 10 significant digits. The number is really 1234567.890 for your computer. And of course this decimal computer doesn't have a decimal point inside the number — it can only store information on where the decimal point is. It won't actually put one there except for display.



So the number is 1234567.89, meaning the decimal point is between the seventh and eight positions. So by storing 7 followed by 1234567890, you can say that the number stored in your special decimal computer is 1234567 point 89, with the trailing zero dropped. Simply by changing your descriptive information you can change the number's power of 10. By storing 12 followed by 1234567890, you automatically know that the number you want is 123,456,789,000. By storing 1 plus 1234567890, the number becomes 1.23456789.

There's no difference in the Color Computer's representation of numbers from the description of this imaginary decimal computer. The five bytes used to describe a floating point number on the Color Computer are in binary. That's the key. To represent one million as

$94 74 24 00 00 is to store it with one descriptive byte telling where the point is, plus a string of binary digits. Here's a case where hexadecimal is pretty useless. Binary is the only solution to seeing it.

The exponent byte comes first, which is a power of 2 — essentially a description of where the decimal point goes. In this case, I'm going to coin a term . . . I think this should be called a binaral point, since this is binary notation. **$80** is the central value around which the binaral point swings. From **$81** to **$FF** represents from 1 to 127 places to the left of the binaral point — numbers greater than one; from **$01** to **$7F** represents from 127 to 1 places to the right of the point.

Back to the number one million, stored as **$94 74 24 00 00. $80** is the pivot point, so **$94** minus **$80** is **$14**. That means that this number has hex **$14** — decimal 20 — digits to the left of the binaral point. I'll write the remainder of this in binary:

```
0111 0100 0010 0100 0000 0000 0000 0000
```

The leftmost bit of these 32 digits is used as the sign bit; as usual, 0 is the positive sign and 1 is the negative sign. In numerical terms (exactly why is difficult to explain but will become clear with experience), this bit is assumed to be a 1 for calculation purposes. That is, since any number's got to have some digit to multiply by other than zero, at least one 1 will appear . . . and that's the case no matter whether the number is positive or negative. So whether the sign bit is 1 or 0, this bit is included in the calculation as if it were a 1. Turning back to the string of binary digits, it becomes (please follow along in the book now):

```
1111 0100 0010 0100 0000 0000 0000 0000
```

Since the point is after the 20th position (hex **$14**), count over from the left. The left, for one of the rare times in computer terms, is called the **first** rather than the zeroeth position. Putting the point in place makes the number read:

```
1111010000100100000.00000000000000
```

Now you do one of two things. The first option is to sum the powers of two to calculate the result, starting from just left of the point. Zero times 2↓0 plus 0 times 2↓1 plus 0 times 2↓2, keeping the sum as you move on up to 1 times 2↓19. That's actually the sum of 2↓19 + 2↓18 + 2↓17 + 2↓16 + 2↓14 + 2↓9 + 2↓6, which is 1,000,000.

Or as an alternative you can break the binary into four-bit groups, again starting from the immediate left of the point, and convert those to hexadecimal: it becomes **$F4240**. According to my hexadecimal calculator, **$F4240** is, indeed, one million in decimal.

I'm going to take you through a few of these for practice. Let me hand you just any five-byte group that comes to mind as I put this lesson together. I'll keep it positive and large until

```
* What is 2 to the zeroeth (2↑0)?

1 (any number to the zero power is 1).

* What is 2 to the 1st (2↑1)?

2

* What is 2 to the 2nd (2↑2)?

4

* What is 2 to the 3rd (2↑3)?

8

* What is 2 to the 16th (2↑16)?

65536

* What is 2↑0 plus 2↑1 plus 2↑2 plus 2↑3?

1+2+4+8, or 15.

* What is 2↑0 plus 2↑1 plus 2↑2 ... up to 2↑15?

1+2+4 ... +32768, or 65535.

* In floating-point binary notation, what is the first byte?

The exponent byte.

* If the exponent byte is $99, what does it indicate?

It indicates 2 to the power $19 (hex) or 2↑25 (decimal).

* If the exponent byte is $81, what does it indicate?

It indicates 2 to the power $01, or 2.

* What are the four bytes following the exponent byte called?

The mantissa.
```

Placing the point

```
* If the four bytes are $5A 00
00 00, what is the binary
mantissa?

The binary mantissa is 0101 1010
0000 0000 0000 0000 0000 0000.

* Which bit is the sign bit?

The leftmost bit is the sign
bit.

* If the mantissa is $5A 00 00
00, what is its sign?

The sign of $5A 00 00 00 is
positive (the leftmost bit is a
zero).

* What is a "normalized"
mantissa?

A mantissa in which the leftmost
bit has been set to a one after
its sign is known.

* If the (normalized) mantissa
is 1101 1010 0000 0000 0000 0000
0000 0000 and the exponent byte
is $88, place the point.

1101 1010. 0000 0000 0000 0000
0000 0000

* What is this number?

2↑1 + 2↑3 + 2↑4 + 2↑6 + 2↑7 =
218

* If the (normalized) mantissa
is 1101 1010 0000 0000 0000 0000
0000 0000 and the exponent byte
is $91, place the point.

1101 1010 0000 0000 0.000 0000
0000 0000

* What is this number?

2↑10 + 2↑12 + 2↑13 + 2↑15 + 2↑16
= 111,616
```

you get the hang of it. Let's say the bytes you see are **$9F 66 7D 80 1F**. Write those bytes down. **$9F 66 7D 80 1F**. The leftmost byte is the power-of-two exponent, you recall, revolving around the **$80** pivot point. **$9F** minus **$80** is **$1F**, so you know that this number is **$1F** (that is, decimal 31) digits long. The digits themselves are **66 7D 80 1F**, which, when translated into binary, become:

```
    0110 0110 0111 1101 1000 0000 0001 1111
```

You can follow along in the book or write those down. The leftmost bit is the sign bit; it's zero, so this is a positive number. Now you can replace the zero with the "normalized" one for calculation purposes. Here's the number:

```
    1110 0110 0111 1101 1000 0000 0001 1111
```

The binaral point is after the **$1F**th digit . . . that's the 31st digit. So the binary number now is:

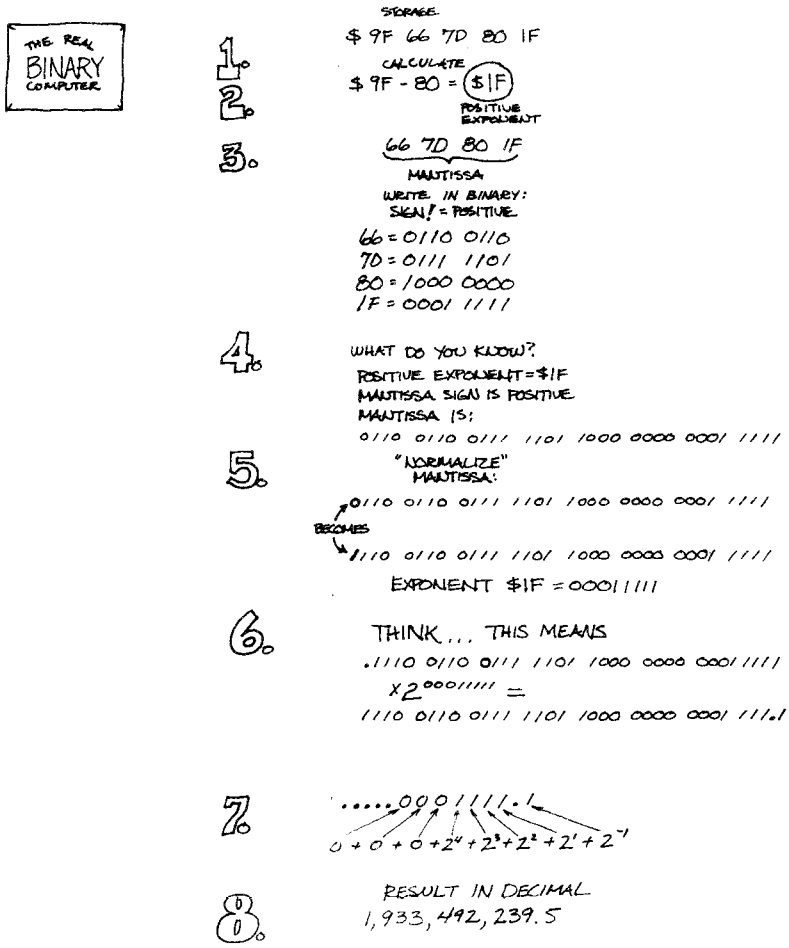
```
    11100110011111101100000000001111.1
```

I'll do the sum with the powers of two method. 2↑0 + 2↑1 + 2↑2 + 2↑3 + 2↑14 + 2↑15 + 2↑17 + 2↑18 + 2↑19 + 2↑20 + 2↑21 + 2↑24 + 2↑25 + 2↑28 + 2↑29 + 2↑30 works out to 1,933,492,239. Remember, you start from the immediate left of the point and sum up the powers of two. The result once again is 1,933,492,239.

But what about that .1 at the end of the binary string? What is that and how do you use it?

In decimal, the numbers to the right of the decimal point represent negative power of 10, or, if you like 1/10ths, 1/100ths, 1/1000ths, 1/10000ths, etc. In binary, the numbers to the right of the point represent — you guessed it — negative powers of two. one-halves, one-quarters, one-eighths, 1/16ths, 1/32nds, 1/64ths, etc. So that ".1" at the end represents 1/2, or in decimal, 0.5. The resulting number should therefore be 1,933,492,239.5 on the Color Computer.

That's both right and wrong. A few minutes ago I said that the Color Computer BASIC's accuracy is only nine digits. That's a choice made mostly for reasons of speed and consistency. If you write X = 1933492239.5 and enter it on the computer, your PRINT X will reveal 1.9334922 4E+09. That's BASIC's scientific notation for 1.93349224 times 10 the 9th. In other words, not 1,933,492,239.5, but rather 1,933,492,240. Only nine significant digits are used, so part of the number gets rounded off and abbreviated.



Another random example. Stay with me. **$7C 91 32 2F 00**. Write it down to work with. **$7C 91 32 2F 00**. **$7C** in the first position is the exponent byte, but this time it's less than the **$80** pivot. This is a number with values all to the right of the decimal point. That is, a number less than 1. **$7C** is –4 binary positions, so you know the binary number begins with .0000 .

The hexadecimal for the rest of the number is **$91 32 2F 00**, which is, in binary:

```
1001 0001 0011 0010 0010 1111 0000 0000
```

The one in the leftmost position means this time we've got a negative number. Now that you know that, you'll also remember that it is "normalized" to one for purposes of calculation. So the result is

```
0001 0001 0011 0010 0010 1111 0000 0000.
```

With .0000 in front of it, it becomes:

```
.0000100100010011001000101111000000000
```

Normalized notation

```
* If the mantissa is 1101 1010 0000 0000 0000 0000 0000 0000 and the exponent byte is $99, place the point.

1101 1010 0000 0000 0000 0000 0.000 0000

* What is this number?

2↑18 + 2↑20 + 2↑21 + 2↑23 + 2↑24 = 28,573,696.

* If the (normalized) mantissa is 1101 1010 0000 0000 0000 0000 0000 0000 and the exponent byte is $A0, place the point.

1101 1010 0000 0000 0000 0000 0000 0000.

* What is this number?

2↑25 + 2↑27 + 2↑28 + 2↑30 + 2↑31 = 3,657,433,088

* What are the two parts of floating-point representation called?

The exponent and the mantissa.

* What is 1 divided by 2↑1?

1/(2↑1) is 1/2.

* What is 1 divided by 2↑2?

1/(2↑2) is 1/4.

* What is 1 divided by 2↑16?

1/(2↑16) is 1/65536.

* If the (normalized) mantissa is 1110 0000 0000 0000 0000 0000 0000 0000 and the exponent byte is $82, place the point.

11.10 0000 0000 0000 0000 0000 0000 0000

* What is this number?

2↑1 + 2↑0 + 1/(2↑1) = 3 1/2 = 3.5
```

* If the normalized mantissa is 1111 1111 1111 1111 1111 1111 1111 1111 and the exponent byte is $90, place the point.

1111 1111 1111 1111. 1111 1111 1111 1111

* What is this number?

2↑0 + 2↑1 + 2↑2 + 2↑3 + 2↑4 + 2↑5 + 2↑6 + 2↑7 + 2↑8 + 2↑9 + 2↑10 + 2↑11 + 2↑12 + 2↑13 + 2↑14 + 2↑15 = 65,535. 1/(2↑1) + 1/(2↑2) + 1/(2↑3) + 1/(2↑4) + 1/(2↑5) + 1/(2↑6) + 1/(2↑7) + 1/(2↑8) + 1/(2↑9) + 1/(2↑10) + 1/(2↑11) + 1/(2↑12) + 1/(2↑13) + 1/(2↑14) + 1/(2↑16) = .99998474121094. Therefore, the number is 65535.99998474121094.

* What would this number be in the Color Computer's most significant digits?

65535.99998 is rounded off (up) and becomes 65536.

* What are the two parts of floating-point representation called?

The exponent and the mantissa.

* What information does BASIC's VARPTR provide?

The address of a BASIC variable.

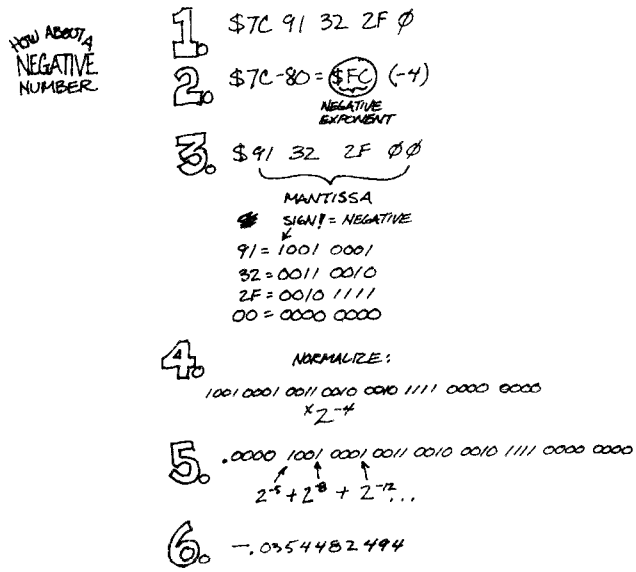
* In the case of floating-point variable N, what is found at VARPTR(N)-2 through VARPTR(N)+4?

VARPTR(N)-2 and VARPTR(N)-1 are the name of the variable; VARPTR(N) is the exponent; and VARPTR(N)+1 through VARPTR(N)+4 comprise the mantissa.

* What does VARPTR mean?

Variable pointer.

These are now fractional powers of two. You must sum 1/2 plus 1/4 plus 1/8, and so on. You can think of this calculation as: zero times 1/2↑1 plus 0 times 1/2↑2, etc. In this case that's 1/2↑5 + 1/2↑8 + 1/2↑12 + 1/2↑15 + 1/2↑16 + 1/2↑19 + 1/2↑23 + 1/2↑25 + 1/2↑26 + 1/2↑27 + 1/2↑28 . . . that is, –.035448249429465 . Your Color Computer would report that as the slightly less precise –.0354482494. The number stored in the computer as **$7C 91 32 2F 00** becomes –.0354482494.



While we're at it, let me quickly bring back one of the examples I gave at the start . . . I said that 0.1 was represented as **$7D 4C CC CC CD**. Have a look. **$7D** is a right-of-the-point prefix of .000 and the number **4C CC CC CD** translates to binary . . .

0100 1100 1100 1100 1100 1100 1100 1101

It's positive; the normalized one changes the number to:

1100 1100 1100 1100 1100 1100 1100 1101

Together with the prefix, it reads:

.0001100110011001100110011001100110011101

Calling that out in powers of two, it's 1/2↑4 + 1/2↑5 + 1/2↑8 + 1/2↑9 + 1/2↑12 + 1/2↑13 + 1/2↑16 + 1/2↑17 + 1/2↑20 + 1/2↑21 + 1/2↑24 + 1/2↑25 + 1/2↑28 + 1/2↑29 + 1/2↑32 + 1/2↑33 + 1/2↑35 . . . and that calculates to .10000000000582. Notice that residual .00000000000582 tacked on to the end of the number. That's the tiny binary error that you've probably experienced creeping into BASIC calculations.

More on that error and other floating point concepts after a break. You might be weary of all these calculations. They'll get both easier and unnecessary later. For the moment, I would like you to review the lesson up to this point, book in

hand, calculating along with me on paper or on your hand calculator. There's a lot to this floating point math, and understanding how to push around those bits is vital if you wish to work with numbers on your computer.

Please review the concept of floating point numbers. When you are confident of the theory of floating point binary notation, return to the tape.

Here's a summary of floating point numbers. As stored in BASIC's variable table, they consist of seven descriptive bytes. The first two bytes are the variable name; the last five represent the number itself. The first byte of the group of five is the binary exponent, from 2↑–127 to 2↑+127. The next four bytes are the mantissa, that is, the number itself, expressed in binary digits. The leftmost bit of the 32-bit group is normalized to a one for purposes of calculation, but as stored it represents the sign of the number.

This complex-sounding process provides, in five bytes, the ability to store decimal numbers across the range +/–1.70141176E+38 (2↑126.9999999) to +/–5.87747201E–39 (2↑–126.9999999), with nine digits of accuracy.

There are two ways to access BASIC variables from machine language. One way is via the USR command, and the other is by accessing BASIC's variable table. The variable table storage is the one I've been describing. There is another slightly different kind of storage when the variable is transferred via USR. It's described in the Extended BASIC manual on pages 147 and 149, under the heading "USR Function Arguments". You've already read part of this, but now it should make more sense; take a few minutes to re-read that now, and pay special attention to the description of the "Floating Point Accumulator".

Open the Extended Color BASIC manual and re-read pages 147 through 149, headed "USR Function Arguments," concentrating on the new information on page 147. Read this thoroughly, as it now applies to your understanding of binary floating point representation. Return to the tape when you have completed the reading.

You haven't been told the whole story in that reading. You should know that the contents of VARPTR(X)–2 and VARPTR(X)–1 are the variable's name. VARPTR is an excellent function, one that machine language programs can use extensively.

```
* What is an integer?

A whole number with no
fractional part.

* What is a floating-point
number?

A number with a fractional
part.

* What are the two parts of
floating-point representation
called?

The exponent and the mantissa.

* The exponent and the mantissa
are in what number system?

The exponent and the mantissa
are in binary (base 2).
```

Just for a taste of the use of VARPTR, type and enter X=1. Simply X=1, and enter. PRINT X will display the number 1. Do it to be certain; PRINT X. Now POKE VARPTR(X),&HFF. That's POKE VARPTR(X),&HFF. PRINT X. The result will be 8.50705918E+37. VARPTR(X) is the exponent of the number, which that POKE with **$FF** has raised to an enormous power. *Now* POKE VARPTR(X),1. That's POKE VARPTR(X),1. The result is just the opposite: PRINT X will reveal the amazingly small value 2.93873588E–39.

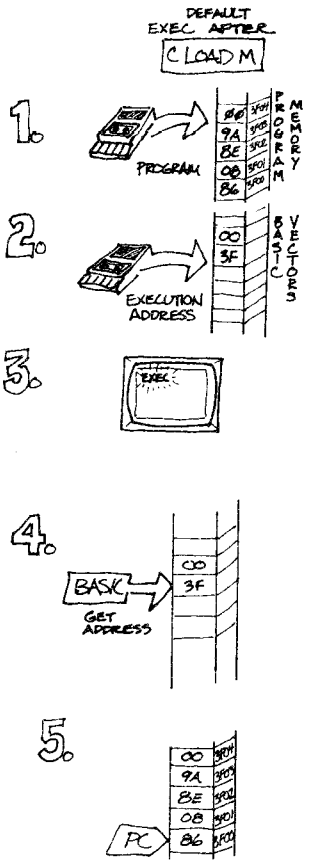
Now get things in range. POKE VARPTR(X),&H88. That's POKE VARPTR(X),&H88. Now PRINT X. You've got 128. Now mess around with the rest of the number. POKE VARPTR(X)+3,1. POKE VARPTR(X)+3,1. And then PRINT X. Now it's 128.000015. How about POKE VARPTR(X)+2,&HAA. It's 128.664078. VARPTR(X) is the power-of-two exponent, VARPTR(X)+1 through VARPTR(X)+4 are the binary digits of the mantissa. POKE around with the 5-byte descriptor at the end of this lesson; it should give you additional perspective on how those numbers are stored. As a real exercise, POKE VARPTR(X) through VARPTR(X)+4 with random numbers, and see if, by knowing the result of PRINT X, you can determine those four numbers. With the description I've given you, you should be able to do it.

One more time for that description: five bytes, power-of-two exponent first, 32 binary digits next, with the leftmost the sign bit, but considered to be a 1 for purposes of calculation. When you've got a good handle on the floating point representation, you're ready for the next session. Give it a try, enjoy it, and I'll talk to you next time.

Program #30, a BASIC program. Turn on the power of your Extended Color BASIC computer. When the cursor appears, type CLOAD and press ENTER. The computer will search (S) and find (F). When the cursor reappears, LIST this program. If the program is not similar to the listing, or if an I/O error occurs, rewind to the start of the program and try again. For severe loading problems, see the Appendix.

```
1 CLEAR500:DIMA(5),A$(5),B$(16):A=0:B=0:FORI=0TO15:READB$(I):NEXT
2 CLS:INPUT"NUMBER TO DISPLAY";A:PRINT:B=VARPTR(A):FORI=1TO5:A(I)=PEEK(B+I-1):NEXT
3 FORI=1TO5:A$(I)=HEX$(A(I)):IFA(I)(16THENA$(I)="0"+A$(I)
4 NEXT:PRINT"   FLOATING POINT STORAGE OF":PRINTTAB(9)A
5 PRINT:PRINTTAB(6)"$ ";:FORI=1TO5:PRINTA$(I)" ";:NEXT:PRINT
6 PRINT:PRINT"   BINARY REPRESENTATION OF":PRINTTAB(9)A:PRINT
7 IFA=0THENC$=STRING$(32,"0"):GOTO14
8 G=VAL("&H"+LEFT$(A$(2),1)):C=8OR G:G=G AND8:A$(2)=HEX$(C)+RIGHT$(A$(2),1)
9 FORI=2TO5:C=VAL("&H"+LEFT$(A$(I),1)):D=VAL("&H"+RIGHT$(A$(I),1))
10 C$=C$+B$(C)+B$(D):NEXT:E=A(1)-128:IFE(0THEN11ELSE12
11 E=-E:C$=".."+STRING$(E,"0")+C$:GOTO13
12 IFE(LEN(C$)THENC$=LEFT$(C$,E)+"."+RIGHT$(C$,LEN(C$)-E) ELSEC$=C$+"("+STRING$(E,"0")+".)"
13 IFB=0THEN5$="(-)"
14 C$=S$+C$:PRINTC$:PRINT:PRINT"  TOUCH (ENTER) TO CONTINUE"
15 DATA0000,0001,0010,0011,0100,0101,0110,0111,1000,1001,1010,1011,1100,1101,1110,1111
16 A$=INKEY$:IFA$=""THEN16ELSERUN
```

21.



You've arrived. This is the time for putting what you know about assembly language together with what you know about BASIC. I'll get right to it by reviewing how BASIC hands over control to a machine language program: through the USR function and via the EXEC call.

EXEC is used for execution after a load from cassette, but EXEC also is the simplest, and probably the best choice for including in a BASIC driver program for fast program speed. EXEC is a direct transfer of control to the machine language subroutine, where no attempt is made to pass along variable information. EXEC 12288, for example, departs from the BASIC program in progress and begins machine language execution at address 12288 (hex **$3000**). The Morse Code program, where the start and end of the message are fixed, and the Game of Life, which simply begins, need no more information than an execution address. In the Game of Life, EXEC 3200 provided the starting address (3200 is **$0C80**) and told BASIC to relinquish control to the machine language program. That was it.

EXEC stores the address you provide in a memory location accessible to BASIC, so that the next time you use EXEC, you need not specify an address; it will automatically use the last one until you change it. Also, the entire BASIC re-entry information is stored, so — unless you've messed around things in BASIC's direct page — all will be intact when your machine language program reaches its final RTS (return from subroutine).

That does bring up the question of BASIC's direct page, and in fact, where you can store the machine language programs you will be writing. The back of your BASIC and EDTASM+ manuals gives you some information about how your computer's operating parameters are organized; turn to page 63 of the EDTASM+ book. Pay careful attention to the opening information on page 63. The manual states that when BASIC is in use, the direct page register is pointed to page00; it notes that BASIC requires certain portions of this page, as well as quite a bit of page

Unless you have high hopes for creating full-scale commercial software, chances are you'll be using BASIC as your home base. A BASIC driver can be simple or complex, performing straightforward program executions, transferring variables back and forth, and creating graphics frameworks using BASIC's powerful drawing commands. No matter what your goal, knowing the relationship between BASIC and assembly language can speed your programming.

* The simplest command to execute a machine language program from BASIC is what?

EXEC.

* What information does EXEC require?

The starting address of the machine language program.

* EXEC&HA0D7 means what? (Try it!)

Begin executing a machine language program beginning at address $A0D7.

* What does "&H" mean in BASIC?

&H means hexadecimal.

* How does a machine language program get back to BASIC?

By executing RTS (return from subroutine).

* When BASIC is operating, where is the 6809's direct page register?

The direct page register is set to $00.

* What is a position independent program?

A program designed to run correctly no matter where it is located in memory.

* Which of these is a position independent command: LBRA TRAK1 or JMP TRAK1?

LBRA TRAK1 is position independent.

* What addressing mode is LBRA TRAK1?

Relative addressing.

* Which of these is a position independent command: LDX #TABLE or LEAX TABLE,PCR?

LEAX TABLE,PCR is position independent.

* What addressing mode is LEAX TABLE,PCR?

Relative addressing (specifically, program counter relative).

* What does LEAX -1,X mean?

Let X become X-1.

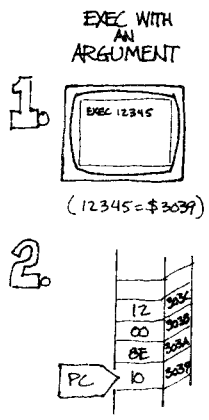
* What does LEAX $45,Y mean?

Let X become Y+$45.

01. Other portions here and there are marked, "can be used for machine-code programs."

Look at number 6 in the third column. Entitled "User Memory," this is described as "Total space for user machine-language routines. No space is reserved for this on start-up, but this can be reset by the CLEAR statement."

Now what about all this? How can you be confident that the program you place in memory will stay there? And that the program will be accessed as expected? How is memory protected from BASIC? What does "protected" mean? And how is your program protected from other machine-language programs?



Frankly, the answers to those questions depend on how you plan to use your software. If you're going to use only your own software, and use your software with BASIC as you see it described in these manuals, then you're safe. But . . .

. . . if you plan to use commercial software, such as special printer drivers or communications programs or math routines or whatever, you run the risk of having your program conflict with that program.

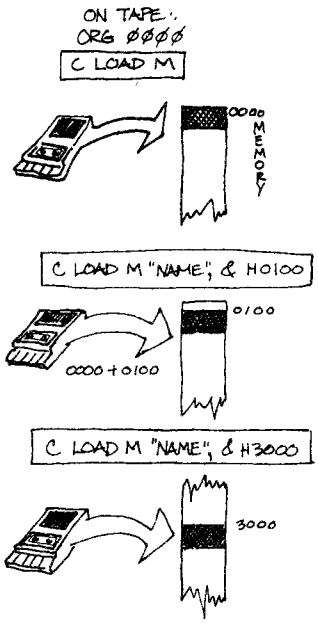
. . . if you plan to use a disk system, especially OS-9, the memory mapping of these devices may alter the area you plan to use.

. . . if you intend to write software to sell, you must expect that memory conflicts will arise with both other commercial software and the user's software — somewhat the converse of what I said earlier.

There are options. I'm not offering any business advice if you plan to sell your software, but I can recommend that you make your commercial machine-language programs position independent. Use the guidelines and approach as presented in session 19. If your program must be position dependent for reasons of speed or memory economy, then provide with it a relocator — a companion machine language routine that will automatically rework the program to fit in another area of memory. Beyond those recommendations, you're on your own as a software businessperson.

For most important programming, I'll stand by my position-independence or relocation recommendations. Let me tell you how to make position independence work for you on the Color Computer — after you've already written the position-intependent software, of course. Here's how:

In your assembly listing, place the origin at **$0000**. You can do that by specifically typing ORG **$0000** for reference, or by leaving out the ORG statement. The assembler assumes **$0000** if you don't specify otherwise. Save the source program to tape, and also assemble and save the object code.



When you want to load and run this machine language program, you use the command CLOADM. If the program's name is "Blurb", for example, you would normally type
CLOADM"BLURB". With position-independent programs, you need something more. You need an offset address, an address which is added to the CLOADMed address to produce a resultant location in memory where the program is going to be stored. For example, if the program is to load into memory beginning at 12288 (hex **$3000**), you would type CLOADM"BLURB",12288 or CLOADM"BLURB",&H3000. The program will add up your origin ($0000) and the offset address ($3000), and begin loading the program into that area of memory. Since your origin is **$0000**, then the offset address turns out to be the same as the loading address. And since the program is position-independent, it can start running as soon as you type EXEC.

I've got a very brief object program coming up. All this program does is load a short ASCII message into memory. The source program goes like this (you can glance in the book):

```
ØØ1ØØ          FCC   /THE MESSAGE IS HERE/
ØØ11Ø          END
```

After I assemble this, I will have nothing but a group of 19 bytes on the resulting tape. The lack of a specified origin means that the assembler will place these 19 bytes beginning at address **0000**. The program coming up next is dumped to tape ten times; its name is "TEXT". So if you want to see this message, what you want to type is CLOADM"TEXT",1024. This will load the message to the first space on the video screen. Then try any location on the screen and see where the subsequent nine messages come into view. Remember that the normal screen is mapped from **$0400** to **$0600** (1024 to 1535 decimal), so to see the whole message, your offset addresses should be in that range. So try these ten messages; I'll be back with a description of what memory is free and how to use it.

Programs #31A to 31J, object code programs. Turn on the power to your Extended Color BASIC computer. When the cursor appears, type CLOADM and press ENTER. The computer will search (S) and find (F). When the cursor reappears, type EXEC and press ENTER. The program will execute automatically. If an I/O error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix. For additional loading of programs 31B through 31J, refer to the text.

What you've just done is load blocks of binary information into memory. Since the binary information was saved to tape with a loading address of **$0000**, the offset addresses you specified in the CLOADM command became the actual loading addresses for the binary data.

* What cassette command loads BASIC programs?

CLOAD.

* What cassette command loads machine language programs directly into memory?

CLOADM.

* What does ORG mean?

Origin, that is, the first byte of a source listing.

* If ORG is left out of an assembly listing, where does the assembler begin assembly (the default ORG)?

At address $0000.

* If a machine language program named TESTER were ORGed at $0000, what BASIC command would load that program?

CLOADM or CLOADM"TESTER".

* If a machine language program named TESTER were ORGed at $0000, what BASIC command would load that program starting at $2000?

CLOADM"TESTER",&H2000 or
CLOADM"TESTER",8192

* What is the value appended to the CLOADM command called?

An offset address.

* Where is the normal video screen in the Color Computer memory map?

At locations 1024 through 1535 ($0400 through $05FF).

* Where do the high-resolution video screens begin on the Color Computer?

At location 1536 ($0600).

Low and graphics memory

* How much space does PCLEAR1 reserve?

1536 bytes ($0600 bytes).

* How much space is reserved by PCLEAR1 through PCLEAR8?

1536 through 12,288 bytes.

* If high-resolution graphics will not be used by BASIC, can machine language programs be stored in the area reserved by PCLEAR1?

Yes.

* How much space is reserved by PCLEAR1, and what are the addresses?

1536 bytes are reserved from $0600 to $0C00.

* What is the purpose of CLEAR?

To reset all BASIC variables.

* What is the purpose of CLEAR N, where N is a number?

To reset all BASIC variables, and to reserve N bytes for BASIC string manipulations.

* What is the purpose of CLEAR N,X where N and X are numbers?

To reset all BASIC variables, to reserve N bytes for BASIC string manipulations, and to protect memory from BASIC beginning at address X.

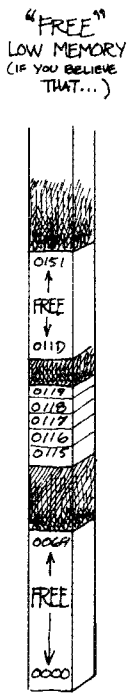
* What effect does CLEAR200,16384 have?

It resets BASIC variables, sets aside 200 bytes for BASIC string manipulations, and makes 16384 ($4000) the start of protected memory.

* What is protected memory?

Memory that is not available for BASIC's use.

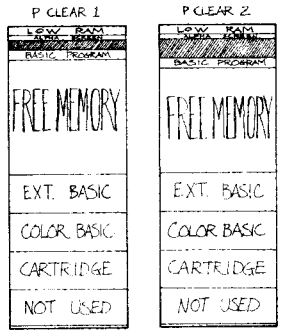
Such position-independent programs can, naturally, be moved as often as you wish. The next question, therefore, is where do you put the programs?



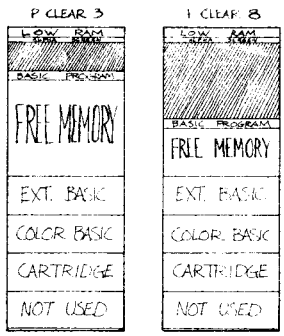
There are four places in your computer's read/write memory for convenient storage of machine language programs. You may store this binary data in low memory as provided in the memory map; in high memory protected from BASIC; in high-resolution graphics memory; and inside a BASIC program line. Each in turn now.

Storing a program in low memory is not safe from **$0000** to **$0069**. These are 106 bytes called "free", and there is also so-called free memory at **$0115** to **$0119** (five bytes), and a block of 53 bytes from **$011D** to **$0151**. None of this is safe for program storage; don't use it. The EDTASM book's phrase "can be used by machine language programs" means that you can store data here while your machine language program is underway. When you return to BASIC (especially if your machine language program is an integral part of a running BASIC program), the information BASIC needs in low memory is likely to be altered. However, with one of the excellent detailed Color Computer memory maps that have been published, you can learn how and when low memory is used by BASIC. So for now your rule of thumb about low memory is: don't.

There are three remaining options, and, with only a few reservations, all of these options are good ones. They are high memory, high-resolution graphics memory, and memory inside a BASIC program.

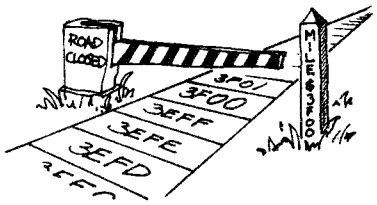


Storing programs in graphics memory is easy and reasonably safe. PCLEAR is BASIC's way of reserving high-resolution graphics memory, and PCLEAR1 is the smallest amount of graphics reserved memory allowed. PCLEAR1 allows 1,536 bytes of memory from **$0600** to **$0BFF** to be used for storing a machine language program. There are two major caveats to this process. Most obvious is the fact that you can't use high resolution graphics if you choose this method. Since this memory is intended to be a graphics screen, using any graphics command risks damaging the stored program. The other warning is the POKE often used on memory location 25, where the PCLEAR number of graphics pages are stored. Since PCLEAR0 is not allowed, a POKE to that location has been popularly used to free up some extra memory. But by doing that, you wipe out the graphics screen and the machine-language program along with it. So, in summary, use graphics memory for your program only if you can be sure BASIC will not be using high-resolution graphics commands.



The most popular mode of storing binary code is by placing it in protected high memory. BASIC is specifically set up to allow this use, and I consider it wisest to follow those recommendations when all other considerations seem equal.

Protecting high memory is an easy task. The BASIC CLEAR statement is used for this. The CLEAR statement performs three functions: CLEAR alone resets all BASIC variables and arrays; CLEAR followed by one number (as in CLEAR200) sets aside space for BASIC's string functions; and finally, CLEAR followed by two numbers (as in CLEAR200,14000) sets aside string space as well as creates a boundary beyond which BASIC may not trespass.



In the case of CLEAR200,14000, memory locations 14000 (that's hex **$36B0**) and above are not used by BASIC. It's as if your computer only had 14,000 bytes of memory instead of 16,384 for a 16K machine or 32,768 for a 32K machine. The only commands that will affect this memory are POKE (which can change any RAM location or output address) and CLOADM (which will attempt to load its data to the specified address whether memory is present there or not).

Here's how it works. If you have a 400-byte machine language program which you want to store in the high memory of a 16K computer, you first determine if the program is position independent. If you wrote the program, then you'll know for sure; otherwise, read the documentation for help. Assuming you know that the program will load and run in those top 400 bytes, you then subtract 400 from the highest memory location in the 16K computer. 16383 minus 400 is 15983. Then you'll need to determine if whatever BASIC program you're about to run will need more or less string space; the space allocated at power-up is 200 bytes. With that in mind, construct the CLEAR statement in the form CLEAR string space comma memory barrier. To protect 400 bytes and have the normal amount of string space, you would enter CLEAR200,15983.

Memory is protected and you are ready to CLOADM your machine language program or other binary data. For more information on CLEAR, use your BASIC manuals.

The final method of placing a binary program into memory is called the in-string or string-packing method. This technique was first popularized on the classic TRS-80 Model I, and remains a favorite for short, position-independent programs. Keep in mind that this isn't a universal technique; it expects certain features found only in Microsoft's dialect of BASIC.

To understand string-packing, you have to do a little rethinking about BASIC itself. The BASIC you're most familiar with is a programming language. You don't often think of a BASIC program as anything but what it represents. But a BASIC program is something different from what it represents... it is something that fills memory. And something that fills memory is binary information. A machine-language program is binary information. If both are binary information, can they coexist in a single listing?

```
* Can a machine language program
be stored in this protected
memory?

Yes.

* Once a machine language
program is in place, what
commands are used to execute the
program?

EXEC or USR.

* In the Color Computer, what
essential computer "matter" does
a BASIC program consist of?

Binary information.

* What does a machine language
program consist of?

Binary information.

* Microsoft created Color and
Extended Color BASIC. What
technique of storing a machine
language program can be used
with Microsoft BASIC?

String-packing.

* What is string-packing?

Placing a machine language
program inside a BASIC string,
within the BASIC program
itself.

* What information does BASIC's
VARPTR provide?

The address of a BASIC
variable.

* What three pieces of
information does VARPTR provide
about a string?

The variable name, the length of
the string, and the address of
the first byte in memory where
the string is located.

* What is the longest line that
can be typed in BASIC?

240 characters.
```

POKEing a string

* If a string variable name takes two bytes, an equal sign takes one byte, and the quotation marks take two bytes, how many bytes are left for the string itself?

235 bytes.

* What is the longest machine language program that can be stored in a BASIC string?

235 bytes.

* What three BASIC commands are essential for string packing?

VARPTR, PEEK and POKE.

* Why is VARPTR necessary for string packing?

VARPTR is needed to locate the address of the string's vital information in memory.

* Why is PEEK necessary for string packing?

After VARPTR provides the address of the string information, PEEK is used to determine the length and address of the string itself.

* Why is POKE necessary for string packing?

The instructions and data that make up the machine language program must be POKEd into memory where the string currently resides.

* Why are commands POKEd in place?

Because all 256 possible combinations from $00 to $FF cannot be typed from the keyboard.

* What two BASIC commands are used to run (access) a machine language program?

EXEC and USR.

Don't think about the source code now. Think about the binary code. And consider that BASIC has at least two situations in which it does not tamper with or interpret information as part of a program. In other words, there are two situations in which BASIC doesn't mess with what you type: after a remark (REM) statement and inside the quotation marks of a string variable.

One of these two situations is of special value. Recall the last session when I talked about floating-point arithmetic. I mentioned a BASIC command called variable pointer, or VARPTR. The command VARPTR points not only to floating-point numbers, it points to any variable. So type, enter and run this one-line program . . .

```
10 A$ = "THESE ARE THIRTY-ONE CHARACTERS"
```

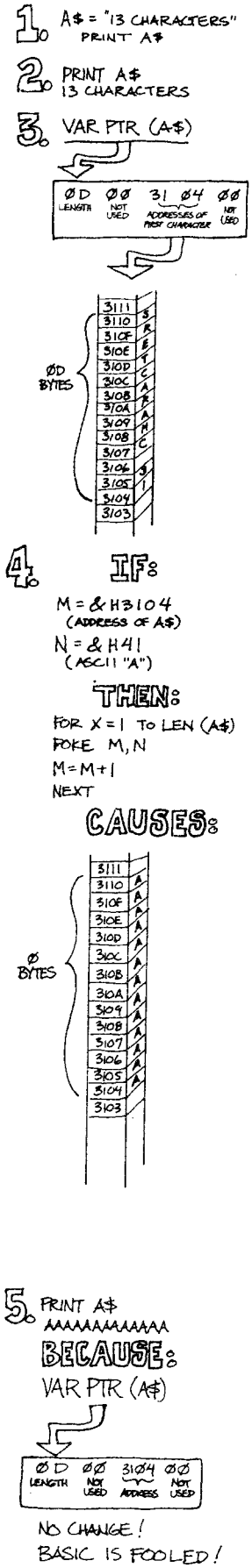
. . . you can then ask for VARPTR(A$). PRINT VARPTR(A$). The computer will report 7726. At memory location 7726 is information about A$, five bytes of it. In your reading of the Extended Color BASIC manual for the last session, you may recall that the first byte is the length of the string, and the third and fourth bytes are the address of the first character in the string.

There's the clue. The third and fourth bytes of this descriptive information are the address of the first character of the string. If you create a string of the correct length, and if you know where the string is in memory, and if you are confident that BASIC won't mess with the strings as they appear in program lines (it won't), and if — at last — you don't plan to use that variable for any other purpose within the program . . . then you can safely store a machine-language program within those quotation marks.

I wish you could just type such a program right into the string. You can't, of course, because you might need any one of the 256 possible bytes for your machine program, but only about 96 are typable from the keyboard. So you have to go in the back door.

The key to the back door is right there in the mailbox. You use the variable pointer VARPTR to find out where A$ sits in memory, you take a listing of your program in hexadecimal bytes, and you POKE them, one at a time, into the place occupied by A$. You could POKE bytes one at a time by hand, but there are easier ways. I have a program to show you the details; list it, but don't run it until I'm back with you.

Program #32, a BASIC program. Turn on the power of your Extended Color BASIC computer. When the cursor appears, type CLOAD and press ENTER. The computer will search (S) and find (F). When the cursor reappears, LIST this program. If the program is not similar to the listing, or if an I/O error occurs, rewind to the start of the program and try again. For severe loading problems, see the Appendix.



```
10 A=0:B=0:C=0:D=0:N=0:Q=0:X=0
20 B$=""
30 A$ = "14 CHARACTERS!"
40 A = VARPTR(A$)
50 B = PEEK(A+2)
60 C = PEEK(A+3)
70 D = B*&H100 + C
80 DEFUSR0 = D
90 FOR X = 1 TO LEN(A$)
100 READ B$
110 POKE D,VAL("&H"+B$)
120 D = D+1
130 NEXT X
140 DATA BD,B3,ED,8E,04,00,E7
150 DATA 80,8C,06,00,26,F9,39
160 STOP
170 INPUT Q
180 N = USR0(Q)
190 FOR X = 1 TO 500 : NEXT
200 GOTO170
```

LIST this program. Notice that A$ is 14 characters long. And notice that in lines 140 through 150 are a group of what look like hexadecimal numbers, presented as DATA statements, and following them is a short routine to read, convert and POKE them in place. The program, which you can hand-disassemble (that is, convert from hex to source code), simply fills the screen with any character you input. Look at these statements:

```
A = VARPTR(A$)
B = PEEK(A+2)
C = PEEK(A+3)
D = B * &H100 + C
DEFUSR0 = D
```

There are ten USR calls allowed by BASIC, USR0 through USR9, meaning you can have up to ten different machine language programs. DEFUSR identifies for BASIC the starting address of the machine-language program. In this case, the program is stored in A$, so variable A finds VARPTR(A$), and variables B and C obtain the two address bytes where A$ can be found. Since BASIC's workings are in decimal, you can't just dip in and pull out a 16-bit address; you've got to combine the most-significant byte with the least-significant byte to get a result. In case you hadn't thought of it this way before, you'll notice that in hexadecimal, the most-significant byte is always **$100** times the least-significant byte. So the resulting address is hex **$100** times B, plus C.

All of this could be combined into the complicated looking formula DEFUSR0 = &H100 * (PEEK(VARPTR(A$)+2) + PEEK(VARPTR(A$)+3). Now matter how you write it, it defines where the machine language program starts.

Now RUN the program; it will BREAK in 160. LIST the program, and have a look at A$. It looks longer now (it isn't) and seems to be garbage. Type and enter PRINT A$. A peculiar but different result.

```
* If a machine language program
is at $3000, use EXEC to access
it.

EXEC&H3000.

* What must be done before a
machine language program can be
accessed with USR?

The entry point must be
defined.

* What BASIC command is used to
define the USR entry point?

DEFUSR.

* How many USR entry points does
Extended Color BASIC offer?
What are they?

Ten entry points, USR0 through
USR9.

* What is the advantage of USR
in certain situations?

USR can transfer information to
the machine language program.

* If the machine language
program begins at $3000, define
the USR0 entry point.

DEFUSR0=&H3000
```

String packing

```
* If the information to be transferred to the program beginning at $3000 is 12345, give the USR0 command to access the program and transfer the information.

PRINT USR0(12345) or N = USR0(12345)

* Give the formula to define a USR9 entry point to a machine language program stored in N$.

DEFUSR9 = &H100 *
( PEEK(VARPTR(N$)+2 +
PEEK(VARPTR(N$)+3 )

* In string packing, what values should be avoided if possible? Why?

$00, because it is BASIC's end-of-line marker; and $22, because it is equivalent to an ASCII quotation mark.

* How does a machine language program get back to BASIC?

By executing RTS (return from subroutine).

* What is a position independent program?

A program designed to run correctly no matter where it is located in memory.
```

Recall many lessons past when I said that a single 8-bit word of memory had to serve many purposes. Now you see them. You have a machine language program stored inside A$'s quotation marks. When you print it, it looks like graphics and ASCII characters. When you list it, it looks like BASIC commands. What is it? It's still your machine language program, but the PRINT routine doesn't know that; the BASIC PRINT routine thinks it's a string to print. The BASIC LIST routine thinks you somehow stuffed commands inside the quotation marks. You can ponder that on your own; I'm getting back to the program.

You've already run this program, so just type and enter CONT for continue. The prompt asks you for a value from 0 to 255. Enter a value. The screen fills with your character and returns to BASIC; nothing new here. The machine language program is at work. Try more. Each time, the screen fills almost instantly.

Now tap BREAK, and LIST this program again. Notice in line 170 that the program inputs variable Q, and in line 180 the command N = USR0(Q) is encountered. The variable Q is passed to the machine-language program, which converts it and uses it to fill the screen.

There you have it: VARPTR used to find a string in a BASIC program, the machine-language program packed into that string, DEFUSR set to point to the start of the program packed into the string, and USR commanded to execute the program. One of the slickest methods ever devised.

Just a few warnings. First, be careful not to use the variable over again in the program. It won't erase where the machine language program is, nor its contents, but at some point either the BASIC or the machine language might end up misinformed about where things are.

Second, save the program before running it, and once you've run the lines containing A$ and all the POKEs, don't run past them again. Here's why: there are two hexadecimal values that can't appear in the string. One of them is **$22** and the other is **$00**. **$00** is used as BASIC's end-of-line pointer, so when it sees **$00**, it thinks it's reached the end of a BASIC line. Again, the program might end up misdirected. The value **$22** happens to be the ASCII value for a quotation mark, and more than two quotation marks will cause a dreaded syntax error.

The value **$00** is the opcode negate direct; you won't use that much. But **$22** is, unfortunately, PSHS, and that one's almost unavoidable. Have a look at this program, and when you're done, I have one more. Study them both before the next session.

Program #33, a BASIC program. Turn on the power of your Extended Color BASIC computer. When the cursor appears, type CLOAD and press ENTER. The computer will search (S) and find (F). When the cursor reappears, LIST this program. If the program is not similar to the listing, or if an I/O error occurs, rewind to the start of the program and try again. For severe loading problems, see the Appendix.

```
1 A=0:X=0:B=0:C=0
2 A$="THIRTY-FIVE CHARACTERS ARE NEEDED!!"
3 A=VARPTR(A$):C=256*(PEEK(A+2))+PEEK(A+3)
4 FORX=1TOLEN(A$):READB$:B=VAL("&H"+B$)
5 POKEX+C-1,B:NEXT:DEFUSR0=C
6 CLS:PRINT:PRINT"YOUR BORDER CHARACTER"
7 INPUT"ENTER A NUMBER FROM 0 TO 255";A
8 IFA<0ORA>255THEN7:ELSEM=USR0(A)
9 FORN=1TO1000:NEXT:GOTO6
10 DATABD,B3,ED,8E,04,00,86,21,E7,80,4A,26,FB,30,1F,86,0E,30
11 DATA88,1F,E7,80,E7,84,4A,26,F6,86,20,E7,80,4A,26,FB,39
```