Apple's Hidden Floating-Point Routines

Lightning-fast number crunching, HIRES graphics and Integer BASIC—all at once.

John Martellaro
2929 Los Amigos Ct., Apt. B
Las Cruces NM 88001

One of the nice things about having a microcomputer is that you can do high-speed mathematics. An Apple II, for example, does a nine-digit floating point division in AppleSoft in about 4 1/2 ms. There may be occasions, however, when you are either not satisfied with the speed of AppleSoft or cannot use it for some reason. If you owned any other computer, you would have to start looking for assembly-language programs that allowed you to work with floating point numbers. Not so with the Apple II.

A versatile, but little known, feature of the Apple II is the set of floating point routines—machine-language programs in ROM (starting at hexadecimal $F425) that do floating point arithmetic on four-byte floating point numbers. These routines have been given little attention because (1) little documentation is supplied with the machine other than the assembly-language listing on page 94 of the Apple II reference manual, and (2) the answer is difficult to decipher—the required normalization in a hex format makes the representation of the number extremely abstract. It is disconcerting to find that the number two is represented as 81 40 00 00.

Nevertheless, these routines can be extremely useful in at least two areas: high-speed number crunching, if you can interpret the result . . . that's what this article is about, and high-speed graphics, which requires some input, then a high-speed calculation of floating point numbers resulting in screen points that are integers. Then the Apple HIRES routine can be used to plot the points.

There is one catch: if your program uses assembly language only, then you are free to use the floating point (FP) routines as they are in ROM. However, if you wish to mix Integer BASIC and assembly language, you will have to reassemble the FP routines into RAM somewhere and change the working registers to somewhere very low in zero page of memory (for example, the "sweet 16" area). This is because Integer BASIC appropriates the area used by the FP routines ($F4 to $FB).

For high-speed graphics, I write an assembly-language subroutine that does all the number crunching using the floating point routines. Then I use the FIX function (explained later), which conveniently transforms the floating point number into a 2 byte signed two's complemented integer (also explained later), which happens to be the same format that Integer BASIC uses. Then, I can shove those two bytes into a memory location of some Integer BASIC variable (just as the HIRES routines do); I can now plot the point or print the value using the BASIC print statements. (See the Apple II reference manual, pp. 34-39, particularly Fig. 3 on p. 39.)

The upshot of all this is that you can now do high-speed number crunching with high-speed HIRES graphics under the control of a fast Integer BASIC! The possibilities opened up by this are unlimited. The capability of the Apple for simulation or real-time applications is enhanced considerably.

This article will give you an introduction to these floating point routines and their use.

The Floating Point Format

What follows contains a lot of strange-looking numbers. Do not let that upset you. That you have read this far means you are interested. If you read carefully, you'll find that everything is explained and that it isn't all that hard.

Actually, the format of the floating point numbers is an old idea; it is just new to today's microcomputerist. Four bytes are used to represent a number: one byte is used for the exponent and the other three form the mantissa. Since three bytes

Binary Exponent Written in Hex	Decimal Equivalent
00	−128
01	−127
.	.
.	.
.	.
7F	−1
80	0
81	+1
.	.
.	.
.	.
FE	+126
FF	+127

Table 1.

01111010	01000000	00000000	00000000	binary
122	64	00	00	decimal
7A	40	00	00	hex

Example 1.

contain 24 bits, you have 24 binary digits of precision.

In order to find out how many decimal digits this is, we write

$2^{24} = 10^N$

where N is an unknown to be found. We next take the log of both sides (natural or common, it doesn't matter) and use the equation $\log(a^b) = b \log a$.

$$\log (2^{25}) = \log (10^N)$$
$$24 \log 2 = N \log 10$$
$$N = \frac{24 \log 2}{\log 10}$$
$$= 7.2$$

So we can represent a number of about seven decimal digits with 24 binary bits. However, since we use floating point numbers, we can have a number with about seven digits of precision times an exponent. This allows us to store much larger (and smaller) numbers.

First, let's look at the exponent. With one byte for the exponent, we can represent 2^8, or 256, different numbers. These are the powers of 2, and the format is selected so that there is no need for a sign bit. Instead, the exponents are in the range of -128 to $+127$ (decimal) represented as 00 through FF (hex). That is, the smallest characteristic is 2^{-128} (about 10^{-38}) and the largest is 2^{+127} (again, about 10^{+38}). Table 1 should make this clear.

Next, the mantissa. The number is normalized so that the leftmost two bits are 01 for a positive mantissa and 10 for a negative mantissa. The binary point (BP) is considered to be just to the right of these two leftmost bits. For example, a positive number might be

01.XXXXX...

where the Xs are any combination of 1s and 0s. A negative number would look like

10.XXXXX...

Of course, the binary point is implied and not stored in the bit string. We keep track of it by the normalization.

This normalization process is similar to what we do for ordinary decimal numbers in scientific notation. Given

0.01×10^{-3} (base ten)

we can multiply the mantissa by 100 and divide the characteristic by 100 to keep the number unchanged:

$1. \times 10^{-5}$ (base ten)

Notice that no zeros were filled in because we really didn't know what was after the 1 in the first number. By convention, we fill in with trailing zeros.

A binary example of the normalization is

$0.001_2 \times 2^{-3}$ (base two)

(This should really be written:

$0.001_2 \times (10)_2^{-011}$

where $(10)_2 = 2_{10}$ in order to eliminate decimal numbers from the binary representation. Take your choice.) Again, we multiply by a suitable power of the base, in this case 2^3, to obtain:

$01.0_2 \times 2^{-6}$ (base two)

The mantissa is properly normalized since it starts with 01. the exponent is -6_{10}, which converts 7A. For presentation purposes, we use the convention that the leftmost byte is the exponent and the mantissa has the most significant byte on the left. This number, which is, by the way, 1/64 base ten, is represented as shown in Example 1.

So the floating point equivalent of 1/64 is 7A 40 00 00. For negative numbers, we would have two's-complemented the mantissa before normalization.

	FP1					FP2			
248	249	250	251	decimal address	244	245	246	247
F8	F9	FA	FB	hex address	F4	F5	F6	F7
X1	M1	M1 + 1	M1 + 2	assembly mnemonic	X2	M2	M2 + 1	M2 + 2
EXP	high		low	contents	EXP	high		low
	Mantissa					Mantissa		

Table 2. Floating point registers.

Function	Mnemonic	Number(s) In	Call Location	Result In
Negate	FCOMPL	FP1	F4A4	FP1
Add	FADD	FP1 + FP2	F46E	FP1
Subtract	FSUB	FP1 − FP2	F468	FP1
Multiply	FMUL	FP1 × FP2	F48C	FP1
Divide	FDIV	FP2/FP1	F4B2	FP1
Float	FLOAT	16 bit integer in M1 & M1 + 1 (M1 + 2 cleared)	F451	FP1
Integer	FIX	FP1	F640	16 bit integer in M1 & M1 + 1

Table 3. Floating point arithmetic routines.

The Floating Point Routines

Now that you see how the floating point numbers are represented, it's time to look at the routines that operate on them. The subroutines operate on one or two registers (depending on whether the function is monadic or dyadic) called FP1 and FP2. These reside in page zero of memory. Their location and contents are shown in Table 2.

To use the routines, load the proper registers with the floating point number(s) and call the proper routine. See Table 3 for

Glossary

Absolute Value: The value of a number without regard to the sign of the mantissa. ABS(−3) = 3.

Binary Point: The analog to the decimal point which indicates the separation between the zeroth power of the base and the −1 power. Hence, $10.1_2 = 2.5_{10}$

Characteristic: A leftover term from slide rules. It is useful to describe the value of $10^{\pm N}$ or $2^{\pm N}$.

Dyadic Function: A function that requires two arguments (inputs). Addition is a dyadic function.

Floating Point: A number representation that uses an N bit mantissa presumed to be multiplied by a characteristic.

Mantissa: The first multiplier of a number in scientific notation. In 2×10^6, the 2 is the mantissa; 10^6 is the characteristic; and 6 is the exponent.

Monadic Function: A function that requires only one argument (input). The log is a monadic function.

Normalization: The process whereby a number in scientific notation is adjusted so that the mantissa lies in a certain range. For decimal numbers, usually $1 \leq M < 10$; for binary, $1/2 \leq M < 1$.

Precision: A measure of the number of digits that can be represented. 1001 has four digits of precision. It is another name for significant digits.

Two's Complement: A method of representing negative binary numbers. First, all 0s are made 1s, and all 1s are made 0s. Then 1 is added. For example, 4 is 00000100. The two's complement is 11111100.

Zero Page: The first 256 memory locations addressable by the computer. Used extensively by the firmware.

Type	Then Hit	Comments
F4:83 60 00 00	RETURN	Put 12 in FP2
F8:82 40 00 00	RETURN	Put 4 in FP1
F4B2G	RETURN	Execute routine at F4B2
F8.FB	RETURN	Examine contents of FP1

Example 2.

If the Format Is	Then
1) 0.00XXX	Move BP right until you get 01.XXX; E = – RN
2) 0.01XXX	Move BP right two places to get 01.XXX; E = – 2
3) 0.10XXX	Move BP right one place to get 01.0XXX; E = – 1
4) 0.11XXX	Move BP right one place to get 01.1XXX; E = – 1

where RN is the number of places moved to the right.

Example 3.

If the Format Is	Then
1) 1.11XXX	Move BP right until you get 10.XXX; E = – RN
2) 1.10XXX	Move BP right two places to get 10.XXX; E = – 2
3) 1.01XXX	Move BP right one place to get 10.1XXX; E = – 1
4) 1.00XXX	Move BP right one place to get 10.0XXX; E = – 1

where RN is the number of places moved to the right.
(Note: Even though the number is less than 1, in two's complement, the leading 0 becomes a 1).

Example 4.

the format.

For example, to divide 12 by 4, hit RESET. If you have AppleSoft in ROM, be sure the switch is down, then hit RESET. Example 2 shows you what to type. You will then see 81 60 00 00, which is decimal 3.

The Algorithm

If you wish to write a program that utilizes these routines, you will have to have two tools: a short assembly-language program that will move data in and out of the FP registers and a precise algorithm for translating decimal numbers into the floating point format. As you will see, this is a tedious process for more than a couple of numbers. The algorithm is as follows:

1. Write down the number in decimal; call it N.

2. Convert to binary, ignoring the sign of the mantissa.

3a. If the absolute value of N is less than 1, then keep only the leftmost 24 significant digits, that is, ignoring leading zeros.

3b. If the absolute value of N is greater than 2^{24}, record the location of the binary point and keep only the leftmost 24 digits.

4. Eliminate the binary point after noting its location.

5. If the mantissa is negative, two's-complement it.

6a. If N is positive and ≥ 1, shift the binary point to the left (if necessary) until there is only one leading zero, that is, the leftmost bits are 01. (If necessary, add a leading zero.) Lop off any bits before the 01 and fill out to the right with zeros to make up 24 bits. The number of place shifts is the binary exponent + E.

6b. If N is positive and <1, then do one of the operations in Example 3.

6c. If N is negative and the absolute value of $N \geq 1$, shift the binary point to the left until the leftmost bits are 10 and add trailing zeros to make up 24 bits. The number of place shifts is the exponent + E. (Exception: If all 1s are to the left of the decimal, shift the binary point to the right one place. Lop off any leading 1s prior to the 1X and add trailing zeros to make up 24 bits. The exponent E = – 1.)

6d. If N is negative and the absolute value of N<1, then do one of the operations in Example 4.

7. Convert the three groups of 8 bits to decimal (as an intermediary, if you wish) then to hex, ignoring the binary point.

8. The exponent is $80_{16} + E_{16}$. See Examples 5 and 6.

After you have done this for a number such as 2.371256×10^{-26}, you will wish there were a way to get the Apple itself to do the work. I thought the same way myself after I had to do several of them, so I wrote an AppleSoft program to do it for me. (Ideally, it would be an assembly-language program, but that's a lot of work. And the AppleSoft program gives the answer in a few seconds.)

This program accepts either decimal input and outputs the floating point format or accepts the floating point format and generates the decimal number. It runs in 5.8K, so if you only have AppleSoft II on cassette, you'll need a 20K or larger machine. The program is available on cassette for $7 ppd.

There is one additional note. Just as Integer BASIC appropriates zero page for its use, so does AppleSoft—so you cannot call these routines from AppleSoft. You could shuffle data in and out of the registers preserving their status before and after each call, or, as mentioned before, you could reassemble the FP routines into RAM and have

Number (Absolute Value)	Floating Point (Positive)	Representation (Negative)
0	00 00 00 00	00 00 00 00
10^{-10}	5E 6D F3 7F	5E 92 0C 81
10^{-5}	6F 53 E2 D6	6F AC 1D 2A
.01	79 51 EB 85	79 AE 14 7B
.2	7D 66 66 66	7D 99 99 9A
.25	7E 40 00 00	7D 80 00 00
.75	7F 60 00 00	7F A0 00 00
.9	7F 33 33 33	7F 8C CC CD
1	80 40 00 00	7F 80 00 00
2	81 40 00 00	80 80 00 00
3	81 60 00 00	81 A0 00 00
4	82 40 00 00	81 80 00 00
5	82 50 00 00	82 B0 00 00
6	82 60 00 00	82 A0 00 00
7	82 70 00 00	82 90 00 00
8	83 40 00 00	82 80 00 00
9	83 48 00 00	83 B8 00 00
10	83 50 00 00	83 B0 00 00
10^5	90 61 A8 00	90 9E 58 00
10^{10}	A1 4A 81 7C	A1 B5 7E 84

Table 4. Some floating point numbers. Note that the positive and negative numbers should add to zero, which they do.

Step Number	Result
1.	+ 12
2.	1100.
6a.	01.10000000000000000000000
	E = 3
7.	01100000 00000000 00000000
	96 00 00
8.	Exponent = 83

The number is 83 60 00 00 when converted to hex.

Example 5. Using + 12.

Step Number	Result
1.	– 12
2.	1100.
5.	. . . 1111111110100
6c.	10.10000000000000000000000
	E = 3
7.	01100000 00000000 00000000
	160 00 00
8.	Exponent = 83

The number is 83 A0 00 00 when converted to hex.

Example 6. Using – 12.

them use different zero page registers. There is room in zero page to do this with Integer BASIC, but precious little with AppleSoft. I haven't done it yet.

Also, AppleSoft does not use these routines for its own arithmetic. The reason for this is that Microsoft BASIC uses a *five* byte floating point number in order to obtain nine-digit precision. This came along after the FP routines were written. Furthermore, the numbers are stored in those five bytes in a different format (ASCII) than that used by the FP routines.

Conclusion

The floating point routines are useful for the assembly-language programmer. There are special occasions when the difficulty of their use is secondary to the advantage of an assembly-language program with seven digits of precision.

This article has shown how to get the numbers into the required format. (However, you should not try to use the routines extensively without consulting the references on error branching and index register use.) Finally, Table 4 lists some floating point numbers for you to practice on with the algorithm and the routines. ■

References

1. Roy Rankin and Stephen Wozniak, "Floating Point Routines for the 6502," *Interface Age*, No. 12, pp. 103-111 (Nov. 1976).
2. Roy Rankin and Stephen Wozniak, "Floating Point Routines for the 6502," *Dr. Dobbs Journal*, No. 7, pp. 17-19 (Aug. 1976).
3. Stephen Wozniak, "Floating Point Package," *The WOZPAK*, ch. 13, Apple Computer, Inc.
4. Don Williams, "Linkage Routines for the Apple II Integer BASIC Floating Point Package," *Peeking at Call A.P.P.L.E.*, Vol. 1 (Apple Puget Sound Program Library Exchange, 6708 39th Ave. SW, Seattle WA 98136).
5. Arpad Barna and Dan Porat, *Introduction to Microcomputers and Microprocessors*, John Wiley, 1976, pp. 34-44.