

Using Complex Numbers in an Apple][Computer

Historical Introduction to Applesoft Basic

The Applesoft BASIC interpreter was licensed from the Microsoft 6502 BASIC which was a popular language before Pascal became available. Microsoft was producing BASIC interpreters for nearly every microprocessor that was being produced in 1975 and 1976. It was calculating on licensing or selling their BASIC interpreter to any company who built a computer around any of those early microprocessors. Richard Weiland, Bill Gates, and Monte Davidoff at Microsoft ported their Intel 8080 BASIC in mid-1976 to the new 6502 microprocessor even though there were no computers being marketed at that time that utilized that particular microprocessor. The tool that these early entrepreneurs used for this software port was a 6800 microprocessor simulator that was written by Marc McDonald, Microsoft's first employee. McDonald was able to accomplish this port because the 6800 had an instruction set that was very similar to the instruction set of the 6502 microprocessor. In August, 1977, Apple made a \$10,500 payment to Microsoft which was the first half of a flat-fee license that Apple and Microsoft were able to negotiate. Microsoft would typically license its BASIC on a royalty basis and they would be paid a set fee for every copy of BASIC that was utilized. In this case, a fee would be paid for every computer that Apple sold. The fact that Microsoft was willing to concede and let Apple license their 6502 BASIC interpreter on a flat-fee basis is a reflection of the financial straits that Microsoft was currently facing.

The Microsoft 6502 BASIC interpreter that Apple purchased was Version 1.1. When Apple received the interpreter, it required the work of several talented individuals in order to correct errors in the original source code and to incorporate unique LORES and HIRES graphic commands. Randy Wigginton and Cliff Huston were both instrumental in this effort. The first manual for Applesoft I was published in November, 1977, and the second manual for Applesoft II, the final version, was published nearly a year later in August, 1978. Applesoft II contained further code changes that had already been incorporated into Microsoft 6502 BASIC Version 2. This final version of Applesoft in ROM has been virtually unchanged in the Apple][computer as it evolved from the Apple][Plus to the Apple //e to the Apple //c. Beginning with the Apple //c, however, slight modifications were made to the ROM code in order for the interpreter to accept the input of statements in lowercase. Some of these modifications were folded into the Applesoft interpreter that was included in the Enhanced Apple //e. The eight year license that Apple purchased for Microsoft 6502 BASIC expired in 1985. In order to obtain a second eight year license for Microsoft 6502 BASIC, Apple exchanged their software code for MacBASIC and simply gave it to Microsoft. This second and final license from Microsoft for Microsoft 6502 BASIC expired in 1993.

Definition of Real (Floating Point) Variables in Applesoft

Applesoft BASIC (Applesoft hereafter) stores and processes floating point numbers in memory and it uses five bytes or forty bits for that number. Applesoft uses one byte or eight bits for the exponent that includes a bias of 128 and four bytes or thirty-two bits for its signed mantissa as shown in Table 1 for a Real Number. Real Numbers are stored in a designated area of memory along with Integer Numbers and Simple Character String variables. Another area of memory stores multi-dimensioned array variable descriptors for the arrays of Real Numbers, the arrays of Integer Numbers, and the arrays of Character String variables. Table 2 shows an example for each of these three array variable descriptors that has two dimensions. Successive Array Element dimension sizes **precede** each other with the first-dimension size in **high/low** byte order always coming **last**. The Array Variable descriptor grows in size as the number of dimensions increase in value. The nominal size of an Array Variable descriptor is seven bytes for a single dimension array. The descriptor increases in size by two additional bytes for each added dimension. Therefore, the

dimension value that is found in Byte 5 of the Array Variable descriptor becomes a critical piece of information that is used to calculate where the Array Elements begin and where the Array Elements end relative to the address of their Array Variable descriptor. The maximum number of dimensions for an Array Variable descriptor is 255 since this variable is limited to an 8-bit quantity.

Variable Type	Byte Definitions						
	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Real Number	name1 +ASCII 65	name2 +ASCII 66	Exponent	Mantissa Byte 1	Mantissa Byte 2	Mantissa Byte 3	Mantissa Byte 4
Integer Number	name1 -ASCII 195	name2 -ASCII 196	High Value	Low Value	0	0	0
Simple Character String	name1 +ASCII 69	name2 -ASCII 198	String Length	Low Address	High Address	0	0

Table 1. Simple Variable Descriptor Definitions in Applesoft

Variable Type	Byte Definitions								
	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9
Real Array	name1 +ASCII 65	name2 +ASCII 66	Low Byte Offset	High Byte Offset	Number of Dimensions K	Size of Kth Dim High Byte	Size of Kth Dim Low Byte	Size of K-1 Dim High Byte	Size of K-1 Dim Low Byte
Integer Array	name1 -ASCII 195	name2 -ASCII 196	Low Byte Offset	High Byte Offset	Number of Dimensions K	Size of Kth Dim High Byte	Size of Kth Dim Low Byte	Size of K-1 Dim High Byte	Size of K-1 Dim Low Byte
Character String Array	name1 +ASCII 69	name2 -ASCII 198	Low Byte Offset	High Byte Offset	Number of Dimensions K	Size of Kth Dim High Byte	Size of Kth Dim Low Byte	Size of K-1 Dim High Byte	Size of K-1 Dim Low Byte

Table 2. Array Variable Descriptor Definitions in Applesoft

Element Type	Byte Definitions				
	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
Real Number Element	Exponent	Mantissa Byte 1	Mantissa Byte 2	Mantissa Byte 3	Mantissa Byte 4
Integer Number Element	High Value	Low Value			
Character String Element	String Length	Low Address	High Address		

Table 3. Single Array Element Descriptor Definitions in Applesoft

Bytes 3 and 4 of the Array Variable descriptor give the offset in bytes to the beginning of the next, if any, Array Variable descriptor relative to the address in memory where the current Array Variable descriptor is located. The Array Elements that belong to an Array Variable descriptor begin immediately after the descriptor whose descriptor size can be calculated knowing the value in Byte 5, or $5 + (\text{value in Byte 5}) * 2$. The definition of each Array Element for each type of Array Variable descriptor is shown in Table 3. These Array Element definitions are essentially the same as the definitions for the respective Simple Variable descriptors without including the name of the array variable. Obviously, the name for all of the Array Elements is the same, and this name is found only in its Array Variable descriptor. Each Array Element for arrays of Real Numbers is five bytes in size and the Array Element contains a 1-byte exponent and its 4-byte mantissa for a Real Number.

Unlike other floating point number notations such as IEEE 754, the Applesoft exponent utilizes all eight bits for its value and it utilizes the most significant bit in its mantissa for the sign bit, and if that bit is OFF, the respective floating point number is positive. The bias that is included in the exponent is utilized to represent very large and very small numbers. As in other floating point number notations, the mantissa utilizes an implicit high-order one bit to yield a full 32-bit significand. An Applesoft floating point number typically provides a numerical range from 10^{-38} to 10^{+38} and it has, at most, nine digits of accuracy. When using floating point numbers in Applesoft, those numbers must be within this numerical range or Applesoft will flag an error or simply convert the number to zero. Applesoft understands scientific notation when a floating point number is either too small or too large to express that number in decimal form. The format of Applesoft scientific notation is $SD.FFFFFFFFESTT$ for an Applesoft floating point number. Both the single digit decimal number D and the double digit exponent TT utilize the sign bit S . If the floating point number is positive, no plus $+$ sign is used before that single digit D . However, the sign of the exponent TT is always expressed in Applesoft scientific notation whether the exponent is positive or negative. The letter E separates the decimal number D and its fractional part $FFFFFFF$ from its exponent TT . Applesoft does not identify IMAGINARY floating point numbers differently from REAL floating point numbers. And, Applesoft does not provide for either double precision Integer Numbers or double precision floating point numbers whether they are REAL or IMAGINARY.

Deficiencies in Applesoft Routines That Process Floating Point Numbers

Applesoft mathematical routines and functions that operate on very small floating point numbers can become problematic. These routines/functions may exhibit non-commutative addition, non-commutative multiplication, non-reflexive equality evaluation, irregularities of the exponent when the exponent is very small or very large, errors in the multiplication algorithm, errors in the binary to decimal conversion, and significant errors in the trigonometric functions that involve very small arguments. Some intermediate arguments depend on a full 40-bit significand since these arguments utilize a guard byte. On the other hand, some intermediate arguments are rounded and they are pushed onto the stack using only their 32-bit significand. Rounding consists of simply inspecting the most significant bit of the guard byte and if that bit is set, the thirty-two bit significand is incremented. When addition, subtraction, or multiplication is initiated, only one operand uses a full forty bit significand and the other operand uses its thirty-two bit significand. In division, only the quotient has any extra significance with two additional bits. Sticky bits are not utilized in Applesoft mathematical routines and functions in order to make numerical rounding decisions. Since the cosine and tangent trigonometric functions depend solely on the sine function, they are equally flawed if not more so. The Applesoft mathematical routines and functions can provide acceptable results if very small or very large arguments are avoided and if the number of significant digits are limited to only what is acceptable given the total range of the numerical values for all arguments.

Applesoft arithmetic does contain known irregularities, some in which the user should not be expected to anticipate. Such irregularities may occur intentionally because certain decisions were made while

designing the arithmetic algorithms. Other irregularities may occur unintentionally because of coding errors or software mistakes. Non-commutative addition means that different results are obtained when the positions of the variables being added are exchanged. Non-reflexive equality means that different evaluations are obtained when the positions of the variables being compared are exchanged. When the exponent of a very small number is equal to -128, a positive quotient will be obtained without regard to the sign of the divisor or the sign of the dividend. When two consecutive variables are nearly zero and they are multiplied, their product is shifted to the right one extra bit. Non-communicative multiplication issues are also confounded by decimal to binary to decimal conversions where an identity might be expected but is simply not obtained. Unless a Taylor series is utilized that has at least thirteen to fifteen iterations, the Applesoft `sine` function exhibits extremely poor accuracy for angles that are near zero. Thus, Applesoft generates zero for all arguments that are greater than $0.5 * 10^{10}$. Apparently, the flaw in the Applesoft `sine` routine for an argument that is very small in value is due to its argument reduction algorithm. And, as already mentioned, the `cosine` and `tangent` trigonometric functions are equally flawed since they are obtained by means of trigonometric identities. Therefore, it is vital that the engineer or mathematician is aware of all of these physical limitations that affect the accuracy of Applesoft arithmetic. And, the engineer or mathematician must accommodate all of their complex floating point variables, arrays, determinants, and inverse arrays for these Applesoft arithmetic irregularities.

Previous Software Publications That Utilize Complex Numbers

S. Keller and L. D'Addario published *A Library of Binary Subroutines for Apple II Plus* and *Complex Math Package for Apple II Plus Computers*, both reports in January, 1982. These publications were Internal Report No. 225 and Internal Report No. 226, respectively, from the Electronics Division of the National Radio Astronomy Observatory in Charlottesville, Virginia. *A Library of Binary Subroutines for Apple II Plus* publication includes an Introduction, Memory Organization, The Pointer Reset Program, The LINK Subroutine, Calling Library Subroutine by Name, and Library Versions 2.0 and 2.1. Figures are provided for Memory Configuration of the Apple II Plus with Binary Library and Configuration of Lower Memory. Tables are provided for Contents of Library Version 2.0 and for Version 2.1. *Complex Math Package for Apple II Plus Computers* publication includes an Introduction, Calling Sequence, Descriptions of Individual Subroutines, Memory Requirements and Addresses, Timing, and Examples and Suggestions. Tables are provided for Entry Point Addresses in Binary Library Version 2.0 and for Subroutine Execution Times. When I inquired about the availability of any further information on Reports No. 225 and No. 226 and perhaps the source code for the complex math subroutines, I have yet to receive a response from the National Radio Astronomy Observatory in Charlottesville, Virginia.

The NRAO complex math package is to be treated as an extension to Applesoft and this package includes functions for adding, multiplying, dividing, and converting coordinate systems that utilize complex floating point numbers. The complex math package can be used to add, subtract, multiply, divide, and invert 2-by-2 arrays that utilize complex floating point numbers. The assembly language routines are accessed by means of a `CALL` statement that utilizes a tailored format in order to allow the passing of floating point variables or 2-by-2 arrays to the various processing algorithms. The reports state that the processing time for these assembly language routines are two to eight times faster than if the equivalent routine is written entirely in Applesoft. Understand that these routines operate on two Applesoft floating point numbers that comprise a complex floating point array. If the same routines utilize Applesoft, twice as many or more operations would be necessary in order to evaluate the same complex floating point arrays. By necessity, a complex floating point variable is comprised of a `REAL` floating point number and an `IMAGINARY` floating point number where both are contained in a single floating point array that has two floating point numbers. Operations on 2-by-2 complex floating point arrays must be handled uniquely because they involve specific operations on the array that contains complex floating point variables. The inversion of a 2-by-2 complex floating point array is even more difficult when the array contains complex

floating point variables. Just to mention a few examples, the conversion of polar coordinates, in-phase/quadrature measurements, current lag in inductive loads, and voltage lag in capacitive loads all require complex mathematical solutions. It is indeed a shame that *A Library of Binary Subroutines for Apple II Plus* and *Complex Math Package for Apple II Plus Computers* appear to be unavailable to the Apple][enthusiast. I suspect that this might be a very interesting project to attempt and recreate. It might be quite difficult! But it certainly would be very educational and a whole lot of fun, indeed!

Memory Page	Bank 2 Description	Bank 1 Description
0x00	Page-zero variables, pointers, routines, and special addressing modes	
0x01	Stack for the 6502-microprocessor	
0x02	INPUT buffer, Applesoft interpretation buffer	
0x03	User buffer, DOS interface routines and vectors	
0x04:0x07	TEXT or LORES graphics Page 1	
0x08:0x0B	Applesoft program start, TEXT or LORES graphics Page 2, or available for software	
0x0C:0x0F	HIRES Graphics Character Generator, or available for software	
0x10:0x1F	Available for software	
0x20:0x3F	HIRES graphics Page 1, or available for software	
0x40:0x5F	HIRES graphics Page 2, or available for software	
0x60:0xAF	Available for software	
0xB0:0xB7	DOS 4.5.06 HIMEM, CXMATH Complex Floating Point Arithmetic Package	
0xB8:0xBD	Available for software	
0xBE:0xBF	DOS 4.5.06 Language Card partition software interface, DOS 4.5.06 bootstrap routines	
0xC0	System Soft Switches	
0xC1:0xC7	Peripheral-card ROM memory for slots 1-7, or CX ROM	
0xC8:0xCF	Peripheral-card expansion ROM memory for slots 1-7, or CX ROM	
0xD0:0xDF	RAM DOS 4.5.06 Command and File Managers; ROM Applesoft Interpreter routines	RAM DOS RWTS, HELP
0xE0:0xE6	RAM DOS 4.5.06 Command and File Managers; ROM Applesoft Interpreter routines	
0xE7:0xEC	RAM DOS 4.5.06 working variables and Workarea buffer; ROM Applesoft Interpreter routines	
0xED:0xF7	RAM DOS 4.5.06 file buffers; ROM Applesoft Interpreter routines	
0xF8:0xFF	RAM Monitor routines that are dynamically modified; ROM Monitor routines	

Table 4. Apple][Memory Utilization with DOS 4.5.06 Installed

Designing The Complex Floating Point Math Package For The Apple II Computer

The purpose of this paper is to document the progress and the solutions that I have developed in order to design and formulate a Complex Floating Point Math Package for the Apple II computer that is based on Report No. 225 and Report No. 226 from the National Radio Astronomy Observatory in Charlottesville, Virginia. These two NRAO reports contain enough information to begin such an interesting project for the Apple][computer enthusiast. I have made extensive use of the internet in order to more fully

understand complex numbers, arrays of complex numbers, and how complex numbers can be manipulated. I have taken many of the design features that are contained in these two reports under consideration and I have developed my own design features. The initial considerations for the practical utilization of a Complex Floating Point Math Package must begin with memory organization. Both NRAO reports suggest that the primary processing software is one of several various Applesoft programs. Applesoft is certainly an excellent choice for software utilization on an Apple][computer when it is used wisely for its best functionality. This Applesoft functionality would include data input from the user, data obtained through RS232 serial communication lines, and data read from the Disk][. Applesoft functionality would also include data output to the TEXT screen, data output to the LORES screen, data output to the HIRES screen, and data written to the Disk][. Furthermore, Applesoft is very useful for data organization, program structure, processing documentation, and the organization of processing procedures. However, Applesoft is not very useful in performing intensive complex floating point mathematical operations. An initial Apple][memory utilization with DOS 4.5.06 installed in memory is shown in Table 4.

The memory utilization that is shown in Table 4 provides for a large number of options vis-à-vis a few assumptions. It is assumed that my Complex Floating Point Math Package CXMATH is installed and initialized at $0xB000$. This initialization will change the address in FRETOP at $0x6F:0x70$ from the value given in HIMEM from $0xBE00$ to $0xB000$. The NARO placed its math package at $0x1400:0x1DBA$. I believe this choice reduced the options in where to best locate its Applesoft program. If the HIRES Graphics Character Generator that is discussed in Report No. 225 is not required and HIRES graphics Page 1 is utilized by the Applesoft program, a smaller sized Applesoft program can be placed at its normal location in memory at $0x0801$ and the program can extend to $0x1FFF$. Rather, if HIRES graphics Page 2 is utilized, the Applesoft program can extend to $0x3FFF$. On the other hand, if the HIRES Graphics Character Generator is required and HIRES graphics Page 1 is utilized, then specific page-zero addresses can easily be modified in order to allow an Applesoft program to load and process starting at memory address $0x4000$. This Applesoft program must allow sufficient memory for its variables and string pool data below FRETOP which has already been set to $0xB000$. Of course, HIRES graphics Page 2 can be utilized rather than HIRES graphics Page 1 and the Applesoft program can load and process starting at memory address $0x6000$ if the $0x2000:0x3FFF$ memory space must be utilized for other software functionality.

Mathematical accuracy as well as speed of calculation should be the primary drivers for the Complex Floating Point Math Package routines. Thus, I spent some time investigating whether my own floating point multiply and divide algorithms are competitive with the Apple ROM routines. What I discovered was that the ROM routines were substantially faster than my routines in that the ROM routines require far fewer instruction cycles in order to process the same floating point data. I wanted to know the differences between the ROM routines and my routines which produce identical results, even for the guard byte. I was ready to take a deep dive into Applesoft ROM software. Bear in mind, these mathematical algorithms were originally designed for the Intel 8080 microprocessor and they were ported to the 6502 microprocessor some fifty years ago using a 6800 microprocessor. So, how does the Apple ROM multiply and divide floating point numbers so efficiently? I am very fortunate to have disassembled and document the 6502 assembly language software that came installed in the ROM of my Enhanced and Platinum Apple //e computers. It is this ROM that I have removed, added, and modified a number of software modules that transformed the original ROM into a far more powerful Apple //e ROM in my most humble opinion. Section II of my publication of the *DOS 4.5 Volume and File Disk Management System Second Edition* precisely details this ROM transformation. I have also made use of a number of references and other publications in order to assist me in documenting many of the software modules, routines, and variables that are utilized in this ROM. Table 5 highlights a number of Applesoft floating point routines that can be incorporated in a Complex Floating Point Math Package. Table 5 also provides a short description for

each routine. As I have come across one detail after another or as I have developed a new conversion tool, I have added that new information or those new results into my ROM source code. Every new bit of information that I have collected, generated, and added to this ROM source code has assisted me in understanding the reasoning and the rationale that is behind many of the ROM floating point algorithms.

Name	Address	Description
FSUB	0xE7A7	Call LOADARG; make FAC negative; fall into FADD.1
FADD	0xE7BE	Call LOADARG; add ARG to FAC
FP1.0	0xE913	Location of the floating point value for 1.0
FPN0.5	0xE937	Location of the floating point value for -0.5
FPLN2	0xE93C	Location of the floating point value for LN(2)
FLOG	0xE941	Take the natural logarithm of FAC leaving value in FAC
FMULT	0xE97F	Call LOADARG; multiply FAC by multiplier in ARG leaving value in FAC
LOADARG	0xE9E3	Use INDEX to load ARG from a memory address; initialize XORSIGN
FMUL20	0xEA39	Fast routine to multiply FAC by ten using addition
FDIV	0xEA66	Call LOADARG; process GUARD1; divide FAC as divisor in ARG leaving value in FAC
LOADFAC	0xEAF9	Use INDEX to load FAC from a memory address; initialize GUARD1
COPYFAC	0xEB2B	Call RNDUP; use INDEX to copy FAC to a memory address
MOVFA	0xEB53	Copy ARGSIGN to FACSIGN; copy the value in ARG to FAC; initialize GUARD1
MOVAF	0xEB63	Call RNDUP; copy the value in FAC to ARG; initialize GUARD1
RNDUP	0xEB72	Process the MSB of GUARD1 to roundup the value of FAC
FSGN	0xEB90	Evaluate FACSIGN and FACMANT for $0 <, =0, >0$
FPCOMP	0xEBB2	Compare supplied floating point number to FAC; return 0xFF, 0x00, or 0x01
FP2INT	0xEBF2	Remove all fractional bits to convert FAC mantissa to an integer value
FPRINT	0xED34	Convert FAC to ASCII base-10 digits with decimal and/or scientific notation
FP0.5	0xEE64	Location of the floating point value for 0.5
FSQR	0xEE8D	Call MOVAF; call LOADFAC with FP0.5; fall into FEXP to compute square root
FPWRT	0xEE97	Raise ARG to the power in FAC leaving value in FAC; call COPYFAC using TEMP3
FEXP	0xEF09	Raise e to the power in FAC leaving value in FAC
FCOS	0xEFEA	Call FADD with PIDIV2; fall into FSIN
FSIN	0xEFF1	Call MOVAF with PIMUL2; compute the sine of FAC in radians leaving value in FAC
FTAN	0xF03A	Compute the tangent of FAC in radians leaving value in FAC
PIDIV2	0xF066	Location of the floating point value for PI/2
PIMUL2	0xF06B	Location of the floating point value for PI*2
FP.25	0xF070	Location of the floating point value for 0.25
FATAN	0xF09E	Compute the arctangent of FAC leaving value in FAC; range $-\pi/2$ to $\pi/2$ radians

Table 5. Selected Applesoft Floating Point Routines

All of the Applesoft floating point routines that are included in Table 5 utilize page-zero variables, pointers, and even a character read routine. Table 6 includes many of these page-zero variables and pointers that are used by these Applesoft floating point algorithms. Table 6 also gives a short description for each page-zero entry. Using page-zero variables certainly accelerate algorithm processing in that fewer microprocessor cycles are needed and the size of the ROM algorithm in bytes is reduced. Also, the

floating point algorithms depend upon many pointers that are used to copy floating point numbers from one memory location to another memory location and to similarly process each of the bytes that are contained in the mantissa of a floating point number. The routines MOVFA and MOVAF are the exception in that these routines use a fast register copy loop rather than the more expensive byte-by-byte processing method which can manipulate data in a far more accelerated fashion. For example, the routine COPYFAC copies each of the bytes of the floating point number individually using the pointer INDEX rather than using a fast register copy loop. This routine, as written, requires forty bytes of code and it executes in 94 µsecs. If a fast register copy loop is utilized instead, this routine would require only thirty-three bytes of code and it would execute in 133 µsecs. If rewritten, COPYFAC takes 41% more time to execute in 17% less code. COPYFAC is called only once in order to transfer the final numerical value of a series of floating point calculations to its final floating point destination. In this instance, I find it easy to justify spending 39 µsecs more if I can save seven bytes of ROM space in order to implement a far better ROM feature.

Name	Memory	Size	Description
INDEX	0x5E:0x5F	2 bytes	Pointer to copy floating point values to/from FAC/ARG
LASTMUL	0x62:0x65	4 bytes	Last multiply or divide result
MULGUARD	0x66	1 byte	Guard byte for LASTMUL (not implemented)
TEMP3	0x8A:0x8B	2 bytes	Pointer to copy floating point values to/from FAC
ARGUARD	0x92	1 byte	Guard byte for ARG
FACEXP	0x9D	1 byte	FAC floating point exponent
FACMANT	0x9E:0xA1	4 bytes	FAC floating point mantissa
FACSIGN	0xA2	1 byte	FAC floating point sign, extracted
ARGEXP	0xA5	1 byte	ARG floating point exponent
ARGMANT	0xA6:0xA9	4 bytes	ARG floating point mantissa
ARGSIGN	0xAA	1 byte	ARG floating point sign, extracted
XORSIGN	0xAB	1 byte	FACSIGN EOR ARGSIGN
FACGUARD	0xAC	1 byte	Guard byte for FAC
CHRGET	0xB1:0xB6	6 bytes	Routine to increment TXTPTR for CHRGOT
CHRGOT	0xB7:0xC8	18 bytes	Routine to load or obtain next character at TXTPTR
TXTPTR	0xB8:0xB9	2 bytes	Pointer for CHRGET and CHRGOT

Table 6. Page-Zero Utilization

I believe the LOADARG and the LOADFAC routines could both be similarly modified and utilize a fast register copy loop in order to obtain eleven and ten bytes of ROM space, respectively, as shown in Table 7. To what extent can one justify changing the documented entry address to a particular floating point routine? Perhaps changing only what is absolutely necessary in order to implement a fix to an Applesoft arithmetic irregularity might very well be justified. I am sure that some intermediate arguments could, perhaps, yield far more accurate results if their guard byte is also pushed onto the stack along with their 32-bit significand rather than simply rounding these intermediate arguments and pushing the rounded 32-bit significand onto the stack. I have no doubt that a number of simple modifications can be implemented that will greatly assist in giving Applesoft arithmetic fewer irregularities and far greater numerical accuracy. It is my intention to present such ROM modifications to particular Applesoft floating point routines. It is certainly not inconceivable to build a new ROM image that would save, restore, and utilize guard bytes for all

Applesoft arithmetic up to the COPYFAC routine. Only at this point in the Applesoft numerical calculations should FAC be rounded by inspecting its guard byte.

Name	Address	Original Code		Modified Code		Delta Difference	
		Size	Time	Size	Time	Size	Time
LOADARG	0xE9E3	0x2B	73	0x20	107	0x0B	34
LOADFAC	0xEAF9	0x25	64	0x1B	105	0x0A	41
COPYFAC	0xEB2B	0x28	94	0x21	133	0x07	39

Table 7. Byte-By-Byte Copy Versus Register-Loop Copy

Name	Val	Exp	Man	Sgn	Eor	Add	Sub	Cry	Exp	Sgn
FAC	12.34	0x84	0xC5	0x45		0x0A		set	0x8A	0x26
ARG	56.78	0x86	0xE3	0x63	0x26		0x02	set	0x82	0x26
FAC	-12.34	0x85	0xC5	0xC5		0x0A		set	0x8A	0xA6
ARG	56.78	0x86	0xE3	0x63	0xA6		0x02	set	0x82	0xA6
FAC	12.34	0x84	0xC5	0x45		0x0A		set	0x8A	0xA6
ARG	-56.78	0x86	0xE3	0xE3	0xA6		0x02	set	0x82	0xA6
FAC	-12.34	0x84	0xC5	0xC5		0x0A		set	0x8A	0x26
ARG	-56.78	0x86	0xE3	0xE3	0x26		0x02	set	0x82	0x26
FAC	56.78	0x86	0xE3	0x63		0x0A		set	0x8A	0x26
ARG	12.34	0x84	0xC5	0x45	0x26		0xFE	clear	0x7E	0x26
FAC	12.34E+05	0x95	0x96	0x16		0x3D		set	0xBD	0x12
ARG	56.78E+10	0xA8	0x84	0x04	0x12		0x13	set	0x93	0x12
FAC	56.78E+10	0xA8	0x84	0x04		0x3D		set	0xBD	0x12
ARG	12.34E+05	0xA9	0x96	0x16	0x12		0xED	clear	0x6D	0x12
FAC	12.34E-05	0x74	0x81	0x01		0xD9		clear	0x59	0x42
ARG	56.78E-10	0x65	0xC3	0x43	0x42		0xF1	clear	0x71	0x42
FAC	56.78E-10	0x65	0xC3	0x43		0xD9		clear	0x59	0x42
ARG	12.34E-05	0x74	0x81	0x01	0x42		0x0F	set	0x8F	0x42
FAC	12.34E+05	0x95	0x96	0x16		0xFA		clear	0x7A	0x55
ARG	56.78E-10	0x65	0xC3	0x43	0x55		0xD0	clear	0x50	0x55
FAC	56.78E-10	0x65	0xC3	0x43		0xFA		clear	0x7A	0x55
ARG	12.34E+05	0x95	0x96	0x16	0x55		0x01	set	0xB0	0x55
FAC	12.34E-05	0x74	0x81	0x01		0x1C		set	0x9C	0x05
ARG	56.78E+10	0xA8	0x84	0x04	0x05		0x34	set	0xB4	0x05
FAC	56.78E+10	0xA8	0x84	0x04		0x1C		set	0x9C	0x05
ARG	12.34E-05	0x74	0x81	0x01	0x05		0xCC	clear	0x4C	0x05

Table 8. Exponent Processing for FMULT and FDIV

Applesoft Exponent Processing For Floating Point Numbers

Exponents are processed uniquely depending upon the arithmetic that is about to be performed. Both the FMULT routine at 0xE97F and the FDIV routine at 0xEA66 utilize a common routine at 0xEA0E that processes the exponents of the FAC and the ARG floating point variables. In multiplication, ARG is the multiplicand, FAC is the multiplier, and LASTMUL contains their product. In division, ARG is the dividend, FAC is the divisor, and LASTMUL contains their quotient. There are instances where more accuracy is the result depending upon which values occupy FAC and ARG. Often, there is no choice in the matter particularly when multiple arithmetic operations are performed sequentially. At the conclusion of FMULT or FDIV, the mantissa is always copied from LASTMUL to FAC and the exponent in FAC is finalized. At 0xEA0E, if ARGEXP is zero, both FACEXP and FACSIGN are cleared to zero and the arithmetic operation is terminated. Otherwise, ARGEXP is added to FACEXP. If the carry flag is clear and the sum is less than 0x80, then both FACEXP and FACSIGN are cleared to zero and the arithmetic operation is terminated as above. On the other hand, if the carry flag is set and the sum is greater than 0x80, the OVERFLOW error handler is called and the arithmetic operation is terminated. Otherwise, the carry flag is cleared and the sum is added to EXPBIAS or 0x80 and that sum is stored at FACEXP. If this sum is not equal to zero, the value that is stored at XORSIGN is copied to FACSIGN. However, if this sum is equal to zero, then zero is stored at FACSIGN.

Table 8 shows a number of different values for FAC and ARG and some of the intermediate results in processing their exponents at 0xEA0E. Only the first byte of their mantissa is shown since that byte contains the sign bit in its MSB. The unaltered mantissa byte is shown in the sign column Sgn and, after EXPBIAS is added, in the mantissa column Man. When LOADARG copies a floating point value to ARG, it also EORs its ARGSIGN with FACSIGN and that value is shown in the Eor column, and that is the value that is saved to XORSIGN. The exponents are added for FMULT and they are subtracted for FDIV. In order to utilize a common routine, FACEXP is subtracted from zero and a true two's complement is created for FACEXP, and that value is saved back to FACEXP in FDIV before FDIV calls 0xEA0E. Thus, Table 8 separates the addition for FMULT in the Add column and the addition for FDIV in the Sub column. The state of the carry flag as a result of this addition is shown in the Cry column and the resulting sum after EXPBIAS is added is shown in the Exp column, and that is the value that is saved to FACEXP. The final value for FACSIGN is copied from XORSIGN and that value is shown in the final Sgn column.

Designing an Applesoft Floating Point Multiply Algorithm

My version of a floating point multiply algorithm handles the exponents slightly differently. I add the two exponents and clear the MSB. If the carry flag is set, I ORA EXPBIAS to their sum, and I save the result to FACEXP in either case. For the sign bit, I mask out the MSB from the first byte of each mantissa, EOR those masked bits, and save the result to FACSIGN. I ORA the first byte of each mantissa with EXPBIAS, the implicit high-order one bit. The multiplication is done in a processing loop that executes thirty-two times, one time for each bit in FAC. I assume no input initial guard byte for FAC. The product does, indeed, include a guard byte and all product mantissa bytes and its guard byte are initialized to zero. The processing loop begins by shifting the multiplier, the FAC mantissa in this algorithm, one bit to the right in order to capture its LSB into the carry flag. Only if the carry flag is set does the algorithm add the multiplicand, the ARG mantissa in this algorithm, to the product mantissa, capturing the resulting carry flag. The carry flag that was clear from the right shift of the multiplier or the carry flag from the addition of the multiplicand and the product is shifted into the product mantissa as those mantissa bytes and its guard byte are all shifted one bit to the right. Whatever bit that is shifted out of the guard byte is lost. At the end of the processing loop, the MSB of the product mantissa is inspected. If its MSB is not set, the product mantissa and its guard byte is shifted one bit to the left until its MSB becomes set in order to create the implicit high-order one bit. Upon each shift of the product mantissa and its guard byte to the

left, FACEXP is decremented. The MSB of the finalized guard byte is now inspected. If its MSB is set, the product mantissa is incremented by one. The algorithm is complete after the MSB of the product mantissa is replaced by FACSIGN. FACEXP and the product mantissa are both copied to the address for the return value. This algorithm is very straightforward and simple to implement, and it requires only thirty-nine bytes for its complete processing.

FMULT makes five calls to common code, the first call on behalf of its multiplier guard byte and one call on behalf of each byte of its four-byte multiplier mantissa, the FAC mantissa to be precise, beginning with the last byte of that mantissa. The fifth call to the common code actually bypasses a simple check for zero and that call enters the byte multiply routine directly. If the multiplier guard byte or any of the last three bytes of the multiplier mantissa is zero, the bytes of the product mantissa, or LASTMUL in this algorithm, are byte-copied to the right where the first byte of LASTMUL is set to zero and the last byte of LASTMUL is copied into its guard byte. This step certainly does expedite this algorithm by eliminating eight full bit-shifts to the right for LASTMUL for every zero that is found in the multiplier guard byte or in the last three multiplier mantissa bytes. On the other hand, when the multiplier guard byte or the last three multiplier mantissa bytes are not equal to zero, and always the first multiplier mantissa byte, these multiplier bytes are processed in a very unique and clever way. The multiplier byte is first bit-shifted to the right in order to capture its LSB into the carry flag and then the MSB of this byte is set to one in order to also utilize this byte as an 8-bit counter. As long as register-A is not equal to zero after each and every register bit-shift to the right, processing will continue within this unique processing loop. If the carry flag is set from the captured LSB, the multiplicand, the ARG mantissa in this algorithm, is added to LASTMUL. The clear carry flag from the captured LSB or the carry flag from the LASTMUL addition is shifted into the first byte of LASTMUL as LASTMUL and its guard byte are shifted one bit to the right. Again, as long as register-A, after the register is shifted one bit to the right, is not equal to zero, processing will continue in this clever loop. After the first byte of the multiplier mantissa is processed which is after the fifth call to the common code, LASTMUL is copied to FAC, its exponent is normalized, and the sign bit is added to the FAC mantissa.

This algorithm is certainly not as straightforward as the one I designed, and it requires seventy-nine bytes for its processing loop which does not even include the instructions that byte-shift LASTMUL and its guard byte to the right when the call to the common code contains a zero byte. This is another example where speed of execution requires far more executable code and/or data as is often noted by Glen Bredon. Even though this algorithm is far more complex than the algorithm I designed, its layout is well organized and situated for optimal speed in execution. The byte-shift to the right routine begins at 0xE8DA and it extends until 0xE912, and this routine is very cumbersome and odd, and it appears to be an attempt to utilize a routine that is designed for several special-purpose requirements as a general-purpose routine. I propose that this routine should be modified and rewritten into several special-purpose routines simply to further its speed of execution. The various floating point numbers and polynomials that begin at 0xE913 can certainly be moved elsewhere. Would any Applesoft arithmetic irregularities be eliminated if these static floating point numbers also included a fifth mantissa or guard byte value? I wonder.

Table 9 summarizes the results that I obtained when I multiplied two floating point numbers using the multiply routine that I designed and the multiply routine that is found in Applesoft ROM. This table also shows that there does exist a difference in timing that depends upon the order of the two floating point numbers. This order is shown only for curiosity purposes and this order cannot be pre-determined to always utilize the variable order for the faster calculation. When there are a series of floating point calculations, usually the order that the variables are processed cannot be altered. When a vast number of similar calculations are required, variable order will probably be random enough such that an average overall timing experience will result. Table 9 definitely shows that the Applesoft ROM contains a

remarkably faster multiply routine, about 2.6 times faster, than the straightforward multiply routine that I designed. I have no doubt that there must have been some very interesting design reviews when Microsoft began to piece together its very first BASIC interpreter.

Variables	Function	Address	Cycle Count	Processing	Guard Byte
My Code FAC = 12.34 ARG = 56.78	Initialization	0xB751	66,391		0x0C
	Multiply Loop	0xB7BB	66,618	227	
	Finish	0xB7E2	71,843	5225	
	End	0xB82F	72,013	170	
My Code FAC = 56.78 ARG = 12.34	Initialization	0xB751	91,274		0x0C
	Multiply Loop	0xB7BB	91,501	227	
	Finish	0xB7E2	96,652	5151	
	End	0xB82F	96,822	170	
ROM Code FAC = 12.34 ARG = 56.78	Initialization	0xE987	52,807		0x0C
	Multiply Loop	0xE994	52,867	60	
	Finish	0xE9AD	54,883	2016	
	End	0xB24E	54,950	67	
ROM Code FAC = 56.78 ARG = 12.34	Initialization	0xE987	64,242		0x0C
	Multiply Loop	0xE994	64,302	60	
	Finish	0xE9AD	66,281	1979	
	End	0xB24E	66,348	67	

Table 9. Timing Results for Multiply Routines

The FMULT routine shows a very high level understanding of assembly language for the 6502 microprocessor that is somewhat analogous to how Egan Ford designed his data byte read routines for the high frequency transmission of diskette data by also using register-A as an eight-bit counter. Both multiply routines show that there is a timing difference that is directly related to variable order. Both routines show that the timing for Initialization and Finish are consistent in that they generate the same values for each routine without regard for their variable order. Both routines yield the same value for their guard byte though I was not expecting to observe the same guard byte value even when the order for these variables was changed. That was a surprise. As I noted above, the layout of the Applesoft ROM FMULT routine is well organized and situated for optimal execution speed. The byte move routine from 0xE8DC to 0xE912 is not written uniquely for FMULT, though FMULT enters this routine at a special entry point at 0xE8DA in order to initialize register-X to point to LASTMUL.

The Byte Move routine is instrumental in normalizing floating point variables so that their mantissas can be added or subtracted. The two's complement of the difference of two exponents is compared to the value of 0xF9. If the exponent difference is equal to or greater than 0xF9, then the Bit Move feature of this routine is entered midway at 0xE907. Otherwise, the Byte Move routine is entered at the bottom of the routine at 0xE8F0. Any remainder from successively subtracting eight from the exponent difference and moving the bytes of a mantissa to the right is processed by the Bit Move portion of the routine that follows. The Byte Move routine is truly complex software for it contains many nuances. Indeed, one of its nuances is utilized by FMULT. Whenever FMULT processes a nonzero mantissa byte in FAC, the carry

flag is always set whenever the eight-bit counter bit is shifted to the right into the carry flag leaving that mantissa byte zero in order to designate the end of processing. If the next FAC mantissa byte happens to be zero, FMULT enters the Byte Move routine at 0xE8DA with the carry flag set. Once the mantissa bytes have been shifted to the right, a calculation is made that, again, leaves the carry flag set, a condition that happens only when FMULT utilizes this routine. Thus, a return is made to FMULT without executing any other instructions. The first four lines of code for the Bit Move routine, which begins at 0xE8FD, could not have been written in a more obtuse way knowing that all three registers are allocated and that no page-zero BIT instruction that uses register-X exists in the 6502 microprocessor instruction set.

The FMULT routine can be modified such that the call to the Byte Move routine at 0xE8DA can be entirely avoided along with its processing nuances that are totally unnecessary for FMULT. In order to accomplish this modification, specific Byte Move logic for FMULT must be inserted at 0xE98D in order to turn this routine into a register copy loop rather than using the call to 0xE8DA, and the mantissa addition logic that begins at 0xE9BC must also be transformed into a register addition loop. When these two modifications are made to FMULT, two bytes are still required in order to prevent the modified FMULT from spilling over into the next routine which is LOADARG at 0xE9E3. Those two bytes can be recovered from FMULT initialization by utilizing the value in register-Y after FMULT calls LOADARG. At the end of LOADARG processing, register-Y becomes zero. Therefore, LASTMUL can be initialized using register-Y rather than loading register-A with zero, thus saving two bytes. I prepared a new ROM image and repeated the same tests that are shown in Table 9 albeit slightly different ROM addresses for the same FMULT functions. I expect somewhat larger timing values. The initial guard byte is processed by the newly installed Byte Move routine and the timing of that processing is similar to the original Byte Move timing. The major timing difference becomes apparent from the new register addition loop. This loop utilizes register-X to load, add, and store mantissa bytes, and these instructions are each four cycles rather than each three cycles as in the original mantissa addition routine. Each pass through the new addition routine requires sixty-nine cycles rather than thirty-six cycles. Therefore, assuming equal number of zero and one bits for FAC, over five hundred additional cycles are spent by the modified FMULT routine when it multiplies 12.34 and 56.78. That is over 0.5 milliseconds more time spent for each call to the modified FMULT routine. This modification to FMULT appears to be impractical if only due to the added timing. Perhaps the modifications to FDIV may not be so impractical?

Designing an Applesoft Floating Point Divide Algorithm

My version of a floating point divide algorithm handles the exponents similar to how I handle the exponents in my multiply algorithm. I subtract the FAC divisor exponent from zero and add that two's complement value to the ARG dividend exponent, clear the MSB, and if the carry flag is set, I ORA EXPBIAS to their sum, save the result to FACEXP, and increment FACEXP, a step that is not done in the multiply algorithm. For the sign bit, I mask out the MSB from the first byte of each mantissa, EOR those masked bits, and save the result to FACSIGN. I ORA the first byte of each mantissa with EXPBIAS, the implicit high-order one bit. The division is handled in a processing loop that executes forty times, that is, one loop for each bit in FAC and eight loops in order to create a guard byte. I assume no initial guard byte for FAC or for ARG though such guard bytes could very well produce far more precise results, so I initialize a guard byte for FAC and a guard byte for ARG and set those bytes to zero. The quotient does, indeed, include a guard byte and all quotient mantissa bytes and its guard byte are initialized to zero. The processing loop begins by shifting the quotient and its guard byte one bit to the left and setting the LSB of its guard byte to zero. Next, I subtract the divisor mantissa from the dividend mantissa and save that difference into a temporary mantissa and I capture the state of the carry flag. Only if the carry flag is set does my algorithm copy the temporary mantissa into the dividend mantissa and increment the guard byte of the quotient whose LSB was previously set to zero. Whether the carry flag is set or clear from the subtraction of the

dividend by the divisor, the divisor is shifted one bit to the right such that its MSB is always set to zero. Whatever bit that is shifted out of the divisor's guard byte is lost. At the end of all processing loops, the MSB of the quotient mantissa is inspected. If its MSB is not set, the quotient mantissa and its guard byte are shifted one bit to the left until its MSB becomes set in order to create the implicit high-order one bit. Upon each shift to the left of the quotient mantissa and its guard byte, FACEXP is decremented. The MSB of the finalized guard byte for the quotient is now inspected. If its MSB is set, the quotient mantissa is incremented by one. The algorithm is complete after the MSB of the quotient mantissa is replaced by FACSIGN. FACEXP and the quotient mantissa are both copied to the address for the return value. This algorithm is very straightforward and simple to implement, and it requires only fifty-seven bytes for its complete processing.

FDIV initializes register-X with 0xFC as a four byte counter and register-A with 0x01 as an eight bit counter. At the top of the processing loop for FDIV, register-Y is utilized in order to make a preliminary compare of ARG and FAC, the dividend and the divisor, respectively, and capture the state of the carry flag into the LSB of register-A. If the carry flag is set, FDIV performs the actual subtraction where the minuend is the dividend and the difference is stored as the new dividend. Whether the dividend is replaced or not, ARG is shifted one bit to the left and its MSB is shifted into the carry flag. If the carry flag is set or if the current ARG mantissa is positive determine the two conditions that will occur if, indeed, the dividend is still smaller than the divisor, and a preliminary compare of the two mantissas at the top of the processing loop is unnecessary. Only when the carry flag is clear and when the ARG mantissa is negative does FDIV make another preliminary compare of the dividend and the divisor. This strategy is brilliant and it confines all processing to what is only necessary in order to efficiently progress to the next processing iteration. In capturing the state of the carry flag into register-A from either the preliminary compare of dividend and divisor or from the shift of ARG one bit to the left as register-A is also shifted one bit to the left, and as long as the MSB of register-A, which shifts into the carry flag, is clear, the next processing iteration will continue for that quotient byte. Similar in how the MSB of register-A is set in FMULT in order to provide an eight bit counter as register-A is shifted right, register-A provides an eight bit counter as register-A is shifted left in FDIV. As soon as this eight bit counter expires, register-X is incremented and that register is used as an index in order to save the quotient value that is currently in register-A to LASTMUL as the first or the next byte in the quotient mantissa. Register-X is also used to determine if FDIV should continue to calculate the next quotient byte and again initialize register-A with 0x01, to initialize register-A with 0x40 in order to calculate only bit 6 and bit 7 of a guard byte, or to conclude FDIV processing entirely and copy LASTMUL into FAC, normalize the FAC exponent, and add in the sign bit to the FAC mantissa.

A side effect of FDIV is that it saves its preliminary guard byte value to memory at 0x66 before the routine shifts that guard byte value six bits to the left and saves that shifted value to GUARD1 at 0xAC. This algorithm is obviously not as straightforward as the one I designed, and its processing loop requires one hundred six bytes for its implementation. Even though this is a far more complex algorithm, its layout is not well organized and it is not situated for optimal speed in execution. On one hand, this algorithm is brilliant in its decision making logic, and on the other hand, its layout and its organization appears as if it was implemented by a student. For example, if the subtraction of FAC from ARG at 0xEAB4 is moved to 0xEAA4, the JMP instruction at 0xEACE is totally unnecessary after changing the BCS branch instruction at 0xEAA4 to BCC at 0xEAA2. The load of register-A with 0x40 at 0xEAD1 could be moved to 0xEAA1 and eliminate its branch instruction. The six bytes of ASL could be replaced with three bytes of ROR at 0xEAD5. The error code 0x85 (DIVISION BY ZERO) could be moved to 0xEAD4 rather than at 0xEAE1 in order to eliminate the JMP instruction at 0xEADE. Ten bytes overall can easily be extracted from the FDIV routine and reduce its processing time by approximately sixty-two microseconds. I am convinced that some Applesoft arithmetic irregularities could definitely be eliminated if these floating point numbers could each be supplied with a guard byte value and only implement mantissa rounding just before a floating

point number is returned to the user and not each time before the FAC exponent is normalized. Of course, a full eight-bit guard byte should be produced by FDIV rather than only two valid bits.

Variables	Function	Address	Cycle Count	Processing	Guard Byte
My Code FAC = 12.34 ARG = 56.78	Initialization	0xB8F5	20,334		0x37
	Divide Loop	0xB96E	20,583	249	
	Finish	0xB9AC	31,774	11,191	
	End	0xB9F9	31,943	169	
My Code FAC = 56.78 ARG = 12.34	Initialization	0xB8F5	24,808		0x5A
	Divide Loop	0xB96E	25,056	248	
	Finish	0xB9AC	36,247	11433	
	End	0xB9F9	36,489	242	
Original ROM Code FAC = 12.34 ARG = 56.78	Initialization	0xEA6E	55,756		0x00
	Divide Loop	0xEA7C	55,819	63	
	Finish	0xEADB	58,053	2234	
	End	0xBA28	58,127	74	
Original ROM Code FAC = 56.78 ARG = 12.34	Initialization	0xEA6E	63,975		0x00
	Divide Loop	0xEA7C	64,033	58	
	Finish	0xEADB	66,274	2241	
	End	0xBA28	66,378	104	
Modified ROM Code FAC = 12.34 ARG = 56.78	Initialization	0xEA6E	66,648		0x31
	Divide Loop	0xEA7C	66,711	63	
	Finish	0xEAE3	69,376	2665	
	End	0xBA28	69,447	71	
Modified ROM Code FAC = 56.78 ARG = 12.34	Initialization	0xEA6E	35,141		0x50
	Divide Loop	0xEA7C	35,199	58	
	Finish	0xEAE3	37,868	2669	
	End	0xBA28	37,969	101	

Table 10. Timing Results for Divide Routines

It is rather simple for FDIV to generate a fully valid eight-bit guard byte, eliminate the side effect of writing to memory 0x66, and include guard byte processing for both ARG and FAC. Register-X must be initialized with 0xFB at 0xEA7C rather than 0xFC. When register-X is incremented at 0xEA9A, if the register is zero, an exit branch is made to 0xEAE3 in order to save the mantissa value in register-A to GUARD1 at 0xAC rather than to 0x66. (Perhaps 0x92 would be a better choice for the ARG guard byte and use 0x66 for the LASTMUL guard byte.) The subtraction of GUARD1 from GUARD2 is included at 0xEAA5 which allows 0x66 to be utilized as GUARD2 for ARG. These simple modifications to FDIV extract twelve rather than ten unused bytes from the FDIV routine and its processing loop only requires eighty-two bytes for this implementation. Of course, having to create a fully valid eight-bit guard byte rather than simply the first two significant bits of a guard byte will impact the processing time of FDIV. Presently, FDIV generates a guard byte that only has two significant bits in order to accommodate exponent normalization which assumes that no more than two bits will ever be shifted into the fourth mantissa byte. This strategy does not provide a useful guard byte for any subsequent processing. I believe the modifications that I have just

described for FDIV does, indeed, utilize guard bytes from previous arithmetic processing and it generates a complete guard byte for subsequent processing. Therefore, with these modifications, FDIV is certainly far more meticulous and the extra processing time is worth generating a fully valid eight-bit guard byte.

Table 10 summarizes the results that I obtained when I divided two floating point numbers using the divide routine that I designed, the divide routine that is found in Applesoft ROM, and a modified Applesoft ROM divide routine. This table also shows that there does exist a difference in timing that depends upon the order of the two floating point variables. This order is shown only for curiosity purposes and this order cannot be pre-determined to always utilize the variable order for the faster calculation. When there are a series of floating point calculations, usually the order that the variables are processed cannot be altered. When a vast number of similar calculations are required, variable order will probably be random enough such that an average overall timing experience will result. Table 10 definitely shows that the Applesoft ROM contains a remarkably faster divide routine, about 5.0 times faster, than the straightforward divide routine that I designed. The FDIV routine shows a brilliant understanding of assembly language logic for the 6502 microprocessor, yet a confused understanding of software layout and implementation. All divide routines show that there is a timing difference that is directly related to variable order. All routines show that the timing for Initialization and Finish are consistent in that they generate the same relative values for each routine particularly when variable order initiates further exponent normalization. All routines yield unique values for their guard byte. The modified ROM version of FDIV is about 19.2% slower, which seems reasonable. The modified ROM version includes the subtraction of dividend and divisor guard bytes and this ROM version generates five mantissa bytes rather than four mantissa bytes. By means of a far more logical software layout, the modified ROM version extracts twelve unused ROM bytes.

Designing an Applesoft Floating Point Print Algorithm

The FPRINT ROM routine at 0xED34 converts a floating point number into an ASCII string that generates base-10 digits for a real number to the left of the decimal point and it may or may not generate base-10 digits to the right of the decimal point. This ROM routine is exceedingly complex and this routine is utilized by the Applesoft PRINT and the STR\$ statements, though each have unique entry points. The PRINT statement enters FPRINT at 0xED34 in order to initialize register-Y to 0x01. On the other hand, the STR\$ statement enters FPRINT two bytes later at 0xED36 after it has already initialized register-Y to 0x00 at 0xE3C8. The beginning of the 6502-microprocessor STACK at 0x0100 is utilized to hold the base-10 digits that are generated for PRINT and the beginning of STACK-1 is utilized to hold the base-10 digits that are generated for STR\$. The FPRINT routine sets register-A/register-Y to the address of the STACK when the FPRINT routine completes its processing. However, STR\$ resets register-A/register-Y to STACK-1 or 0x00FF at 0xE3CF. Out of curiosity, I changed the address for the call at 0xE3CA to 0xED34 rather than 0xED36 and I retained the address of the STACK in register-A/register-Y when the routine returned at 0xE3CD. The STR\$ function provided identical results which was certainly no surprise. This appears to be yet another example where teams of programmers utilize a common routine with alternate startup parameters. The STR\$ function utilizes additional and totally unnecessary instructions that only inflate the size of ROM software, increase overall processing time, and provide no additional capabilities.

The FPRINT routine leverages off the observation that when the exponent and the mantissa of any floating point number is either multiplied or divided by ten until that number resides anywhere between 99,999,999.9 and 999,999,999.0 and its mantissa is shifted to the right such that its exponent equals 0xA0, all fractional bits are removed from that floating point number. The FP2INT ROM routine at 0xEBF2 converts the mantissa of a floating point number to an integer value by multiplying or dividing that number by ten some number of times. Base-10 digits can then be generated, even including the insertion of a decimal point when appropriate, by using successive addition. In order to utilize FP2INT, a Count variable is generated during the multiply or the divide preprocess that is used to generate a base-

10 exponent EXP for very small or very large floating point numbers and a Digit count for the number of digits to the left of the decimal point. Table 11 displays the values for Count, EXP, and Digit that are obtained when a floating point number is preprocessed and converted to an integer before employing successive addition in order to generate each base-10 digit.

Value	Display	FAC Value	Count	EXP	Digits
0.0001	1E-04	0x73 51B7175A	0xF4	0xFC	0x01
0.001	1E-03	0x77 03126E98	0xF5	0xFD	0x01
0.01	0.01	0x7A 23D70A3E	0xF6	0x00	0xFF
0.1	0.1	0x7D 4CCCCCD	0xF7	0x00	0x00
1	1	0x81 00000000	0xF8	0x00	0x01
10	10	0x84 20000000	0xF9	0x00	0x02
100	100	0x87 7A000000	0xFA	0x00	0x03
1000	1000	0x8A 7A000000	0xFB	0x00	0x04
10,000	10000	0x8E 1C400000	0xFC	0x00	0x05
100,000	100000	0x91 43500000	0xFD	0x00	0x06
1,000,000	1000000	0x94 74240000	0xFE	0x00	0x07
10,000,000	10000000	0x98 18968000	0xFF	0x00	0x08
100,000,000	100000000	0x9B 3EBC2000	0x00	0x00	0x09
1,000,000,000	1E+09	0x9E 6E6B2800	0x01	0x09	0x01
10,000,000,000	1E+10	0xA2 1502F900	0x02	0x0A	0x01
100,000,000,000	1E+11	0xA5 3A43B740	0x03	0x0B	0x01

Table 11. Floating Point Number Evaluation Process

Index	Address	Mantissa	Value
0x00	0xEE69	0xFA0A1F00	-100,000,000
0x04	0xEE6D	0x00989680	10,000,000
0x08	0xEE71	0xFFF0BDC0	-1,000,000
0x0C	0xEE75	0x000186A0	100,000
0x10	0xEE79	0xFFFFD8F0	-10,000
0x14	0xEE7D	0x000003E8	1,000
0x18	0xEE81	0FFFFFFF9C	-100
0x1C	0xEE85	0x0000000A	10
0x20	0xEE89	0FFFFFFFFF	-1

Table 12. Successive Addition Values

The FPCOMP ROM routine at 0xEBB2 evaluates the floating point number that currently resides in FAC at 0x9D against the supplied floating point number at 0xED0A for 99,999,999.9 or the supplied floating point number at 0xED0F for 999,999,999.0. The FPCOMP routine returns zero when FAC and the supplied floating point number are absolutely equal, 0x01 when FAC is greater than the supplied floating

point number, or $0xFF$ when FAC is less than the supplied floating point number. FAC is either multiplied or divided by ten in order to place FAC anywhere within this unique numerical range. FAC is then rounded up by adding an incredibly small floating point value of 0.5 . After FAC is converted to an integer value such that its exponent is now equal to $0xA0$, its mantissa alone can be processed simply by counting the number of successive additions that are needed to generate each of the base-10 digits. A table of nine four-byte integer values is found at $0xEE69$ as shown in Table 12, and, therefore, a maximum of up to nine base-10 digits can be generated for any floating point number.

A typical routine that utilizes successive subtraction in order to convert a base-16 number to a base-10 number is required to toss the last subtraction or to restore the last subtracted value when the difference becomes negative. The succeeding lower digits can then be extracted in a similar fashion. If successive addition of negative values is employed, on the other hand, an overshoot of one addition must toss the new sum or subtract what was added when that sum becomes negative. Successive addition is employed in the FPRINT routine such that the routine ping pongs about zero. The first value that is used from Table 12 is $-100,000,000$, and successive addition is employed until that sum becomes negative. When successive addition overshoots one extra addition and that sum becomes negative, the number of iterations is added to $0x30-1$ in order to generate an ASCII base-10 digit. At least one iteration will always occur in order to generate an ASCII base-10 0 . The next value that uses successive addition is $10,000,000$. This value is added until the sum become positive. What is actually happening here is quite clever. Instead of having to subtract the extra $-100,000,000$ that was added for the overshoot, the next successive addition of $10,000,000$ cancels the $-100,000,000$ such that the number of times $10,000,000$ is added creates a digit that can be derived from $10 - n$, where n is the number of iterations. Therefore, a 2's compliment is taken of the number of iterations, ten is added to that compliment, and $0x30-1$ can now be added in order to generate an ASCII base-10 digit. The remaining seven digits can be extracted in the same manner even if some of those remaining digits are ASCII 0 s. The decimal point is inserted into the ASCII string data according to Table 11. If only ASCII 0 s are found after the decimal point, those ASCII 0 s and the decimal point are removed by cleverly locating the string terminator $0x00$ in place of the decimal point.

Two examples will easily demonstrate the preprocessing that is required to utilize successive addition in order to obtain the whole and fractional base-10 digits in order for Applesoft to display a floating point number. The floating point value 473.12 is represented by $0x896C8F5C29$. It must be multiplied six times by ten in order to locate this number between $99,999,999.9$ and $999,999,999.0$. The resulting value is $0x9DE199E801$ and after adding 0.5 , the value becomes $0x9DE199E805$. According to Table 11 for numbers less than 1000 , Count is $0xFA$ which results in a value of $0x00$ for EXP and a value of $0x03$ for Digit. A value of $0x00$ for EXP simply means that scientific notation is not used for the display of this number and a value of $0x03$ for Digit means that three base-10 digits are followed by a decimal point. The exponent and the mantissa are converted from $0x9D$ to $0xA0$ in order to remove all fractional values: the mantissa is divided by eight by shifting the mantissa three bits to the right because $0xA0 - 0x9D = 0x03$ and the value becomes $0xA01C316840$. Thus, the mantissa has now been converted to an integer value, and $0x1C333D00$ in HEX is exactly $473,120,000$. The floating point value 7364.57 is represented by $0x8D66248F5C$. It must be multiplied five times by ten in order to locate this number between $99,999,999.9$ and $999,999,999.0$. The resulting value is $0x9EAF95C4A1$ and after adding 0.5 , the value is $0x9EAF95C4A3$. According to Table 11 for numbers less than $10,000$, Count is $0xFB$ which results in a value of $0x00$ for EXP and a value of $0x04$ for Digit. Therefore, scientific notation is not used for the display of this number and four base-10 digits are followed by a decimal point. The exponent and the mantissa are converted from $0x9E$ to $0xA0$ in order to remove all fractional values: the mantissa is divided by four by shifting the mantissa two bits to the right because $0xA0 - 0x9E = 0x02$ and the value becomes $0xA02BE57128$. Thus, the mantissa has now been converted to an integer value, and $0x2BE57128$ is exactly $736,457,000$.

The fast multiply by ten routine FMUL10 at 0xEA39 that is utilized by the FPRINT routine until a floating point number resides anywhere between 99,999,999.9 and 999,999,999.0 is unfortunately flawed. The algorithm is fast and simple enough in that it only uses addition rather than multiplication. FMUL10 copies FAC to ARG, adds 0x02 to the exponent of FAC which multiplies FAC by four, and adds ARG to FAC which produces a value in FAC that is five times greater. Finally, the exponent of FAC is incremented which multiplies FAC by two, thus multiplying the original number in FAC by ten. The flaw is introduced by the MOVAF routine at 0xEB63 that copies FAC to ARG and is perpetuated by the addition routine. MOVAF calls the RNDUP routine at 0xEB72 to possibly increment the FAC mantissa by inspecting and destroying its guard byte, and returning in order to initialize the FAC guard byte to 0x00. The FAC guard byte at 0xAC should be copied to the ARG guard byte, perhaps at 0x92. However, the addition routine FADD.2 at 0xE7CE initializes the ARG guard byte to 0x00. Therefore, both guard bytes are initialized to 0x00, and when added, the FAC guard byte remains set to 0x00. If the FMUL10 routine is called a number of times, which it is in FPRINT, all concept of extra precision is completely lost. For example, when 0.001234 is added to 5678.9, the resulting sum is 5678.901234 which is 0x8D317735B9,F8. When 0x8D317735B9,F8 is copied to ARG, ARG will contain 0x8D317735BA,00 and FAC will contain 0x8D317735B9,00 for the first stage of preprocessing by the FPRINT routine. The actual sum 5678.901234 is printed as 5678.90124 which is due to this flaw in the FMUL10 routine: the number has been incorrectly rounded far too many times. Perhaps if the ARG guard byte is initialized earlier in the addition/subtraction routine, a far better utilization of the ARG guard byte and the FAC guard byte could be realized. It seems like such a waste of effort to have available eight bits and make extraordinary decisions based only on a single bit.

Designing Applesoft Trigonometric Function Algorithms

The Applesoft trigonometric functions cosine and tangent are entirely based on the Applesoft sine function and on the trigonometric identities. Applesoft computes $\text{COS}(x) = \text{SIN}(x + \pi/2)$ simply for the sign of the generated numerical value even though there is no such trigonometric identity. Applesoft computes $\text{TAN}(x) = \text{SIN}(x) / \text{COS}(x)$. If there are irregularities in the Applesoft sine function, those irregularities will be perpetuated in the Applesoft cosine function and tangent function. The Applesoft sine function uses six pre-calculated polynomials, thus utilizing a modified Taylor series that can yield quite precise values when sufficient polynomials are included, particularly for angles that are not near the limit of their function, that is, near zero and near $\pi/2$. The polynomial routine at 0xEF5C uses successive multiplication followed by addition for each pre-calculated polynomial that is included in the POLY.SIN table, a table used solely to calculate the Applesoft sine function. Each time FAC is copied to TEMP1 at 0x0093 or to TEMP2 at 0x0098, FAC is rounded and its guard byte is initialized to 0x00. After each multiplication or each addition, FAC is normalized and rounded, and its guard byte is initialized to 0x00. Any attempt at maintaining numerical precision throughout the Taylor series is completely sacrificed. It would be rather interesting to compare the results of a Taylor series implementation if numerical rounding is performed once and only when the final result of the Applesoft sine function is returned to the user. The Taylor series that is utilized by the Applesoft sine function is expressed as follows:

$$\sin(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

This Taylor series will converge because the sign of the terms alternate, the factorial denominators become far greater than their numerators, and the radius of convergence is at infinity. Are these five polynomials sufficient for Applesoft to provide a minimum of nine digits of accuracy for the sine function?

Polynomial	Index	Applesoft Value	Base-10 Value	True Value	Base-10 Value
Entries	0x00	0x05	5	0x05	5
$-(2\pi)^{11/11!} * x^{11}$	0x01	0x84 E61A2D1B	-14.3813907	0x84 F183A7F2	-15.09464258
$(2\pi)^{9/9!} * x^9$	0x06	0x86 2807FBF8	42.0077971	0x86 283C1A44	42.05869395
$-(2\pi)^{7/7!} * x^7$	0x0B	0x87 99688901	-76.7041703	0x87 99696672	-76.70585975
$(2\pi)^{5/5!} * x^5$	0x10	0x87 2335DFE1	81.6052237	0x87 2335E33B	81.60524928
$-(2\pi)^{3/3!} * x^3$	0x15	0x86 A55DE728	-41.3417021	0x86 A55DE731	-41.34170224
$(2\pi) * x$	0x1A	0x83 490FDAA2	6.283185307	0x83 490FDAA2	6.283185307

Table 13. Applesoft Sine Function Polynomials

Polynomial	Index	Real Value	Base-10 Value
Entries	0x00	0x07	7
$-(2\pi)^{15/15!} * x^{15}$	0x01	0x80 B7D6DCF8	-0.718122302
$(2\pi)^{13/13!} * x^{13}$	0x06	0x82 747A1A6A	3.819952585
$-(2\pi)^{11/11!} * x^{11}$	0x0B	0x84 F183A7F2	-15.09464258
$(2\pi)^{9/9!} * x^9$	0x10	0x86 283C1A44	42.05869395
$-(2\pi)^{7/7!} * x^7$	0x15	0x87 99696672	-76.70585975
$(2\pi)^{5/5!} * x^5$	0x1A	0x87 2335E33B	81.60524928
$-(2\pi)^{3/3!} * x^3$	0x1F	0x86 A55DE731	-41.34170224
$(2\pi) * x$	0x24	0x83 490FDAA2	6.283185307

Table 14. Expanded Applesoft Sine Function Polynomials

Table 13 lists all of the pre-calculated polynomials that are used for the Applesoft sine function. Table 13 also includes the true values for these five pre-calculated polynomials which exposes a degree of mathematical manipulation to the ROM sine polynomials. I determined that if the true pre-calculated value for these five polynomials are used in the ROM image instead, sufficient calculation differences are obtained from the Applesoft sine function that are not trivial. At least two software modifications can be made to the ROM image that will provide sufficient space for two additional pre-calculated polynomials. The 2π floating point variable is part of the table of pre-calculated polynomials for the Applesoft sine function and it must remain within this table. However, the 2π floating point variable at 0xF06E may be eliminated and its use at 0xEFF4 can be modified accordingly. There are ten unreferenced bytes at 0xF094 that can also be eliminated. These bytes, a little example of narcissism, when exclusively or'd with 0x87 produce the unusable backward ASCII string MICROSOFT! Now, two additional pre-calculated polynomials can be added as shown in Table 14 and a new ROM image can be generated. I found a single difference in the output of the Apple ROM image versus a newly generated ROM image that utilizes seven pre-calculated polynomials for the Applesoft sine function. Taking the sine of 1.5701 yields 0.999999757 for the Apple ROM image and 0.999999758 for the newly generated ROM image. What I have observed, is that the five pre-calculated polynomials that Apple has selected for its Applesoft sine function have been mathematically modified such that they produce virtually the same results as if seven accurately pre-calculated polynomials have been utilized for the Applesoft sine function. Without knowing any details as to how the pre-calculated polynomials were mathematically modified and the mathematical rationale that was utilized for those modifications, I prefer to calculate the

Applesoft sine function using the seven pre-calculated polynomials that are shown in Table 14. At this time, sufficient space is currently available for those two additional pre-calculated polynomials.

Designing an Applesoft Natural Logarithm Algorithm

Applesoft calculates the natural logarithm FLOG at 0xE941 when an expression is provided with the LOG statement. In my opinion, this statement should be written as LN and the LOG statement should be reserved for the base-10 logarithm as they are commonly used in mathematics and in engineering functions. Of course, one can easily compute the base-10 logarithm LOG from the natural logarithm LN using the relationship $\log(x) = \ln(x) * \log(e)$ where e is about equal to 2.718281828, thus $\log(e)$ is about equal to 0.434294482. A conversion may be computed from $\ln(x)$ for any base-n logarithm simply by knowing the value of $\log(e)$ for that base-n. There exists a number of methods to calculate the natural logarithm of any positive floating point number. The Apple ROM factors out and saves n, the powers of 2 from the exponent of the input argument minus EXPBIAS, and reduces the value of the argument so that it is near the value of 1 in order to utilize the identity $\ln(x * 2^n) = \ln(x) + n * \ln(2)$. When an argument is reduced in this manner and its value is near the value of 1, the Taylor series expansion for $\ln(x)$ can be utilized because this series expansion produces an excellent approximation in this finite numerical range. The Taylor series expansion for the natural logarithm will converge faster when x is closer to 1.

$$\ln(x) = \int_1^x \frac{dt}{t} = \sum_{k=0}^{\infty} \frac{2}{2k+1} \left(\frac{x-1}{x+1} \right)^{2k+1}$$

After the Apple ROM routine saves and replaces the exponent of the input argument with EXPBIAS, the function adds the square root of 0.5 to the argument that is in FAC and FAC becomes the divisor that is used in order to divide the square root of 2.0. That quotient is then subtracted from 1.0. These simple add, divide, and subtract mathematical steps can be transformed as follows:

$$1 - \frac{\sqrt{2}}{x + \sqrt{.5}} = \frac{x - \sqrt{.5}}{x + \sqrt{.5}} = \frac{x\sqrt{2} - 1}{x\sqrt{2} + 1}$$

Now, FAC is in the correct format such that only four polynomials are required to very accurately calculate the natural logarithm and these four polynomials are shown in Table 15. After all of the polynomials have been processed, -0.5 is added to FAC, and n, that was saved earlier from the input argument, is converted into a floating point variable in order to multiply it with the natural log of 2.0.

An example will illustrate the Applesoft FLOG function. The statement LOG (7.5) yields the floating point value 2.01490302. When the FLOG function begins at 0xE941, FAC is equal to 0x83F0000000 and 0x03 is extracted from the exponent and saved. The exponent in FAC is now changed to 0x80 and the floating point number becomes 0x80F0000000 which is equal to 0.9375 after removing the sign bit. The square root of 0.5 is added to FAC and FAC becomes 0x81D282799A which is a positive number and equal to 1.644606781. Next, FAC is divided into the square root of 2.0 and FAC becomes 0x80DC230D36, also a positive number which is equal to 0.859909845. Finally, FAC is subtracted from 1.0 and FAC becomes 0x7E8F73CB28, still a positive number which is equal to 0.140090155. It is easy to verify that $(x\sqrt{2} - 1) / (x\sqrt{2} + 1)$ is equal to 0.140090155 when x is equal to 0.9375, thus confirming that these simple add, divide, and subtract mathematical steps transform the input value into the correct format before utilizing the modified Taylor series expansion polynomials. After processing using the modified Taylor series expansion polynomials, FAC becomes equal to 0x7FD053F6D1, which is a positive number and equal to

0.406890595. Negative 0.5 is added to FAC and FAC becomes equal to 0x7DBEB024B9, which is a negative number and equal to -0.093109405. The value n that was saved after subtracting EXPBIAS from the exponent of the original input number 7.5 is recalled, converted into a floating point number, and added to FAC, and FAC becomes equal to 0x82BA0A7EDA, which is a positive number and equal to 2.906890595. The last step in the Applesoft FLOG function is to multiply FAC by the natural LOG of 2.0 which is 0.693147181. FAC becomes equal to 0x8280F42BCC which is a positive number and equal to 2.01490302, the same value that Applesoft printed on the display.

Table 15 shows the values for all four modified Taylor series expansion polynomials that are used to process the transformed input value in order to calculate the natural logarithm for that input value. The Applesoft values deviate from the expected Taylor series expansion values. However, even if the true values that are shown in Table 15 are used along with the true values for the fifth and sixth polynomials, the resulting value is still not even close to the value that is obtained from using the modified Applesoft values for the four polynomials that are shown. Obviously, I have no idea how these modified polynomial values were calculated and the mathematical rationale that was utilized for these calculations. I would certainly champion the effort to change the statement symbol LOG to LN and include the simple multiplication of 0.434294482 in order to calculate the base-10 LOG of an input variable.

Polynomial	Index	Applesoft Value	Base-10 Value	True Value	Base-10 Value
Entries	0x00	0x03	3	0x03	3
$(2/7)(\sqrt{2}) * x^7$	0x01	0x7F 3E56CB79	0.43425594	0x7F 4EE115F3	0.404061018
$(2/5)(\sqrt{2}) * x^5$	0x06	0x80 139B0B64	0.57658454	0x80 10D0C292	0.565685425
$(2/3)(\sqrt{2}) * x^3$	0x0B	0x80 76389316	0.96180075	0x80 715BEEF0	0.942809042
$2(\sqrt{2}) * x$	0x10	0x82 38AA3B20	2.88539007	0x82 3504F336	2.828427125

Table 15. Applesoft Natural Log Function Polynomials

Designing an Applesoft Arctangent Routine Algorithm

The Applesoft arctangent routine FATAN begins at 0xF09E. This routine calculates the arctangent of the input value that is currently in FAC and it leaves the result, which is in radians, also in FAC. The routine saves the sign of the input value and compliments the input value if it is negative. The routine saves the exponent of the input value before inverting it if it is greater than 1.0. A modified Taylor series expansion having twelve polynomials is utilized to process this pre-modified value. If the saved exponent indicates that the input value is greater than 1.0, the result from Taylor series processing is subtracted from 2π . If the saved sign indicates that the input value was negative, the current value in FAC is complimented. The Taylor series expansion for the arctangent of any input value x is given as follows:

$$\tan^{-1}(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{2k+1}$$

Polynomial	Index	Applesoft Value	Base-10 Value	True Value	Base-10 Value
Entries	0x00	0x0B	11	0x0B	11
$-(1/23)*x^{23}$	0x01	0x76 B383BDD3	-6.84793912E-04	0x7C B21642D1	-0.043478261
$(1/21)*x^{21}$	0x06	0x79 1EF4A6F5	4.85094216E-03	0x7C 430C30DE	0.047619048
$-(1/19)*x^{19}$	0x0B	0x7B 83FCB010	-0.0161117018	0x7C D79435EA	-0.052631579
$(1/17)*x^{17}$	0x10	0x7C 0C1F67CA	0.034209638	0x7C 70F0F0D5	0.058823529
$-(1/15)*x^{15}$	0x15	0x7C DE53CBC1	-0.0542791328	0x7D 88888893	-0.066666667
$(1/13)*x^{13}$	0x1A	0x7D 1464704C	0.0724571965	0x7D 1D89D8A0	0.076923077
$-(1/11)*x^{11}$	0x1F	0x7D B7EA517A	-0.0898023954	0x7D BA2E8BA6	-0.090909091
$(1/9)*x^9$	0x24	0x7D 6330887E	0.110932413	0x7D 638E38E0	0.111111111
$-(1/7)*x^7$	0x29	0x7E 9244993A	-0.142839808	0x7E 92492496	-0.142857143
$(1/5)*x^5$	0x2E	0x7E 4CCC91C7	0.19999912	0x7E 4CCCCCD	0.2
$-(1/3)*x^3$	0x33	0x7F AAAAAA13	-0.333333316	0x7F AAAAAA8	-0.333333333
$1.0*x$	0x38	0x81 00000000	1.0	0x81 00000000	1.0

Table 16. Applesoft Arctangent Function Polynomials

This Taylor series expansion is very slow to converge. Isaac Newton suggested an accelerated means for this convergence that was later published by Leonhard Euler. Apple modified the polynomials as shown in Table 16 from the true values. The Apple polynomials grow substantially smaller as the denominator increases in value. I have found that the sample of test data that I utilized in order to compare the angle values that the Applesoft FATAN routine generates versus the angle values that a modern day calculator generates agree to all nine fractional digits. I have no access to the details as to how these pre-calculated polynomials that are shown in Table 16 were mathematically modified and the mathematical rationale that was utilized for those modifications.

Polynomial	Index	Applesoft Value	Base-10 Value	True Value	Base-10 Value
Entries	0x00	0x07	7	0x07	7
$(LN(2)^7)/7!*x^7$	0x15	0x71 34583E56	2.14987637E-05	0x70 7FE5FE2B	1.525273380E-05
$(LN(2)^6)/6!*x^6$	0x1A	0x74 167EB31B	1.43523140E-04	0x74 2184897B	1.540353039E-04
$(LN(2)^5)/5!*x^5$	0x1F	0x77 2FEEE385	1.34226348E-03	0x77 2EC3FF3E	1.333355815E-03
$(LN(2)^4)/4!*x^4$	0x24	0x7A 1D841C2A	9.61401701E-03	0x7A 1D955B7E	9.618129108E-03
$(LN(2)^3)/3!*x^3$	0x29	0x7C 6359580A	0.0555051269	0x7C 635846B8	0.05550410867
$(LN(2)^2)/2!*x^2$	0x2E	0x7E 75FDE7C6	0.240226385	0x7E 75FDEFFD	0.2402265070
$LN(2)/1!*x$	0x33	0x80 31721810	0.693147186	0x80 317217F8	0.6931471806
1.0	0x38	0x81 00000000	1.0	0x8100000000	1.0

Table 17. Applesoft Exponential Function Polynomials

Designing an Applesoft Exponential Routine Algorithm

According to the *Basic Programming Reference Manual for Applesoft II*, the Applesoft square root FSQR function at 0xEE8D is a special implementation that executes more quickly than $\sqrt{}$. However,

according to my analysis of the Applesoft ROM routine at 0xEE8D, this statement could not be more incorrect. The Applesoft FSQR function does use the Applesoft ^ function FPWRT at 0xEE97 where the value of 0.5 is loaded into FAC and the value that is in ARG, the input value, is thus taken to the power of 0.5. There is nothing special about this implementation and this implementation executes no faster than using FPWRT and 0.5. A far better approach to calculate the square root of a positive floating point number is to utilize a Newton-Raphson iteration. Programmatically, ARG contains the input value, FAC contains 0.5, and the Applesoft FSQR routine performs the following mathematical operation:

$$\text{ARG} \wedge \text{FAC} = \text{FEXP}(\text{FLOG}(\text{ARG}) * \text{FAC})$$

The Applesoft exponential routine FEXP begins at 0xEF09. This routine calculates e to the power of the input value that is currently in FAC and it leaves its result in FAC. The FEXP routine converts the input value to a power of 2 by multiplying the input value times the base-2 log of e . The base-2 log of e is the natural log of e divided by the natural log of 2, or $\text{LN}(e) / \text{LN}(2) = 1 / \text{LN}(2) = 1.442695041$. The value of 0x50 is added to the FAC guard byte at 0xAC and saved to 0x92, the location of the ARG guard byte. If that sum sets the carry flag, the FAC mantissa is rounded up. FAC is copied to ARG and if the FAC exponent is less than 0x88, processing continues, otherwise an overflow error message is generated. FAC is converted into an integer in order to generate a new exponent for the final fractional value. A modified Taylor series expansion having eight polynomials is utilized in order to process the fractional input value. The saved exponent is added to the final value that is obtained from polynomial processing. The Taylor series expansion for e to the power of any input value x is given as follows:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

This Taylor series expansion will converge moderately quickly. Apple modified the polynomials as shown in Table 17 from the true values by increasing the polynomials that have even factorials and decreasing the polynomials that have odd factorials. I have no access to the details as to how these pre-calculated polynomials that are shown in Table 17 were mathematically modified and the mathematical rationale that was utilized for those modifications.

Function	Expression	Intrinsic Function
Secant	SEC(X)	1 / COS(X)
Cosecant	CSC(X)	1 / SIN(X)
Cotangent	COT(X)	1 / TAN(X)
Inverse Sine	ARCSIN(X)	ATN(X / SQRC(1 - X*X))
Inverse Cosine	ARCCOS(X)	$\pi/2 - \text{ATN}(X / \text{SQRC}(1 - X*X))$
Inverse Secant	ARCSEC(X)	$\text{ATN}(\text{SQRC}(X*X - 1)) + (\text{SGN}(X) - 1) * \pi/2$
Inverse Cosecant	ARCCSC(X)	$\text{ATN}(1 / \text{SQRC}(X*X - 1)) + (\text{SGN}(X) - 1) * \pi/2$
Inverse Cotangent	ARCCOT(X)	$-\text{ATN}(X) + \pi/2$
Hyperbolic Sine	SINH(X)	$(\text{EXP}(X) - \text{EXP}(-X)) / 2$

Table 18. Applesoft Derived Arithmetic Functions

General Applesoft Design Considerations For Intrinsic Functions

The previous pages have explored in detail the finer points of some of the Applesoft arithmetic functions that are available directly to the Apple][user. These intrinsic functions may be used in order to calculate all of the derived functions that are shown in Table 18. From the beginning, I have made it abundantly clear that Applesoft routines and functions exhibit non-commutative addition, non-commutative multiplication, non-reflexive equality evaluation, irregularities of the exponent when the exponent is very small or very large, errors in the multiplication algorithm, errors in the binary to decimal conversion, and significant errors in the trigonometric functions that involve very small arguments. Some of the intermediate arguments depend on a full 40-bit significand since these arguments utilize a guard byte. But some intermediate arguments are rounded to their 32-bit significand. Furthermore, addition, subtraction, and multiplication utilize a mixture of a full forty bit significand and a thirty-two bit significand. In division, only the quotient has any extra significance. Even if the Applesoft arithmetic functions were to include guard bytes at all stages of their processing up until the final result is obtained for storage into memory, subsequent processing steps would still rely on only the saved thirty-two bit significands.

Problems that utilize significant floating point numerical processing such as rotational matrices and in-phase/quadrature radar mode instrumentation will succeed on the Apple][platform if close attention is paid to the limitations that are inherent in the Applesoft arithmetic functions. If solutions using such numerical values cannot be respectfully given in terms of the aforementioned limitations of these functions, another platform should be utilized that can provide double precision calculations at a minimum. It would be pointless to build a library of binary and complex floating point arithmetic routines for the Apple][platform using anything other than what is already available within the Applesoft ROM. Certainly, the Applesoft arithmetic functions are fully capable of providing solutions to problems that utilize floating point variables of reasonable size, that is, certainly a variable size that is somewhat distant from producing an underflow or an overflow error message. It is not the attempt of this document to dictate what can be and what can not be implemented within Applesoft knowing full well the absolute limitations of the floating point numerical system that is currently utilized by the Apple][computer.

Name	Usage	Description
LGNOT	CALL CM, "LGNOT", X%, Z%	Returns the EOR of X and 0xFF in Z
LGAND	CALL CM, "LGAND", X%, Y%, Z%	Returns the AND of X and Y in Z
LGORA	CALL CM, "LGORA", X%, Y%, Z%	Returns the OR of X and Y in Z
LGEOR	CALL CM, "LGEOR", X%, Y%, Z%	Returns the EOR of X and Y in Z
CXADD	CALL CM, "CXADD", X(0), Y(0), Z(0)	Returns X + Y in Z
CXSUB	CALL CM, "CXSUB", X(0), Y(0), Z(0)	Returns X - Y in Z
CXMUL	CALL CM, "CXMUL", X(0), Y(0), Z(0)	Returns X * Y in Z
CXDIV	CALL CM, "CXDIV", X(0), Y(0), Z(0)	Returns X / Y in Z
CX2PL	CALL CM, "CX2PL", X(0), Z(0)	Returns the magnitude and phase of X in Z
PL2CX	CALL CM, "PL2CX", X(0), Z(0)	Returns the complex number of X in Z
AYADD	CALL CM, "AYADD", X(0,0,0), Y(0,0,0), Z(0,0,0)	Returns array X + array Y in array Z
AYSUB	CALL CM, "AYSUB", X(0,0,0), Y(0,0,0), Z(0,0,0)	Returns array X - array Y in array Z
AYMUL	CALL CM, "AYMUL", X(0,0,0), Y(0,0,0), Z(0,0,0)	Returns array X * array Y in array Z
AYINV	CALL CM, "AYINV", X(0,0,0), Y(0), Z(0,0,0)	Returns det X in Y and X ⁻¹ in array Z
CSMUL	CALL CM, "CSMUL", X(0,0,0), Y(0), Z(0,0,0)	Returns array X * scalar Y in array Z

Table 19. Complete CXMATH Arithmetic Package

The Complete CXMATH Arithmetic Package for Applesoft

I have made a number of attempts to obtain a copy of the software and the algorithms that are detailed in Internal Report No. 225 and Internal Report No. 226 from the Electronics Division of the National Radio Astronomy Observatory. The individual PDF files for these reports are readily available by means of the internet. These are the reports that I have utilized in order to model the design and the development of my own version of binary and complex floating point routines that are contained in a package of arithmetic routines for the employment of complex floating point numbers. My math package is called CXMATH for Complex Floating Point Arithmetic Package. CXMATH includes integer logical functions, complex floating point functions, polar coordinate functions, and 2-by-2 complex floating point array and scalar functions. Table 19 summarizes the package contents of the CXMATH arithmetic routines for the Apple][computer.

CXMATH resides at $0xB000:0xB792$ as shown previously in Table 4. The loading Applesoft program should protect this region of memory and set HIMEM to $0xB000$. Only the first call to CXMATH does the LOADER routine set HIMEM to $0xB000$, so if this first call is very early in data processing, that is, before any character strings are moved into the Character String Pool, then CXMATH will protect itself. The LOADER routine first calls the LIBENT routine in order to process the name of the desired CXMATH routine. Many alternative choices exist for calling a particular routine among many other routines that are available in a package such as CXMATH. One can provide specific addresses, indices, or offsets in order to begin processing the intended routine from a base address, such as $0xB000$ in the case for CXMATH. Unfortunately, I have yet to write any software where I might make a few adjustments or improvements at some later date. Any change to a software routine will invariably alter its length and thus the entry addresses for all subsequent routines except, perhaps, in the case of jump tables. The insertion of a new routine, unless it is added last, will also cause a change to the entry addresses for all subsequent routines or to the content of a jump table. Obviously, the deletion of an unused or unnecessary routine will also cause a change to the entry addresses or to a jump table. I chose to utilize a unique calling procedure that would be totally insulated from any software changes, updates, insertions, and deletions. If the routine exists, CXMATH will process the intended inputs and provide the intended output or outputs.

Error	Calling Program	Error Message
ERR1	LIBENT	No variables included with command.
ERR2	LIBENT	Command size is wrong.
ERR3	LIBENT	Command not found in table.
ERR4	LINKER	Wrong number of arguments.

Table 20. CXMATH Error Messages

Each of the routines that are contained within CXMATH has a unique five letter ASCII name that is held between quotation marks as shown in Table 19. The LIBENT routine processes each character of the five-letter ASCII routine name and it generates a unique hash value from those letters. LIBENT then compares that hash value against a list of known hash values, one hash value for each routine within CXMATH. Not only does a matching hash value verify that the intended routine exists, it also provides an index for further information about that routine and its required input variables and its output variables. If a comma is not found after the five letter ASCII routine name, the ERR1 message is written to the screen as shown in Table 20 and DOSWARM at $0x3D0$ is entered which terminates further Applesoft processing. The unique five-

letter ASCII routine name may not be replaced using a character string variable such as CM\$ = "LGNOT" either because the wrong number of ASCII characters will be processed or because the wrong hash value will be generated regardless of the correct content of the character string variable. Thus, if LIBENT does not process five ASCII characters for a routine name, the ERR2 message is written to the screen and further Applesoft processing is terminated. If the generated hash value is not among the known hash values in HASHTBL, the ERR3 message is written to the screen and further Applesoft processing is terminated.

The CXMATH LOADER, LINKER, and LIBENT Routines

The LOADER routine next calls the LINKER routine in order to process the input and the output variables that are required by the specified CXMATH routine. The LINKER routine uses the index that LIBENT generates when LIBENT searches for a valid hash value, and that index is used to determine the number of variables that should follow the intended CXMATH routine name. If there is a mismatch in the number of these variables, the ERR4 message is written to the screen and further Applesoft processing is terminated. Typical complex floating point variables are presented as an array of two floating point numbers in Applesoft. Applesoft provides LINKER with the address of a variable or of an array of variables when LINKER calls the Applesoft routine PTRGET at 0xDFE3 following a call to CHRGET at 0xB1 when no comma is found on the command line. Once LINKER has the address for a variable or for an array of variables, it saves that address on page-zero to REALPTRS that is indexed by variable count. The first variable LINKER encounters is saved to 0x08:0x09, the second to 0x0A:0x0B, and the third to 0x0C:0x0D. Whether or not LINKER obtains an address for a complex number, LINKER calculates the address for the IMAGINARY variable that might follow the REAL variable if, indeed, the address is for a two-dimensioned array that represents a complex floating point number. Since a floating point number is five bytes in size, the IMAGINARY variable is found in memory five bytes after the REAL variable. Therefore, the calculated address for the first IMAGINARY variable is saved to 0x18:0x19, the second to 0x1A:0x1B, and the third to 0x1C:0x1D. These page-zero addresses are also saved locally within CXMATH data so that any REAL or IMAGINARY array variable may be accessed at any time and in any order as they are required by CXMATH.

The CXMATH Logical Operator Routines

The LG routines process integer numbers that are comprised of signed sixteen-bit integers. Applesoft integer numbers are designated using the percent % sign that follows the variable name. Only integer numbers may be used with the LG routines. Floating point numbers may never be used with LG routines. Table 1 shows that Applesoft integer numbers and Applesoft floating point numbers are vastly different in structure, in size, and in the definition of their numerical data. Positive integers range from 0 to 32767 or 0x0000 to 0x7FFF. Negative integers range from -1 to -32768 or 0xFFFF to 0x8000.

The CXMATH Complex Number Routines

The CX routines process complex numbers that are composed of two floating point numbers, a REAL number and an IMAGINARY number. The easiest method to handle complex numbers within Applesoft is to dimension each complex number as a two-dimensioned array. The REAL number always comes first in the two-dimensioned array, and the REAL number is followed by the IMAGINARY number in memory. There are absolutely no differences between a REAL number and an IMAGINARY number in terms of structure, identification, or labeling. An Applesoft complex number is simply a two-dimensioned array having two floating point numbers where the REAL number is followed by the IMAGINARY number. In order to provide the CX routines always with the first coefficient of a complex floating point number, the first element of a two-dimensioned array is specified with each variable's name. Applesoft allows floating point arrays, or all numerical and TEXT arrays for that matter, to begin with element 0. Therefore, when an array is dimensioned as in DIM X(1), Y(1), Z(1), each variable is an array of two elements, 0 and 1. Certainly, if this nomenclature seems unmanageable or counterintuitive, all two-dimensioned arrays may

also be dimensioned as DIM X(2), Y(2), Z(2), and each variable would refer to its REAL and IMAGINARY elements as 1 and 2, respectively. Element 0 will still be available in memory whether you wish it to be there or not, and each floating point element 0 of each floating point array will still consume five bytes of memory whether it is utilized or not. My preferences always tend to favor the conservation of memory especially when so little memory is actually available in an Apple][computer.

The CXMATH Conversion Routines

Both of the conversion routines for complex numbers and for polar coordinates utilize their two-dimensional floating point arrays uniquely for their particular purpose. In other words, the CX2PL routine expects a complex floating point number in X and the routine returns the magnitude in the first element of Z and the phase angle in the second element of Z. Both elements of Z are each a floating point number. The routine PL2CX expects a floating point magnitude in the first element of X and a floating point phase angle in the second element of X. This routine returns a complex floating point number in Z. The phase angle is a value that ranges between 0 and 2π . If a phase angle is greater than 2π when it is submitted to PL2CX, the Applesoft FSIN and FCOS routines can manage multiple instances of 2π . Both Applesoft routines process all phase angles appropriately and only in the first quadrant from 0 to $\pi/2$. As in the CX routines, the input and the output variables to CX2PL and PL2CX are dimensioned as DIM X(1), Z(1), where each variable is an array of two elements, 0 and 1.

The CXMATH Complex Array Routines

The AY routines process 2-by-2 complex floating point arrays where each array element is a complex floating point number. These 2-by-2 arrays are defined as three-dimensional arrays and the array variables are dimensioned as DIM X(1,1,1), Y(1,1,1), Z(1,1,1). I prefer to order the array dimensions with the REAL/IMAGINARY dimension first and followed by the 2-by-2 array dimension second, though this preference is totally arbitrary. However, ordering the dimensions in this order requires less code which yields faster processing. It is the CXMATH array routine that must calculate the physical address for each array element before it imposes relevant Applesoft processing. Applesoft calculates the physical address of each array element when it must access and print the floating point value for an array element. The REAL/IMAGINARY dimension could very well come second, after the 2-by-2 array dimension. My decision on dimension ordering is also based on how I wanted to present the variables in memory: for each array element, memory contains four complex floating point numbers, that is, four instances of a REAL floating point number followed by its companion IMAGINARY floating point number. I found that this memory organization is more general and far easier to manipulate. The AYINV routine requires a single input variable as a 2-by-2 complex floating point array X, and this routine produces two output variables, the complex floating point determinant Y and the 2-by-2 complex floating point inverse array Z. If either element of Y is zero, all elements of Z are set to 7.922816252E+28.

Managing Applesoft Program and Routine Memory Locations

A typical Applesoft program is shown in Figure 1 along with other potential Applesoft programs in order to explain the DOS CHAIN command in my publication *DOS 4.5 Volume and File Disk Management System Second Edition*. What is relevant in Figure 1 are the page-zero addresses in order to locate Simple Variables, Array Variables, and the Character String Pool in memory. The LG routines store their variables in the Simple Variables area. All other CXMATH routines store their variables in the Array Variables area of memory. As mentioned above, it is important to set HIMEM, and thus FRETOP, early in complex math processing to 0xB000 in order to protect the CXMATH routines. If it should ever become necessary to locate the variables of an Applesoft program, Table 1, Table 2, Table 3, and Figure 1 will provide all of the needed memory resources. Tables 1 and 2 are gentle reminders that only the first two characters of a

variable's name is stored and utilized by Applesoft. In other words, the variables TODAY and TOMORROW point to the very same floating point number whether that is desirable or not. Report No. 225 suggests that their system requirements, at times, include a purchased High Resolution Graphics Character Generator program that occupies memory at 0x0C00:0x0FFF and its data font arrays at 0x1000:0x13FF. I have written a number of utilities that utilize HIRES page 1 at 0x2000:0x3FFF for the display of upper and lower case characters on my original Apple][+ which did not contain a lower case character generator ROM. My CHARTBL only required 0x300 bytes of data for all printable ASCII characters. I assume the purchased program mentioned in Report No. 225 also displays additional characters besides the normal ASCII characters such as, perhaps, α, β, δ, η, φ, λ, μ, π, σ, τ, ω, etcetera. Having to include a graphics generator program would force the relocation of the main Applesoft program from 0x0800 to 0x4000. A simple Applesoft Loader EXEC file as shown in Figure 2 could easily accommodate this requirement and appropriately modify the page-zero variables PRGTAB and HIMEM that are shown in Figure 1.

Page-Zero Pointer Addresses	Start Program	Small Program	Problematic Program	Large Program
	0x0000	0x0000	0x0000	0x0000
PRGTAB - 0x67:0x68	0x0801	0x0801	0x0801	0x0801
	Start Applesoft Program	Small Chained Applesoft Program	Problematic Chained Applesoft Program	Large Chained Applesoft Program
PRGEND - 0xAF:0xB0 VARTAB - 0x69:0x6A	Simple Variables			
ARYTAB - 0x6B:0x6C	Array Variables			
STREND - 0x6D:0x6E				
	Free Space			
FRETOP - 0x6F:0x70				
HIMEM - 0x73:0x74	Character String Pool			
	0xFFFF	0xFFFF	0xFFFF	0xFFFF

Figure 1. Example Applesoft Program Layout in Memory

```
POKE 103,0
POKE 104,64
POKE 115,0
POKE 116,176
LOAD TEST AYMUL
RUN
```

Figure 2. EXEC File Applesoft Loader

```
LGNOT Inputs:
X% = 90
Z% = 0

LGNOT Outputs:
X% = 90
Z% = -91

]
```

Figure 3. LGNOT Example Processing

```
LGAND Inputs:
X% = 90
V% = 105
Z% = 0

LGAND Outputs:
X% = 90
V% = 105
Z% = 72

]
```

Figure 4. LGAND Example Processing

```
LGORA Inputs:
X% = 90
V% = 105
Z% = 0

LGORA Outputs:
X% = 90
V% = 105
Z% = 123

]
```

Figure 5. LGORA Example Processing

```
LGEOR Inputs:
X% = 90
V% = 105
Z% = 0

LGEOR Outputs:
X% = 90
V% = 105
Z% = 51

]
```

Figure 6. LGEOR Example Processing

Implementing The CXMATH Logical Operator Routines

The CXMATH logical operator routines LGNOT, LGAND, LGORA, and LGEOR are easy to implement for an assembly language programmer and not so easy to implement for an Applesoft programmer. While these

logical operator routines are still possible to program by the savvy Applesoft programmer, they would consume abundant program statements and data. Thus, these logical operators are provided by CXMATH because these operators are easy to implement and they consume few instructions and little data. Examples in using the CXMATH logical operators are shown in Figures 3 through 6. Figure 7 is a programmatically generated display of the Applesoft program for TEST LGAND. All of the logical operator test programs are similar to Figure 7 except for the utilization of the respective logical operator routine.

```
10 PRINT CHR$( 4 ); "BLOAD FPCXMATH,A$B000"  
20 CM = 45056  
40 PRINT CHR$( 4 ); "PR#0"  
50 X% = 90  
60 Y% = 105  
70 Z% = 0  
100 HOME  
110 PRINT : PRINT "LGAND Inputs:" : GOSUB 200  
120 CALL CM, "LGAND", X%, Y%, Z%  
130 PRINT : PRINT "LGAND Outputs:" : GOSUB 200  
140 END  
200 PRINT  
210 PRINT "X% = "; X%  
220 PRINT "Y% = "; Y%  
230 PRINT "Z% = "; Z%  
240 PRINT  
250 RETURN
```

Figure 7. TEST LGAND Applesoft Program

Implementing The CXMATH Complex Number Routines

The CXMATH complex floating point routines CXADD, CXSUB, CXMUL, and CXDIV can be implemented by an Applesoft programmer using a number of FOR/NEXT loops within a GOSUB subroutine-type structure, for example. Applesoft is certainly not the language of choice in order to easily implement subroutines and subprograms. However, program line numbers do afford that general capability. But Applesoft is an interpreted language in that a mathematical function or a TEXT operation is implemented only after its nouns, verb, and adjectives have been processed. When a GOSUB statement is encountered, that Applesoft statement is re-interpreted again and again and this consumes a vast amount of overhead. The CXMATH complex floating point routines are highly efficient and their assembly language instructions can easily access large sets of data. The CXMATH routines afford far better program organization in that these routines can be used in place, that is, where they are precisely needed without having to insert a GOSUB instruction whenever a complex floating point operation is required. An Applesoft programmer typically installs all subroutines and/or subprograms at the beginning of an Applesoft program in order to accelerate their utilization since an Applesoft program is driven entirely by the acquirement of line numbers. Again, the CXMATH routines are used *in situ* and their convenience is overwhelmingly beneficial. Examples in using the CXMATH complex floating point routines are shown in Figures 8 through 11. Figure 12 is a programmatically generated display of the Applesoft program for TEST CXMUL. All of the complex floating point test programs are similar to Figure 12 except for the utilization of the respective complex floating point routine.

```

CXADD Inputs:
X(0) = 12.34      X(1) = 87.65
Y(0) = 56.78      Y(1) = 43.21
Z(0) = 0          Z(1) = 0

CXADD Outputs:
X(0) = 12.34      X(1) = 87.65
Y(0) = 56.78      Y(1) = 43.21
Z(0) = 69.12      Z(1) = 130.86

]

```

Figure 8. CXADD Example Processing

```

CXSUB Inputs:
X(0) = 12.34      X(1) = 87.65
Y(0) = 56.78      Y(1) = 43.21
Z(0) = 0          Z(1) = 0

CXSUB Outputs:
X(0) = 12.34      X(1) = 87.65
Y(0) = 56.78      Y(1) = 43.21
Z(0) = -44.44     Z(1) = 44.44

]

```

Figure 9. CXSUB Example Processing

```

CXMUL Inputs:
X(0) = 12.34      X(1) = 87.65
Y(0) = 56.78      Y(1) = 43.21
Z(0) = 0          Z(1) = 0

CXMUL Outputs:
X(0) = 12.34      X(1) = 87.65
Y(0) = 56.78      Y(1) = 43.21
Z(0) = -3086.6913 Z(1) = 5509.9784

]

```

Figure 10. CXMUL Example Processing

```

CXDIV Inputs:
X(0) = 12.34      X(1) = 87.65
Y(0) = 56.78      Y(1) = 43.21
Z(0) = 0          Z(1) = 0

CXDIV Outputs:
X(0) = 12.34      X(1) = 87.65
Y(0) = 56.78      Y(1) = 43.21
Z(0) = .881547395 Z(1) = .872813263

]

```

Figure 11. CXDIV Example Processing

Given two complex floating point numbers $X = a + bi$ and $Y = c + di$, the processing for CXMATH complex floating point routines can be shown mathematically as follows.

For CXADD, $Z = X + Y = (a + bi) + (c + di) = a + c + bi + di = (a + c) + (b + d)i$

For CXSUB, $Z = X - Y = (a + bi) - (c + di) = a - c + bi - di = (a - c) + (b - d)i$

The multiplication of two complex floating point numbers is similar in multiplying two floating point polynomials. The following polynomial identity is used to multiply two complex floating point numbers:

$$(a + b) * (c + d) = ac + ad + bc + bd, \text{ therefore } (a + bi) * (c + di) = ac + adi + bci + bdi^2, i^2 = -1$$

For CXMUL, $Z = X * Y = (a + bi) * (c + di) = (ac - bd) + (ad + bc)i$

The division of two complex floating point numbers is similar in dividing two floating point polynomials.

$$Z = \frac{X}{Y} = \frac{(a + bi)}{(c + di)} = \frac{(a + bi)(c - di)}{(c + di)(c - di)} = \frac{(ac + bd) + (bc - ad)i}{c^2 + d^2} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}i$$

```

10 PRINT CHR$( 4 ); "BLOAD FPCXMATH,A$B000"
20 CM = 45056
30 DIM X( 1 ), Y( 1 ), Z( 1 )
40 PRINT CHR$( 4 ); "PR#0"
50 X( 0 ) = 12.34 : X( 1 ) = 87.65
60 Y( 0 ) = 56.78 : Y( 1 ) = 43.21
70 Z( 0 ) = 0 : Z( 1 ) = 0
100 HOME
110 PRINT : PRINT "CXMUL Inputs:" : GOSUB 200
120 CALL CM, "CXMUL", X( 0 ), Y( 0 ), Z( 0 )
130 PRINT : PRINT "CXMUL Outputs:" : GOSUB 200
140 END
200 PRINT
210 PRINT "X(0) = "; X( 0 ); : HTAB 20 : PRINT "X(1) = "; X( 1 )
220 PRINT "Y(0) = "; Y( 0 ); : HTAB 20 : PRINT "Y(1) = "; Y( 1 )
230 PRINT "Z(0) = "; Z( 0 ); : HTAB 20 : PRINT "Z(1) = "; Z( 1 )
240 PRINT
250 RETURN

```

Figure 12. TEST CXMUL Applesoft Program

```

CX2PL Inputs:
X(0) = 90          X(1) = 165
Z(0) = 0          Z(1) = 0

CX2PL Outputs:
X(0) = 90          X(1) = 165
Z(0) = 187.949461 Z(1) = 1.07144961

] *

```

Figure 13. CX2PL Example Processing

```

PL2CX Inputs:
X(0) = 123.456    X(1) = 1.12345
Z(0) = 0          Z(1) = 0

PL2CX Outputs:
X(0) = 123.456    X(1) = 1.12345
Z(0) = 53.403919 Z(1) = 111.307706

] *

```

Figure 14. PL2CX Example Processing

Implementing The CXMATH Conversion Routines

The CXMATH complex/polar conversion routines are highly convenient in order to convert a complex number into its polar coordinates or its polar coordinates into its complex number rather than using

Applesoft alone. Certainly, Applesoft can manage these conversions without much effort, yet the CXMATH routines do expedite these particular conversions in far less time. These complex/polar conversion routines can be implemented precisely where they are needed within a very complicated Applesoft program. Algorithms that require the summation of a number of coordinates can easily be implemented along with the other CXMATH complex floating point routines so that together they form a complete and comprehensive solution. Examples in using the CXMATH complex/polar conversion routines are shown in Figures 13 and 14. Figure 15 is a programmatically generated display of the Applesoft program for TEST CX2PL.

Given a complex number $X = a + bi$

$$Z(0) = \sqrt{a^2 + b^2}$$

$$Z(1) = \text{ATN}(b / a) \quad \text{when } a > 0$$

$$Z(1) = (\pi/2) * \text{SGN}(b) \quad \text{when } a = 0$$

$$Z(1) = \text{ATN}(b / a) + \pi \quad \text{when } a < 0$$

Given a polar coordinate $X = a, b$

$$Z(0) = a * \text{COS}(b)$$

$$Z(1) = a * \text{SIN}(b)$$

```

10 PRINT CHR$( 4 ); "BLOAD FPCXMATH,A$B000"
20 CM = 45056
30 DIM X( 1 ), Z( 1 )
40 PRINT CHR$( 4 ); "PR#0"
50 X( 0 ) = 90 : X( 1 ) = 165
60 Z( 0 ) = 0 : Z( 1 ) = 0
100 HOME
110 PRINT : PRINT "CX2PL Inputs:" : GOSUB 200
120 CALL CM, "CX2PL", X( 0 ), Z( 0 )
130 PRINT : PRINT "CX2PL Outputs:" : GOSUB 200
140 END
200 PRINT
210 PRINT "X(0) = "; X( 0 ); : HTAB 20 : PRINT "X(1) = "; X( 1 )
220 PRINT "Z(0) = "; Z( 0 ); : HTAB 20 : PRINT "Z(1) = "; Z( 1 )
230 PRINT
240 RETURN

```

Figure 15. TEST CX2PL Applesoft Program

Implementing The CXMATH Complex Array Routines

The CXMATH complex floating point array routines AYADD, AYSUB, AYMUL, and AYINV are the most difficult routines of all to implement even by the assembly language programmer let alone an Applesoft programmer. These routines involve following specific rules that govern how the coefficients of an array are ordered, accessed, and processed. The addition and subtraction of identically-sized arrays are the most straightforward of these routines. The multiplication of arrays, sized such that the number of rows of the first array must equal the number of columns of the second array, utilize the DOT product of complex floating point numbers. Therefore, the complex floating point array routines utilize the CXMATH complex floating point routines wherever possible. In other words, CXADD is used by AYADD, CXSUB is used by AYSUB, CXMUL is used by AYMUL, and CXMUL and CXDIV are both used by AYINV. Examples in using the

CXMATH complex floating point array routines are shown in Figures 16 through 19. Figure 20 is a programmatically generated display of the Applesoft program for TEST AYMUL. All of the complex floating point array test programs are similar to Figure 20 except for the utilization of the respective complex floating point array routine.

AYADD Test				
Inputs	0,0	0,1	1,0	1,1
Real X	90.1	85.2	80.3	75.4
Real Y	105.4	115.3	125.2	135.1
Real Z	0	0	0	0
Imag X	135.4	125.3	115.2	105.1
Imag Y	75.1	80.2	85.3	90.4
Imag Z	0	0	0	0
Outputs	0,0	0,1	1,0	1,1
Real X	90.1	85.2	80.3	75.4
Real Y	105.4	115.3	125.2	135.1
Real Z	195.5	200.5	205.5	210.5
Imag X	135.4	125.3	115.2	105.1
Imag Y	75.1	80.2	85.3	90.4
Imag Z	210.5	205.5	200.5	195.5

Figure 16. AYADD Example Processing

AYSUB Test				
Inputs	0,0	0,1	1,0	1,1
Real X	90.1	85.2	80.3	75.4
Real Y	105.4	115.3	125.2	135.1
Real Z	0	0	0	0
Imag X	135.4	125.3	115.2	105.1
Imag Y	75.1	80.2	85.3	90.4
Imag Z	0	0	0	0
Outputs	0,0	0,1	1,0	1,1
Real X	90.1	85.2	80.3	75.4
Real Y	105.4	115.3	125.2	135.1
Real Z	-15.3	-30.1	-44.9	-59.7
Imag X	135.4	125.3	115.2	105.1
Imag Y	75.1	80.2	85.3	90.4
Imag Z	60.3	45.1	29.9	14.7

Figure 17. AYSUB Example Processing

AYMUL Test				
Inputs	0,0	0,1	1,0	1,1
Real X	90.1	85.2	80.3	75.4
Real Y	105.4	115.3	125.2	135.1
Real Z	0	0	0	0
Imag X	135.4	125.3	115.2	105.1
Imag Y	75.1	80.2	85.3	90.4
Imag Z	0	0	0	0
Outputs	0,0	0,1	1,0	1,1
Real X	90.1	85.2	80.3	75.4
Real Y	105.4	115.3	125.2	135.1
Real Z	-693.050002	-287.149999	287.150002	705.050008
Imag X	135.4	125.3	115.2	105.1
Imag Y	75.1	80.2	85.3	90.4
Imag Z	43992.79	47467.75	37762.75	40737.79

Figure 18. AYMUL Example Processing

AYINV Test				
Inputs	0,0	0,1	1,0	1,1
Real X	90.1	85.2	80.3	75.4
Real Y	105.4	115.3	125.2	135.1
Real Z	0	0	0	0
Imag X	135.4	125.3	115.2	105.1
Imag Y	75.1	80.2	85.3	90.4
Imag Z	0	0	0	0
		V(0) = 0	V(1) = 0	
Outputs	0,0	0,1	1,0	1,1
Real X	90.1	85.2	80.3	75.4
Real Y	105.4	115.3	125.2	135.1
Real Z	-.142358363	.18124108	.161799721	-.200682438
Imag X	135.4	125.3	115.2	105.1
Imag Y	75.1	80.2	85.3	90.4
Imag Z	.493068827	-.573214835	-.533141832	.61328784
		V(0) = 156.000004	V(1) = -197.959999	

Figure 19. AYINV Example Processing

Given three complex floating point arrays X, Y, and Z, the CXMATH complex floating point array routines can be shown mathematically as follows.

$$X = \begin{bmatrix} a + bi & c + di \\ e + fi & g + hi \end{bmatrix} \quad Y = \begin{bmatrix} j + ki & l + mi \\ n + oi & p + qi \end{bmatrix} \quad Z = \begin{bmatrix} r + si & t + ui \\ v + wi & x + yi \end{bmatrix}$$

For AYADD, $Z = X + Y$

$$r = a + j, s = b + k, t = c + l, u = d + m, v = e + n, w = f + o, x = g + p, y = h + q$$

For AYSUB, $Z = X - Y$

$$r = a - j, s = b - k, t = c - l, u = d - m, v = e - n, w = f - o, x = g - p, y = h - q$$

For AYMUL, $Z = X * Y$ using the dot product of X and Y

$$\begin{aligned}(r + si) &= (a + bi) * (j + ki) + (c + di) * (n + oi) \\(t + ui) &= (a + bi) * (l + mi) + (c + di) * (p + qi) \\(v + wi) &= (e + fi) * (j + ki) + (g + hi) * (n + oi) \\(x + yi) &= (e + fi) * (l + mi) + (g + hi) * (p + qi)\end{aligned}$$

```
10 PRINT CHR$( 4 ); "BLOAD FPCXMATH,A$B000"
20 CM = 45056
30 DIM X( 1, 1, 1 ), Y( 1, 1, 1 ), Z( 1, 1, 1 )
40 PRINT CHR$( 4 ); "PR#3"
50 X( 0, 0, 0 ) = 90.1 : X( 1, 0, 0 ) = 135.4
52 X( 0, 0, 1 ) = 85.2 : X( 1, 0, 1 ) = 125.3
54 X( 0, 1, 0 ) = 80.3 : X( 1, 1, 0 ) = 115.2
56 X( 0, 1, 1 ) = 75.4 : X( 1, 1, 1 ) = 105.1
60 Y( 0, 0, 0 ) = 105.4 : Y( 1, 0, 0 ) = 75.1
62 Y( 0, 0, 1 ) = 115.3 : Y( 1, 0, 1 ) = 80.2
64 Y( 0, 1, 0 ) = 125.2 : Y( 1, 1, 0 ) = 85.3
66 Y( 0, 1, 1 ) = 135.1 : Y( 1, 1, 1 ) = 90.4
70 Z( 0, 0, 0 ) = 0 : Z( 1, 0, 0 ) = 0
72 Z( 0, 0, 1 ) = 0 : Z( 1, 0, 1 ) = 0
74 Z( 0, 1, 0 ) = 0 : Z( 1, 1, 0 ) = 0
76 Z( 0, 1, 1 ) = 0 : Z( 1, 1, 1 ) = 0
100 HOME
110 PRINT : PRINT "AYMUL Test:" : PRINT "Inputs"; : HTAB 10 : PRINT "0,0"; : HTAB 28 :
    PRINT "0,1"; : HTAB 46 : PRINT "1,0"; : HTAB 64 : PRINT "1,1" : GOSUB 200
120 CALL CM, "AYMUL", X( 0, 0, 0 ), Y( 0, 0, 0 ), Z( 0, 0, 0 )
130 PRINT : PRINT "Outputs:"; : HTAB 10 : PRINT "0,0"; : HTAB 28 : PRINT "0,1"; :
    HTAB 46 : PRINT "1,0"; : HTAB 64 : PRINT "1,1" : GOSUB 200
140 END
200 PRINT "-----"
205 A$ = "Real" : FOR I = 0 TO 1 : IF I = 1 THEN A$ = "Imag"
210 PRINT A$; " X"; : HTAB 10 : PRINT X( I, 0, 0 ); : HTAB 28 : PRINT X( I, 0, 1 );
215 HTAB 46 : PRINT X( I, 1, 0 ); : HTAB 64 : PRINT X( I, 1, 1 )
220 PRINT A$; " Y"; : HTAB 10 : PRINT Y( I, 0, 0 ); : HTAB 28 : PRINT Y( I, 0, 1 );
225 HTAB 46 : PRINT Y( I, 1, 0 ); : HTAB 64 : PRINT Y( I, 1, 1 )
230 PRINT A$; " Z"; : HTAB 10 : PRINT Z( I, 0, 0 ); : HTAB 28 : PRINT Z( I, 0, 1 );
235 HTAB 46 : PRINT Z( I, 1, 0 ); : HTAB 64 : PRINT Z( I, 1, 1 )
240 PRINT : NEXT
250 RETURN
```

Figure 20. TEST AYMUL Applesoft Program

For AYINV, $Y = \text{DET}(X)$ and $Z = X^{-1}$ where Y is a complex number and is the determinant of X , and Z is a complex floating point array and is the inverse of X .

$$Y = \text{DET}(X) = (a + bi) * (g + hi) - (c + di) * (e + fi)$$

$$Z = X^{-1} = \frac{1}{\text{DET}(X)} * \begin{bmatrix} g + hi & -(c + di) \\ -(e + fi) & a + bi \end{bmatrix}$$

The Identity Matrix I is the matrix equivalent of the number 1. When an array is multiplied by I , the original array is obtained. When an array is multiplied by its inverse array, the Identity Matrix is obtained. This is the verification test that is used for the AYINV floating point array routine. The results that are obtained for the Z array as shown in Figure 19 are used to initialize the inputs for the Y array and the inputs for the X array are the same as shown in Figure 19. The TEST AYMUL Applesoft program is used to confirm the Z array elements. The results of this verification test are shown in Figure 21 and the REAL coefficients 0,0 and 1,1 for the Z array are essentially equal to 1 and all other coefficients of the Z array are essentially equal to 0.

```

AYINV Verify Test
Inputs  0,0          0,1          1,0          1,1
-----
Real X  90.1        85.2          80.3          75.4
Real Y  -142358363    .18124108     .161799721   -200682438
Real Z  0             0             0             0
-----
Imag X  135.4         125.3         115.2         105.1
Imag Y  .493068827     -573214835   -533141832   61328784
Imag Z  0             0             0             0
-----
Outputs 0,0          0,1          1,0          1,1
-----
Real X  90.1        85.2          80.3          75.4
Real Y  -142358363    .18124108     .161799721   -200682438
Real Z  1.00000009    -1.49011612E-07  5.96046448E-08 .999999881
-----
Imag X  135.4         125.3         115.2         105.1
Imag Y  .493068827     -573214835   -533141832   61328784
Imag Z  -5.96046448E-08  8.94069672E-08 -5.96046448E-08 7.4505806E-08

```

Figure 21. TEST AYINV VERIFY Processing

```

CSMUL Test
Inputs  0,0          0,1          1,0          1,1
-----
Real X  90.1        85.2          80.3          75.4
Real Z  0             0             0             0
-----
Imag X  135.4         125.3         115.2         105.1
Imag Z  0             0             0             0
-----
          Y(0) = 105.4          Y(1) = 75.1
Outputs 0,0          0,1          1,0          1,1
-----
Real X  90.1        85.2          80.3          75.4
Real Z  -671.999996    -429.950002   -187.899996   54.1500028
-----
Imag X  135.4         125.3         115.2         105.1
Imag Z  21037.67       19605.14     18172.61     16740.08
-----
          Y(0) = 105.4          Y(1) = 75.1

```

Figure 22. CSMUL Example Processing

The CXMATH complex floating point scalar multiplication routine CSMUL utilizes the CXMUL routine in order to multiply each of the array coefficients of X with a complex floating point scalar variable Y. There is no special row or column ordering rules to follow in multiplying a scalar value with each of the coefficients of an array. An example in using the CXMATH complex floating point scalar routine is shown in Figure 22. Figure 23 is a programmatically generated display of the Applesoft program for TEST CSMUL.

Given the complex floating point array definitions for X and Z above and let $Y = (j + ki)$.

For CSMUL, $Z = X * Y$ using the scalar product of X and Y

$$\begin{aligned}(r + si) &= (a + bi) * (j + ki) \\(t + ui) &= (c + di) * (j + ki) \\(v + wi) &= (e + fi) * (j + ki) \\(x + yi) &= (g + hi) * (j + ki)\end{aligned}$$

```

10 PRINT CHR$( 4 ); "BLOAD FPCXMATH,A$B000"
20 CM = 45056
30 DIM X( 1, 1, 1 ), Y( 1 ), Z( 1, 1, 1 )
40 PRINT CHR$( 4 ); "PR#3"
50 X( 0, 0, 0 ) = 90.1 : X( 1, 0, 0 ) = 135.4
52 X( 0, 0, 1 ) = 85.2 : X( 1, 0, 1 ) = 125.3
54 X( 0, 1, 0 ) = 80.3 : X( 1, 1, 0 ) = 115.2
56 X( 0, 1, 1 ) = 75.4 : X( 1, 1, 1 ) = 105.1
60 Y( 0 ) = 105.4 : Y( 1 ) = 75.1
70 Z( 0, 0, 0 ) = 0 : Z( 1, 0, 0 ) = 0
72 Z( 0, 0, 1 ) = 0 : Z( 1, 0, 1 ) = 0
74 Z( 0, 1, 0 ) = 0 : Z( 1, 1, 0 ) = 0
76 Z( 0, 1, 1 ) = 0 : Z( 1, 1, 1 ) = 0
100 HOME
110 PRINT : PRINT "CSMUL Test:" : PRINT "Inputs"; : HTAB 10 : PRINT "0,0"; : HTAB 28 :
    PRINT "0,1"; : HTAB 46 : PRINT "1,0"; : HTAB 64 : PRINT "1,1" : GOSUB 200
120 CALL CM, "CSMUL", X( 0, 0, 0 ), Y( 0 ), Z( 0, 0, 0 )
130 PRINT : PRINT "Outputs:"; : HTAB 10 : PRINT "0,0"; : HTAB 28 : PRINT "0,1"; :
    HTAB 46 : PRINT "1,0"; : HTAB 64 : PRINT "1,1" : GOSUB 200
140 END
200 PRINT "-----"
205 A$ = "Real" : FOR I = 0 TO 1 : IF I = 1 THEN A$ = "Imag"
210 PRINT A$; " X"; : HTAB 10 : PRINT X( I, 0, 0 ); : HTAB 28 : PRINT X( I, 0, 1 );
215 HTAB 46 : PRINT X( I, 1, 0 ); : HTAB 64 : PRINT X( I, 1, 1 )
220 PRINT A$; " Z"; : HTAB 10 : PRINT Z( 1, 0, 0 ); : HTAB 28 : PRINT Z( I, 0, 1 );
225 HTAB 46 : PRINT Z( I, 1, 0 ); : HTAB 64 : PRINT Z( I, 1, 1 )
230 PRINT : NEXT
240 HTAB 15 : PRINT "Y(0) = "; Y( 0 ); : HTAB 40 : PRINT "Y(1) = "; Y( 1 )
250 RETURN

```

Figure 23. TEST CSMUL Applesoft Program

Name	LOADER	LIBENT	LINKER	Routine	Total CALL
LGNOT	1800	652	1097	42	1842
LGAND	2357	641	1665	61	2418
LGORA	2346	630	1665	61	2407
LGEOR	2335	619	1665	58	2393
CXADD	8207	608	7548	1027	9234
CXSUB	8196	597	7548	1104	9300
CXMUL	8185	586	7548	10454	18639
CXDIV	8174	575	7548	20955	29129
CX2PL	5621	575	4995	94617	100238
PL2CX	5610	564	4995	53950	59560
AYADD	15677	542	15084	4070	19747
AYSUB	15666	531	15084	5049	20715
AYMUL	15655	520	15084	88862	104517
AYINV	13121	509	12548	99733	112854
CSMUL	13121	498	12572	42397	55518

Table 21. CXMATH Library CPU Cycles

Name	Size	Description
LOADER	0x37	Initialize FRETOP first time; process selected routine
LIBENT	0x43	Parse for routine name, create hash, obtain index to tables
LINKER	0x4B	Parse for variables, save addresses of input/output data
PRNTERR	0x88	Prints all error messages, includes four error messages
LGNOT	0x10	Calculates the NOT logical operator
LGAND	0x1A	Calculates the AND logical operator
LGORA	0x1A	Calculates the OR logical operator
LGEOR	0x18	Calculates the EOR logical operator
CXADD	0x2A	Calculates the addition of two complex floating point numbers
CXSUB	0x2A	Calculates the subtraction of two complex floating point numbers
CXMUL	0x62	Calculates the multiplication of two complex floating point numbers
CXDIV	0xBD	Calculates the division of two complex floating point numbers
CX2PL	0x7D	Calculates the polar coordinates from a complex floating point number
PL2CX	0x30	Calculates a complex floating point number from polar coordinates
AYADD	0x15	Calculates the addition of two complex floating point arrays
AYSUB	0x15	Calculates the subtraction of two complex floating point arrays
BLDARGS1	0x30	Calculates the next set of coordinates for AYADD and AYSUB
AYMUL	0x60	Calculates the multiplication of two complex floating point arrays
BLDARGS2	0x59	Calculates the next set of coordinates for AYMUL
AYINV	0xE7	Calculates the determinate and the inverse of a complex floating point array
BLDARGS3	0x59	Calculates the next set of coordinates for the AYINV determinate
BLDARGS4	0x47	Calculates the next set of coordinates for the AYINV inverse array
CSMUL	0x15	Calculates the multiplication of a complex scalar and a complex array
BLDARGS5	0x47	Calculates the next set of coordinates for CSMUL

Table 22. CXMATH Routine Sizing

Comparing The CXMATH Library And The NRAO Library

The number of CPU cycles that each of the CXMATH complex floating point routines and subroutines require in order to complete its processing is shown in Table 21. The LOADER routine, depending on the processing of the LIBENT and LINKER routines, can take from 1800 cycles to 15677 cycles. The LIBENT routine creates a hash value from the provided five-letter ASCII routine name. However, depending on the location of that hash value within the HASHTBL table, LIBENT can take from 498 cycles to 652 cycles to generate and process that hash value in order to determine a CMDNDX index value. The processing time for LINKER depends on the number and the complexity of the variables that follow the provided routine name. This processing time ranges from 1097 cycles for the simple variables that follow the LGNOT routine to 15084 cycles for the very complex variables that follow the AYADD and similar routines. Even Applesoft must locate the descriptor for a referenced variable by processing the first two letters of the variable's name and utilizing the descriptor's address, the extracted offset, the extracted number of dimensions, and the extracted beginning dimension of each dimension set in order to calculate the final address where the requested data for that variable is located in memory. This is an immensely difficult processing task and the processing must be done precisely. Even if Applesoft were to also perform the complex floating point processing, the specialty of these CXMATH routines, these same highly complex Applesoft address extraction routines would be part of the general overhead that is required in order to manipulate the data for a single or for multiple complex floating point numbers. Only when there is an error condition would LIBENT or LINKER utilize the PRNTERR routine, and in those cases, the desired CXMATH routine would never begin its processing.

The size in bytes for each CXMATH complex floating point routine and subroutine is shown in Table 22. Table 22 also includes a brief description of each routine and subroutine. The logical operator routines, the add and subtract routines, and the scalar multiply routine all require a small number of instructions in order to fulfill their processing requirements. Only when multiply and divide operations are needed do the CXMATH complex floating point routines become far more complicated. The BLDARGn subroutines also increase in complexity as its respective CXMATH routine increases in complexity. The complex floating point routines are designed to be as efficient as possible rather than using as few instructions as possible.

Truly, the unique processing of LGNOT, LGAND, LGORA, and LGEOR only require a few CPU cycles of processing in order to achieve the desired mathematical output from these selected CXMATH routines compared to the general overhead that these routines require before their data processing can even commence. For these four CXMATH routines, the CPU cycles that are shown in Table 21 gives the number of cycles each routine requires only for its unique processing and for its total processing in order to complete its CALL statement. In other words, only about 2.3% to 2.5% of the processing duration for these four CXMATH routines is actually spent doing the intended work. The processing cycles that are spent in LOADER overhead is used to generate the address for the referenced variables that are included in the Applesoft CALL statement. As the complexity increases for the intended work by the CXMATH routines, fewer comparative cycles are spent for overhead. For the CXADD and CXSUB routines, about 11.0% and 11.8% of their respective processing cycles are spent for their unique processing, thus fewer comparative cycles are spent for overhead. CXMUL spends about 56.0% of its processing cycles multiplying and adding the coefficients of complex floating point numbers and CXDIV spends about 72.7% of its processing cycles multiplying, adding, and dividing the coefficients of complex floating point numbers. For these two routines, the LOADER overhead is virtually the same as for the CXADD and CXSUB routines. The LOADER overhead ranges from 8174 to 8207 cycles for these four CXMATH routines. The conversion routine CX2PL spends about 94.4% of its processing cycles multiplying, adding, taking the square root, dividing, and taking the arctangent of the coefficients of a complex floating point number and its LOADER overhead is only 5621 cycles. The conversion routine PL2CX spends about 92.3% of its processing cycles

multiplying, taking the cosine, or taking the sine of the given polar coordinates and its LOADER overhead is only 5610 cycles. Because only one input and only one output variable is required for these conversion routines, the LOADER overhead is substantially reduced.

The CXMATH array and scalar routines utilize specific subroutines in order to initialize their REAL and their IMAGINARY page-zero addresses before calling the included complex add, subtract, multiply, and divide routines whenever they are required. Normally, LINKER adds 0x05 to the REAL variable address in order to initialize the IMAGINARY variable address of a complex number. In the array routines, each array requires four complex numbers, one complex number for each array coefficient. As the various coefficients are added or subtracted in AYADD and AYSUB, respectively, BLDARGS1 initializes its page-zero addresses so that these specific calculations can utilize the intended coefficients according to the rules for manipulating arrays. The processing cycles that are shown in Table 21 for these two routines includes the specific processing cycles for AYADD or AYSUB and the total processing cycles includes LOADER and BLDARGS1. These routines spend about 20.5% and 24.4% of their processing cycles either adding or subtracting complex arrays, respectively. The AYMUL routine uses BLDARGS2 in order to initialize its page-zero addresses before the routine manipulates the intended complex floating point coefficients according to the rules for multiplying two arrays. This routine spends about 85.0% of its processing cycles multiplying and adding complex floating point variables in order to multiply two complex arrays. The AYINV routine actually delivers two products after processing the coefficients of one complex array. Those two products include a determinant which is a complex number and the inverse of the input array which is also an array. When an array is multiplied by its inverse array, the Identity Matrix is produced. The AYINV routine uses BLDARGS3 in order to compute the determinant and it uses BLDARGS4 in order to compute the inverse array. The AYINV routine spends about 88.7% of its processing cycles producing a determinant and an inverse array. The CSMUL routine uses BLDARGS5 in order to initialize its page-zero addresses so that included CXMATH routines can utilize the intended coefficients of the input variables and compute the multiplication of a complex floating point scalar with each coefficient of an array of complex floating point numbers. CSMUL spends about 76.3% of its processing cycles producing its output product.

Name	Cycle Time	Loop Time	NRAO Time	Size	NRAO Size
LOADER				0x37	0x21
LIBENT				0x43	0x71
LINKER				0x4B	0x4E
LGAND	2369	3280		0x1A	0x18
LGORA	2359	3280		0x1A	0x1B
LGEOR	2345	3280		0x18	0x1B
CXADD	9049	10000	10000	0x2A	0x41
CXMUL	18265	19200	17500	0x62	0x85
CXDIV	28544	29500	26000	0xBD	0xD0
CX2PL	98226	98950	100000	0x7D	0xC9
PL2CX	58364	59100	62000	0x30	0x4E
AYADD	19351	20850	16000	0x15	0x4F
AYSUB	20299	22100	17000	0x15	0x4F
AYMUL	102419	103600	62000	0xB9	0x1D0
AYINV	110589	111700	62000	0x187	0x1C1

Table 23. CXMATH Routine Sizing and Timing Comparison to NRAO

NRAO Reports No. 225 and No. 226 include sizing and timing information for some of its library routines. Report No. 226 even provides an example Applesoft test program that calls a library routine 1500 times in order to extract timing information. The author subtracts the FOR/NEXT statement overhead and determines the time the library routine processes its CALL statement by dividing the total processing time by 1500. This procedure would certainly provide a satisfactory value that would include some Applesoft overhead, library function overhead, and specific routine processing time. Table 23 compares the sizing in bytes and the timing in microseconds for the CXMATH routines and for those NRAO routines when that information is available in either Report No. 225 or Report No. 226. I converted CPU cycles that are reported in Table 21 to microseconds by dividing CPU cycles by the *average* microprocessor speed in the Apple][computer which is 1.020484 MHz and that topic is presented in detail in *DOS 4.5 Volume and File Disk Management System Second Edition*. The converted CPU cycles is reported as Cycle Time in Table 23. I also constructed an Applesoft FOR/NEXT loop, determined the FOR/NEXT statement overhead, and included those timing measurements in Table 23 as Loop Time. For the faster routines, I ran 5000 loops and for the slower routines I ran 2000 loops. My loop timings are consistently a little longer than my CPU cycle timings. My timings appear to be consistent with the NRAO published timings except for the timings for AYMUL and for AYINV. I am very impressed that the NRAO could achieve such outstanding and remarkable processing speeds particularly for their AYMUL and AYINV routines.

I am somewhat skeptical of a few of the sizing values for the NRAO routines such as CXADD, CXMUL, AYADD, and AYSUB. My CXADD routine utilizes only page-zero pointers and I use LOADFAC, FADD, and COPYFAC for both the REAL coefficients and for the IMAGINARY coefficients. I cannot understand why the NRAO CXADD routine requires 0x41 bytes for a routine that only requires 0x2A bytes at most. What would the NRAO be doing with those additional 0x17 bytes for such a simple routine that just adds two floating point numbers for the REAL coefficients and two floating point numbers for the IMAGINARY coefficients? The NRAO logical operator routines do include a call to LINK before its coefficients are processed. Why? I call my LINKER routine, which is comparable to the NRAO LINK routine, early in routine processing and not during coefficient processing. Perhaps, then, the NRAO CXADD routine includes other calls as well. The NRAO CX2PL routine seems excessively large in my opinion for the type of processing that this routine is required to perform. In other words, taking the square root of two squared floating point numbers, dividing two floating point numbers, and taking the arctangent of that quotient should not require 0xC9 bytes. Both of the NRAO AYADD and AYSUB routines do not possibly require 0x4F bytes for their processing when all four complex numbers of both arrays can simply be added by using four calls to an included routine like CXADD. The most troublesome routines are the NRAO AYMUL and AYINV routines for their processing size and for their process timing. Both of these routines are the most complex, yet they can easily be processed by using multiple calls to CXMUL and CXDIV. Is it because these routines are so huge in size that they are able to process nearly twice as fast as the equivalent CXMATH routines? I probably think so. It truly would be most interesting to me to be able to compare my code logic with the code logic that the NRAO designed and developed for their complex floating point library.

And yet, if the NRAO library was actually available for inspection, perhaps I might not have been so inclined to build the CXMATH library and miss the valuable opportunity I was given to document Using Complex Numbers in an Apple][Computer.

Adding An LN(*aexpr*) Statement To Applesoft

The Apple][Applesoft interpreter is designed to process 107 Applesoft statements, or commands. All of these commands fall into four unique categories: general statements, function statements, special function statements, and operator statements. Each statement is assigned a specific token identification number that ranges from 0x80 for the END statement to 0xEA for the MID\$ statement. Thus, an Applesoft program

consists of Applesoft tokens that have their MSB set ON and all other integer numbers, floating point numbers, and ASCII character data have their MSB set OFF. Processing any Applesoft program that has any ASCII character data with its MSB set ON will result in failure. That is why the Cornelis Bongers garbage collection routine, for example, must take great care in never accidentally leaving the MSB of any ASCII character data set ON.

The four categories that Applesoft statements fall into are determined by the statement's usage. All 64 general statements typically require no additional parameter. If a parameter is required, the statement contains an = equal sign such as HCOLOR= and SCALE=. The parameter would simply follow the Applesoft statement. All of the addresses that are given for general statements are designed to be pushed onto the stack and statement processing begins with the very next RTS instruction. The token identification number ranges from 0x80 to 0xBF for general statements. The processing address index for general statements is determined by subtracting 0x80 from the token identification number and multiplying that difference by two. The table of processing addresses for general statements begins at 0xD000 and ends at 0xD07F. All 22 function statements require a single parameter, enclosed in parenthesis, and the parameter becomes part of the statement. Examples of function statements are SGN(A), INT(A), and ABS(A). The 3 special function statements require two or more parameters, enclosed in parenthesis, each parameter separated by a , comma, and the parameters become part of the statement. An example of a special function statement is LEFT\$(A\$,B). All of the addresses that are given for function and special function statements are designed to be copied to page-zero as the address for a JMP instruction at 0x90. The token identification number ranges from 0xD2 to 0xEA for function and special function statements. The processing address index for function and special function statements is determined by subtracting 0x80 from the token identification number, multiplying that difference by two, and subtracting 0x24. The table of processing addresses for function and special function statements begins at 0xD080 and ends at 0xD0B1. The 10 operator statements are processed uniquely from a table of thirty entries where the first entry contains a tag value and the second and third entries is the processing address for that operator. Example operator statements are the + plus operator and the AND operator. The token identification number ranges from 0xC8 to 0xD1 for operator statements. The table of tags and processing addresses for operator statements begins at 0xD0B4 and ends at 0xD0D1. The remaining 8 statements include example statements such as TO, AT, and STEP. These remaining statements are processed by virtue of their token identification number alone which ranges from 0xC0 to 0xC7.

An Applesoft token identification number is assigned to an Applesoft statement by the Applesoft interpreter while the interpreter scans a NULL-terminated table of Applesoft statements in DCI format. When an Applesoft statement is located within this ASCII data table that ranges from 0xD0D0 to 0xD25F, its statement count number beginning with zero is added to 0x80. As each Applesoft line is entered onto the Apple command line, all Applesoft statements are tokenized that are within that command line. Those tokens along with all other numerical and ASCII data are entered into memory that is prefaced by a line number and the number of bytes to the next Applesoft line number. The Applesoft program is terminated by three NULL bytes. Every time this Applesoft program is restored into memory, each of its Applesoft tokens are processed exactly in the same manner as they are intended according to their token identification number. The token identification number 0xBA, for example, will always signify PRINT. It is generally not possible to add any additional statements to Applesoft in order to expand its repertoire.

I have removed the cassette WRITE routine from Apple //e Monitor ROM software among other ROM routines in preference for ROM features that I would rather utilize. In removing the WRITE routine, I have also disabled the SHLOAD, RECALL, STORE, and SAVE Applesoft statements as well as the GETARYPT routine at 0xF7D9. I have restored the LOAD Applesoft statement in order to support the Egan Ford Insta-Disk routines that read and process high frequency AIFF data files. I have simply replaced the processing

address for the SHLOAD, RECALL, STORE, and SAVE Applesoft statements with the address of IORTS at 0xFF58. If the token identification number for these four Applesoft statements are ever encountered and processed, they will return immediately without error or data. If I should replace any of these four Applesoft statement names with another statement name in DCI format, the processing for that statement cannot accept a parameter in parenthesis such as LN(*aexpr*). The Applesoft interpreter begins a scan of the Applesoft statement ASCII data table at 0xD590. The interpreter separates the general statements from all other Applesoft statements at 0xD82A where the routine pushes the general statement processing address minus one onto the stack for processing to begin with the very next RTS instruction. The function statements are separated from the special function statements at 0xDF0C. Parenthesis and parameter processing is handled at 0xDEB2.

A new Applesoft statement such as LN(*aexpr*) cannot be added to the function statements after CHR\$ as that would cause the Applesoft interpreter to process the LEFT\$ statement on the Apple command line and generate an 0xE9 token identification number instead of its correct 0xE8 token identification number. If the LN(*aexpr*) Applesoft statement is added after the last Applesoft statement MID\$, the LN(*aexpr*) statement would be processed as a special function statement and that would generate an error message because a comma would be expected within the parenthesis of its parameter. Obviously, the routine at 0xDF0C must be modified in order to process the new Applesoft token identification number 0xEB for LN(*aexpr*) as a function statement rather than as a special function statement. The code for the 0xDF0C routine pushes the offset of the processing address, that is based on token identification number, using BASADDR or 0xD000 minus 0x24 as its base address. Unfortunately, the routine must also push the integer values for the additional parameters of a special function statement onto the stack and the initial offset to BASADDR that was pushed first must be pulled, pushed again, and pulled again. Using a page-zero location to save that initial offset to BASADDR would certainly tidy up this coding extravaganza and allow the insertion of another immediate compare and branch instruction pair that requires four bytes. Actually, the resurrected routine comes with an additional byte as well that is unused at 0xDF4E. Now, the LN(*aexpr*) statement provides the same natural log value that the previous LOG(*aexpr*) statement generated so that the LOG(*aexpr*) statement now generates the correct base-10 log value according to $\log(x) = \ln(x) * \log(e)$. The new base-10 log function processing is done at 0xEAD7 which only requires the five-byte floating point value of log(e) at 0xEAF2.

Summary

As in all investigations, the learning curve is initially steep. A great effort was employed by Microsoft to design a BASIC for one of the early microprocessors and to port that software package to other microprocessors and their platforms. In the software ports that I witnessed at Sierra On-Line, most of the software was portable and some of the software was not portable. I have no doubt that some concessions were made when Microsoft ported their 8080 BASIC to the 6502 microprocessor and specifically to the Apple][platform. The Applesoft code is highly compacted which contributes to the difficulty in exploring its routines and functions. Only when I designed my own multiply routine did I fully appreciate how elegantly Microsoft designed their multiply routine. Keeping fundamental routines at their established memory location is another contribution to the difficulty in investigating alternative processing choices. Would it really be worth the time and the effort to verify improvements to floating point handling if guard bytes were also pushed onto and popped from the stack? Certainly, there are still remaining routines that continue to interest and amaze me that propel my exploration into the further depths of Applesoft. Yes, there is bad code as well as brilliant code since Applesoft is a collection of routines written by a collection of very different individuals. I would hope that eventually, someday, Apple Computer, Inc., may actually publish the Applesoft source code though the number of interested parties is exponentially dwindling.