

My Applesoft JourneyTM

Walland Philip Vrbancic, Jr.

April 20, 2025

Table of Contents

Introduction to My Applesoft Journey™	1
Understanding the Deficiencies in	2
Applesoft Mathematical Routines and Functions	2
Applesoft Variables.....	3
Applesoft Floating-Point Variables	7
Management of Applesoft Floating-Point Registers.....	8
Introduction to Applesoft Source Code	10
The Applesoft Statements	11
The Applesoft Interpreter.....	15
String and Numeric Variables.....	25
Floating-Point Arithmetic Operations.....	35
Transcendental Arithmetic Operations	43
Applesoft Initialization & Miscellaneous Statements.....	50
Management of LORES and HIRES Graphics	52
Derived Transcendental Arithmetic Operations.....	64
Testing Applesoft Floating-Point Routines.....	68
Installing Applesoft.....	77
Appendix A	81
Appendix B	87
Appendix C	91
Appendix D.....	101
SHLOAD Command.....	102
SHSAVE Command.....	103

List of Figures

Figure 1. Example Applesoft Program Layout in Memory	4
Figure 2. Vertical Coordinate Conversion to GBAS	56
Figure 3. Unmodified Applesoft DRAWCMD	61
Figure 4. Modified Applesoft DRAWCMD	61
Figure 5. Test 1 Applesoft Program.....	68
Figure 6. Test 1 Unmodified Applesoft	69
Figure 7. Test 1 Modified Applesoft.....	69
Figure 8. Test 2 Unmodified Applesoft	70
Figure 9. Test 2 Modified Applesoft.....	70
Figure 10. Test 3 Unmodified Applesoft	70
Figure 11. Test 3 Modified Applesoft.....	70
Figure 12. Test 4 Unmodified Applesoft	71
Figure 13. Test 4 Modified Applesoft.....	71
Figure 14. Test 5 Unmodified Applesoft	72
Figure 15. Test 5 Modified Applesoft.....	72
Figure 16. Test 6 Unmodified Applesoft	72
Figure 17. Test 6 Modified Applesoft.....	72
Figure 18. Test 7 Unmodified Applesoft	74
Figure 19. Test 7 Modified Applesoft.....	74
Figure 20. Test 7.1 Unmodified Applesoft	74
Figure 21. Test 7.1 Modified Applesoft.....	74
Figure 22. Test 8 Unmodified Applesoft	74
Figure 23. Test 8 Modified Applesoft.....	74
Figure 24. Test 9 Unmodified Applesoft	75
Figure 25. Test 9 Modified Applesoft.....	75
Figure 26. Test 10 Unmodified Applesoft	76
Figure 27. Test 10 Modified Applesoft.....	76
Figure 28. Test 11 Unmodified Applesoft	76
Figure 29. Test 11 Modified Applesoft.....	76
Figure 30. BLDROM EXEC File	78
Figure 31. BLDV2ROM EXEC File.....	78
Figure 32. SETUP Command File	79

List of Tables

Table 1. Simple Variable Descriptor Definitions in Applesoft.....	5
Table 2. Array Variable Descriptor Definitions in Applesoft.....	5
Table 3. Single Array Element Descriptor Definitions in Applesoft.....	6
Table 4. Routines That Copy Floating-Point Registers or Numbers	9
Table 5. General Layout of the C0:FF ROM	10
Table 6. Bongers Simple Variable Descriptor Processing in Pass 1	31
Table 7. Bongers Array Variable Element Processing in Pass 1	31
Table 8. Applesoft Natural Log Routine Polynomials.....	37
Table 9. Applesoft Exponential Function Polynomials	44
Table 10. Applesoft Sine Function Polynomials	48
Table 11. Expanded Applesoft Sine Function Polynomials	48
Table 12. Applesoft Arctangent Function Polynomials.....	50
Table A.1. Page-Zero Definitions	86
Table B.1. Modified Applesoft Statements.....	89
Table C.1. Applesoft and Modified Applesoft Entry Points.....	99
Table D.1. Binary File Commands in DOS 4.5.08H	101

My Applesoft Journey™

Introduction to My Applesoft Journey™

I was graduated with my Bachelor of Science degree in Electrical Engineering in June, 1982. Having a 4.0 GPA provided me with many opportunities in securing an engineering position. Later that year, I decided to accept employment with Rockwell International in Downey, California. I lived close to Downey so I was fortunate to have a rather short commute. Of all the department managers who interviewed me, I selected to work under the manager of the Simulation Laboratory as an Initialization Engineer. Management encouraged all of the newly hired engineers to become familiar with Fortran 77 which had been recently released on the Nova computers by DEC. I soon began to realize my affinity for this computer language and my multi-player Black Jack Fortran program was a hit among my colleagues. Rockwell established a home computer purchase program for all employees the following year. After careful consideration, I selected the Apple][+ which included the AutoStart ROM. In short order I mastered assembly language for the 6502 microprocessor and Applesoft BASIC. By marrying assembly language routines with Applesoft programs using a tool I had developed which utilized that very technique, I advanced my knowledge of both languages far more quickly. In 1985 I accepted an offer from Ken Williams to work as an assembly language programmer at Sierra On-Line in Oakhurst, California.

I had become fascinated with numerical sort routines and very high speed graphic animation routines early in my computer language education, and Sierra On-Line was certainly the ideal environment to learn and to implement those and many other computer routines. One of my later assignments was to assist in the development of HomeWord Speller, the companion home productivity product to HomeWord which had already been released. My tasks included providing all of the diskette input and output routines and to develop the routines that would draw graphical icons on the HomeWord Speller initialization and configuration screens. I had already designed many of the diskette input and output routines for my hybrid Applesoft programs that utilized embedded assembly language routines. Developing the software to draw graphical icons was going to be a challenge. I turned to the graphical routines in Applesoft for help.

I developed a hybrid Applesoft program that could draw a collection of very simple High Resolution shapes like dots, vertical lines, horizontal lines, boxes, and parallel lines in order to create a complete and complex icon. Williams said that I can always assume that his customers owned Apple computers that contained ROMs that were installed with Applesoft Version 2; that it was safe to utilize any ROM routine. Unfortunately, I found that if I utilized the Applesoft HRPLOT routine at 0xF457 and the HLIN routine at 0xF53A, these routines do not correctly calculate the delta difference for the horizontal and the vertical start to end points for my particular use and requirements. And so began my lifelong Applesoft journey.

The BASIC language interpreter was certainly a wise choice for Apple Computer to purchase from the fledging Microsoft Corporation. The interpreter code for their BASIC was small enough to occupy about 0x2200 bytes of a 0x3000 byte Apple][ROM. This gave Apple engineers around 0x600 bytes in order to include Low Resolution and High Resolution graphics that was unique vis-à-vis the Apple][hardware design. The final 0x800 bytes was reserved for Steven Wozniak's brilliant ROM Monitor. The finished ROM that contains Microsoft's BASIC interpreter and Apple's graphic routines comprise Applesoft. Apple Computer has yet to publish the source code for Applesoft. Several publishers such as the Apple Orchard and Call A.P.P.L.E. have reprinted *Applesoft Internals* by John Crossley. The Sander-Cederlof

DocuMentor has also been used to provide, perhaps, the most complete source code documentation for Applesoft internals. Mr. Sander-Cederlof even includes his own personal comments within this documentation which identifies coding errors, routines that contain dangerous code under specific conditions, and routines that can utilize improvable code or replacement code. It was many, many years after I had already sourced the Enhanced Apple //e ROM Monitor and the Applesoft interpreter when I came across the S-C DocuMentor for Applesoft. Therefore, I have the benefit of both my own personal investigation into the Applesoft interpreter and the investigation of the Applesoft interpreter by Mr. Sander-Cederlof. I have taken all of the comments by Mr. Sander-Cederlof under advisement as I have made various modifications to my version of the Applesoft interpreter.

I like to think of assembly language mnemonics such as LDA and STA as instructions. And, I like to think of Applesoft tokens such as FOR and NEXT as statements. Applesoft token numbers range from 0x80 to 0xEA and these token numbers are always fixed to their assigned statement. If a new statement could be added to Applesoft, that statement would always be interpreted as 0xEB. If another new statement could be added, it would always be interpreted as 0xEC, and so forth. If an Applesoft statement is ever made unusable, its token number can never be retired.

Applesoft is heavily dependent on page-zero variables for many reasons, for example: Applesoft occupies a ROM which can only be read and never written; there is no dedicated CX page of bytes for Applesoft variables and pointers as there are for slot cards; there is no dedicated TEXT page of bytes for Applesoft variables and pointers as there are for slot cards; all Applesoft variables and arrays require page-zero pointers for their administration; all Applesoft TEXT manipulation routines require page-zero variables and pointers; all Applesoft floating-point routines require page-zero multi-byte registers, variables, and pointers; and, all Applesoft graphic routines require page-zero variables and pointers. Applesoft is heavily dependent on page-one, or the STACK, for many reasons, for example: to tokenize instructions entered on the Apple Command Line; to display a line of Applesoft instructions and statements; to display a hexadecimal floating-point number as a base 10 number; to save the parameters for a defined function as it is utilized; and, to implement recursion. For all of these and many more reasons, I have included all of the definitions for page-zero variables in Appendix A that are used in the ROM Monitor, in Applesoft, and in DOS 4.5.08H. Appendix B lists all of the Applesoft statements, their token number, and the location in Applesoft where that statement is processed. Appendix C contains all of the internal Applesoft entry locations for the modified version of Applesoft and whether these entry locations are the same or different in the unmodified version of Applesoft. The modified version of Applesoft is available for download.

Understanding the Deficiencies in Applesoft Mathematical Routines and Functions

Applesoft mathematical routines and functions that operate on very small floating-point numbers can become problematic. These routines and 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 32-bit significand is incremented. When addition, subtraction, or multiplication is initiated, only one operand uses a full 40-bit significand and the other operand uses a 32-bit significand. In division, only the quotient has any extra significance having two additional bits. Sticky bits are not utilized in Applesoft mathematical routines and functions in order to assist in making more intelligent numerical rounding decisions. Since the `cosine` and the `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 is limited to only what is acceptable given the total range of the floating-point numerical values for all Applesoft arguments.

Applesoft arithmetic also contains known irregularities that were purposefully implemented, some in which the user would not be expected to anticipate. These irregularities occur because certain decisions were made while designing the arithmetic algorithms. Other irregularities may also 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, for example, 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 and binary to decimal conversions where an identity might be expected but cannot be obtained. Unless a Taylor series is utilized that has at least thirteen to fifteen iterations, the Applesoft `sine` function exhibits extremely poor accuracy for arguments that are near zero. And, the Applesoft `sine` function generates 0 for all arguments that are greater than $0.5 * 10^{10}$. Apparently, the flaw in the Applesoft `sine` function for an argument that is very large in value is due to the `sine` argument reduction algorithm. And, as previously mentioned, the `cosine` and the `tangent` trigonometric functions are equally flawed since they are obtained by means of trigonometric identities that are solely based on the Applesoft `sine` function. Therefore, it is vital that the engineer or the mathematician is aware of all of the numerical limitations of the algorithms that are implemented in Applesoft and how each function can affect the accuracy of Applesoft arithmetic. And, the engineer or the mathematician must accommodate all of their complex floating-point variables, arrays, determinants, and inverse arrays for these Applesoft arithmetic irregularities. The modified Applesoft eliminates most of these irregularities.

Applesoft Variables

Applesoft utilizes two areas of memory for numerical and character string variables that include the Simple Variables and the Array Variables, or Simple/Array Variables, or SAVs for short. Figure 1 shows an example Applesoft program that resides in memory beginning at memory address 0x0801. In that figure, Free Space exists because the Applesoft Program, its SAVs, and its Character String Pool do not exceed the value that is stored in HIMEM minus 0x0801, the memory address where the Applesoft program and all other regular Applesoft programs traditionally load into and reside in memory. Applesoft also utilizes a large number of byte-pair memory locations in page-zero for its use. Many of these memory locations are to store addresses in low/high byte order that can easily be used as pointers in memory management routines. Even though DOS 4.5.08H is capable of loading an Applesoft program into any selected memory location, DOS usually loads an Applesoft program into memory at address 0x0801, which is the value that is found in PRGTAB. Using the size of the Applesoft program, DOS calculates the end address of the Applesoft

program and saves that information in PRGEND. Initially, DOS sets VARTAB to PRGEND and Applesoft sets ARYTAB and STREND to PRGEND and FRETOP to HIMEM.

Page-Zero Pointer Addresses	Applesoft Program
	0x0000
PRGTAB – 0x67:0x68	0x0801
	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

Figure 1. Example Applesoft Program Layout in Memory

When the Applesoft program begins to process its instructions, the program begins to create simple variables that include floating-point variables, integer variables, and character string variables. These variables reside in the Simple Variables area of memory as simple descriptors whose memory address is found in VARTAB. The definition of the descriptors for the variables that comprise the Simple Variables is shown in Table 1. As more and more Simple Variable descriptors are added by Applesoft, the Array Variables area is pushed higher and higher **up** in memory that reduces the size of Free Space. Simple Variable descriptors are always seven bytes in size, and depending upon the variable type, some of the descriptor bytes may not even be used. Table 1 shows that floating-point numbers require all seven bytes

for the variable name, the exponent, and its 4-byte mantissa. Integer numbers require only four bytes for the variable name and its value in **high/low** byte order, leaving the remaining three bytes initialized to 0. Finally, simple character string variables require five bytes for the variable name, the 8-bit length of the character string in bytes, and the memory address in **low/high** byte order where the character string resides in memory, leaving the remaining two bytes initialized to 0. Obviously, a simple character string variable cannot contain more than 255 ASCII characters since the number of characters in the simple character string variable is limited to a single 8-bit quantity. Applesoft programs should never define a character string variable to contain more than 255 ASCII characters.

Variable Type	Byte Definitions						
	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Floating-point 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
Floating-point 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

The definition of the descriptors for Applesoft Array Variables is shown in Table 2. As shown in Figure 1, the start address of the Array Variables area of memory is found in ARYTAB and the end address of the Array Variables is found in STREND. This area of memory contains single and multi-dimensioned Array Variable descriptors for arrays of floating-point numbers, arrays of integer numbers, and arrays of character string variables. Table 2 shows an example variable descriptor 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 they 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. Applesoft limits the number of dimensions for an array to eighty-eight.

Element Type	Byte Definitions				
	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
Floating-point 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 this Array Variable descriptor is located. The Array Elements that belong to an Array Variable descriptor begin immediately after the descriptor whose descriptor size can easily 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 that are shown in Table 1 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. The Array Element for arrays of floating-point numbers is five bytes in size and it contains the exponent of the floating-point number and its 4-byte mantissa. The Array Element for arrays of integer numbers is two bytes in size and it contains its integer value in **high/low** byte order. The Array Element for arrays of character string variables is three bytes in size and it contains the 8-bit length of the character string in bytes and the memory address in **low/high** byte order where the character string resides in memory. It should be apparent that all of the character string elements of a character string array do not necessarily have to contain the same number of characters, but any single character string element cannot contain more than 255 characters since its string length variable is limited to a single 8-bit quantity. Applesoft programs should never define a character string element to contain more than 255 ASCII characters.

Quite often an Applesoft program contains the text of some character string variable. As long as there is **no** text operation on that character string variable such as $A\$ = A\$ + B\$,$ for example, the string pointer address that is found in the Simple Variable or in the Array Element descriptor points to the actual character string data that is within the memory contents of the Applesoft program. In order for this character string variable or array element variable to be available, for example, to a Chained program, the actual character string data must be relocated into the Character String Pool. A simple way to force this character string relocation is to perform some menial data operation on that character string variable or that array element variable, such as $A\$ = A\$ + ""$ or $A\$(0) = A\$(0) + ""$. This simple operation does nothing to the character

string A\$ or to A\$(0) except to cause the actual data of A\$ or A\$(0) to be copied from within the contents of the Applesoft program into the contents of the Character String Pool.

The Character String Pool that is used in Applesoft to hold character string variables can create many side effects after character string variables have been processed in multiple ways. Bit and pieces of old character string data that have no descriptor as a result of this processing can clutter the Character String Pool. This additional string data clutter reduces the size of the Free Space and this can have a direct effect on the processing speed of many Applesoft string operations. When Free Space reaches a critical limit in size, Applesoft automatically calls the GARBAG routine that attempts to clear out all of the character string data that have no string descriptor. Many Garbage Collection algorithms have been previously published that accomplish the same results as GARBAG in far less time, but there can be a number of caveats when using some of these algorithms. For instance, normal Applesoft programs save all character string data in **lower** ASCII where the MSB is clear for each character byte in the string. And, normal Applesoft programs never allow more than one character string descriptor to point to the same character string data in memory. Multiple character string variable and array element descriptors may each point to identical character string data sets, but these identical sets of character string data must reside at different memory locations. Some Garbage Collection algorithms depend upon these constraints. If either constraint is not found to be true, a catastrophe will happen during the course of subsequent Applesoft processing! Of course, if the character string data of an Applesoft program is kept normal and these constraints are observed, there will be no subsequent processing problems. If assembly language routines, possible appendages to the Applesoft program itself, or other code segments perform exotic manipulations to the character string descriptors or to the contents of the Character String Pool, these constraints might very well be violated.

Applesoft Floating-Point Variables

Applesoft conducts all of its numerical processing, even for obvious integers such as those that are used in FOR/NEXT loops, using only floating-point variables. Signed 16-bit integer variables and arrays are provided in Applesoft, and an integer variable may be utilized in most Applesoft statements. 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 in order 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 0. 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 fractional part FFFFFFFF of decimal number D from its exponent TT. The fractional part FFFFFFFF of decimal number D contains eight numerical digits at most. Applesoft does not identify IMAGINARY floating-point numbers differently from REAL floating-point numbers. And, Applesoft does not define or provide any resources for either double precision integer numbers or double precision floating-

point numbers whether they are REAL or IMAGINARY. The modified Applesoft always prefaces a small floating-point number that is in decimal form with a 0 unless scientific notation is used otherwise.

Integer numbers as large as 1,048,576 or 2^{20} can be precisely expressed as an Applesoft floating-point number. The Applesoft integer to floating-point and the floating-point to integer conversion routines are designed to conduct these particular conversions without residual error. In fact, it is very straightforward to convert an integer number into an Applesoft floating-point number. For example, take the decimal number 937 and convert that number to hexadecimal, or 0x3A9, and then to binary, or %001110101001. Express that binary number into a mantissa of four bytes, or %0000000000000000000000001110101001. Count the number of zero-bits until the first one-bit is reached, or 22, which is 0x16. Since the maximum value for the floating-point exponent for 2^{20} is 0xA0, subtract the zero-bit count from 0xA0, or 0xA0 - 0x16 = 0x8A. Once the exponent is calculated, those first 22 zero-bits can be removed. That first one-bit that was encountered when counting the number of zero-bits is called the implicit high-order one bit and the sign bit is substituted for that implicit high-order one bit. In this example, the sign bit is positive, so that first mantissa bit becomes a zero-bit. The resulting floating-point value becomes 0x8A6A400000. It is easier to see the original integer, or %001110101001, when the entire floating-point number is expressed as a complete binary number that contains a space character between each byte:

```
%10001010 01101010 01000000 00000000 00000000
```

The reverse conversion of this floating-point number extracts the first bit of the mantissa as the sign bit and replaces this bit as the implicit high-order one bit. The exponent is subtracted from 0xA0 and the mantissa is shifted to right that many bits as shown:

```
%10011010 00000000 00000000 00000011 10101001
```

The third and fourth bytes of the mantissa contain the integer value for the original decimal number 937.

Management of Applesoft Floating-Point Registers

Applesoft utilizes a number of floating-point registers in order to assist the various floating-point routines that comprise all of the mathematical functions that are available in Applesoft. The primary Applesoft floating-point register is FAC and the secondary floating-point register is ARG. Both of these floating-point registers consist of five bytes where the first byte is used for its exponent and the next four bytes are used for its mantissa. Both FAC and ARG each utilize an 8-bit guard byte. The guard byte for FAC is FACGUARD and the guard byte for ARG is ARGGUARD. Both of these guard bytes are utilized in all four primary mathematical functions, that is, in addition, in subtraction, in multiplication, and in division. FACGUARD is utilized to hold the final guard byte value once the mathematical function is complete. The Applesoft addition function uses FAC and ARG for the addends and the function puts their sum into FAC. The Applesoft subtraction function uses ARG for the minuend and FAC for the subtrahend and the function puts their difference into FAC. The Applesoft multiplication function uses ARG for the multiplicand and FAC for the multiplier and the function puts their product into MULMANT which is a four-byte register that holds only the floating-point mantissa. The Applesoft division function uses ARG for the dividend and FAC for the divisor and the function puts the quotient into FAC. The exponents for both addition and subtraction are handled by a common routine that normalizes their exponents to equality. The exponents for multiplication and for division are handled by a different common routine that adds or subtracts the exponents.

Polynomial processing utilizes TEMP1 and TEMP2 which are also five-byte floating-point registers. The Applesoft SINE, COSINE, ARCTANGENT, and exponential function all use polynomial processing. The Applesoft TANGENT function utilizes TEMP3 in addition to the other two temporary floating-point registers. TEMP3 is a six-byte floating-point register and its fifth mantissa byte is utilized for its 8-bit guard byte, T3GUARD. The Applesoft square root function, or SQR requires TEMP1 and TEMP3. Polynomial processing had limited usage for guard bytes since TEMP1 and TEMP2 do not have associated guard bytes. After careful review of TEMP1 and TEMP2 utilization, I found that TEMP1 is used to hold the rounded user input value for SQR, to obtain the next 5-byte polynomial value in polynomial processing, or to hold the rounded user input range value for RND. TEMP2 is used to hold the initial X or X^2 term during active polynomial processing. My review shows that mathematical accuracy would not benefit if TEMP1 utilized a guard byte. However, mathematical accuracy in polynomial processing would definitely benefit if TEMP2 utilized a guard byte. Since TEMP2 is utilized only in this single intensely mathematical processing function, I found that it was possible to incorporate an 8-bit T2GUARD byte when the FAC register is copied to the TEMP2 register and when the TEMP2 register is copied to the ARG register. The T2GUARD byte becomes a critical component in maintaining the mathematical accuracy in polynomial processing in the modified Applesoft.

Memory	Name	Description
0xDE23	FRMSTAK3	Push FAC and FACGUARD onto the STACK; jump to (INDEX)
0xDE40	NOTMATH4	Pull ARG, ARGGUARD, ARGSIGN from STACK; set FACSIGN and XORSIGN; load FACEXP
0xE3AF	FNCDATA	Pull five numerical bytes from STACK into (FUNCNAM) indexed by Y-register
0xE9E3	LOADARG	Copy five bytes from (INDEX) into ARG; set ARGSIGN, XORSIGN, and ARGGUARD
0xEAE6	COPYM2F	Copy four bytes from MULMANT into FACMANT; normalize the exponent
0xEAF9	LOADFAC	Copy five bytes from (INDEX) into FAC; set FACSIGN and FACGUARD
0xEB1E	COPYF2T2	Copy (TEMP2) into INDEX; copy FAC into (INDEX); set FACSIGN; leave FACGUARD alone
0xEB21	COPYF2T1	Copy (TEMP1) into INDEX; copy FAC into (INDEX); set FACSIGN; leave FACGUARD alone
0xEB27	COPYF2FR	Copy (FORPNT) into INDEX; copy FAC into (INDEX); set FACSIGN; leave FACGUARD alone
0xEB2B	COPYFAC	Call RNDUP; copy FAC into (INDEX); set FACSIGN; leave FACGUARD alone
0xEB53	COPYA2F	Copy ARGSIGN to FACSIGN; copy ARG to FAC using indexed loop; set FACGUARD
0xEB63	COPYF2A	Copy FACSIGN to ARGSIGN; copy FAC to ARG; set ARGGUARD
0xF1BA	COPYF2T3	Copy (TEMP3) into INDEX; call COPYFAC2, avoid RNDUP; copy FACGUARD to T3GUARD
0xF695	COPYT32A	Copy (TEMP3) into INDEX; call LOADARG; copy T3GUARD to ARGGUARD

Table 4. Routines That Copy Floating-Point Registers or Numbers

The first three floating-point data transfer routines that are shown in Table 4 are used to push FAC onto the STACK, pull ARG from the STACK, or pull numerical data from the STACK and save that data to a specific memory location as a floating-point number. These routines push or pull numerical data onto or from the STACK byte by byte in order to affect the fastest data transfer rate at the expense of Applesoft space. The FRMSTAK3 routine pushes the entire contents of the FAC register onto the STACK after RNDUP is called. I modified this routine to push FACGUARD onto the STACK rather than call RNDUP. Its complement routine NOTMATH4 transfers floating-point data that is on the STACK into the ARG register, and I also modified this routine to pull ARGGUARD before it pulls ARGSIGN from the STACK. FNCDATA is a routine that pulls a floating-point number from the STACK and copies that number to memory whose address resides in FUNCNAM. And, in addition to those three routines, there are eleven routines that copy the contents of one

of the five floating-point registers to memory, or memory to one of the registers, or one register to another register. Ten of these data transfer routines copy numerical data to or from a floating-point register byte by byte in order to affect the fastest data transfer rate at the expense of Applesoft space. The COPYAZF data transfer routine, however, favors Applesoft space at the expense of numerical data transfer rate and this routine is only utilized by POWER. Actually, ADD uses COPYAZF in order to return the value that is in the ARG register when the FAC register is 0. I did unwind the COPYFZA data transfer routine that utilized an indexed register loop because this routine is used by many functions. However, it was not necessary to set the data transfer loop indexing register to its terminating value. Table 4 presents all of these floating-point copy routines, their location in Applesoft, their names, and a brief description of their function in the modified Applesoft.

Page	Topic	Description
0xC0	I/O	Memory, video, and slot card management soft switches.
0xC1-0xC2	Monitor Support	ROM Monitor input and 40/80-column output support routines.
0xC3	Video Output	Claims 0xC8:CF space; cannot be used by 0xF8:FF routines.
0xC4	Interrupt Handler	Apple //e configuration is captured; the interrupt handled; system is restored.
0xC5	STEP and TRACE	Mini-assembler routines.
0xC6-0xC7	GARBAG; SWEET16	Several garbage collection routines and <i>SWEET16</i> Metaprocessor.
0xC8-0xCE	40/80 column handlers	Routines to display 40 and 80 columns.
0xCF	STEP and TRACE	Mini-assembler routines.
0xD0-0xD3	Addresses and Names	Applesoft statement addresses, names, and error messages.
0xD4-0xD6	Interpreter	Applesoft interpreter, restart, parser, tokenizer, memory management
0xD7-0xD8	Routines	FOR, TRACE, RESTORE, STOP, END, CONT, LOAD, RUN routines
0xD9-0xDA	Routines	RUN, GOSUB, GOTO, RETURN, POP, DATA, REM, LET, PRINT routines
0xDB-0xDF	Routines	GET, INPUT, READ, NEXT, PDL, DIM routines
0xE0-0xE6	Routines	POS, DEF, STR\$, GARBAG, CHR\$, LEFT\$, RIGHT\$, MID\$, LEN, ASC routines
0xE7-0xEB	Routines	VAL, PEEK, POKE, WAIT, SUB, ADD, LN, MULT, DIV, SGN routines
0xEC-0xEF	Routines	ABS, INT, FPOUT, SQR, POWER, EXP, LOG, PI, RND, COS, SIN routines
0xF0-0xF1	Routines	TAN, ATAN, CHRGET, COLDSTRT , CALL, IN, PR routines
0xF2	Routines	PLOT, HLIN, VLIN, COLOR, VTAB, SPEED, TRACE, NOTRACE routines
0xF3	Routines	INVERSE, FLASH, HIMEM, LOMEM, ONERR, RESUME, DEL, GR routines
0xF4	Routines	TEXT, READ, HGR2, HGR, POSN, HRPLOT routines
0xF5-0xF6	Routines	HLIN, DRAW, XDRAW routines
0xF7	Routines	HCOLOR, HPLLOT, ROT, SCALE, TITLE, 40/80 column patches, HTAB routine
0xF8-0xFF	ROM Monitor	Modified ROM Monitor that supports 40/80 column display routines.

Table 5. General Layout of the C0:FF ROM

Introduction to Applesoft Source Code

The general layout of the Apple //e ROM, that is, the CXROM addition, the Applesoft interpreter, and the ROM Monitor is shown in Table 5. The COLDSTRT routine that is shown in boldface in Table 5 appears to complete Version 1.1 that Apple Computer purchased from Microsoft as Applesoft I, the 6502 BASIC

interpreter. The Applesoft statements that follow the COLDSTRT routine begin in the 0xF2 page and they handle the unique LORES and HIRES graphic commands which were provided by Randy Wigginton and Cliff Huston. The Applesoft interpreter that was provided in the Enhanced Apple //e uses the last half of the 0xF7 page for patches that support 40-column and 80-column displays and to provide a correctly functioning Applesoft HTAB statement. I have not changed the entry location address for any of the Applesoft statements. What I have changed are the routines that are used by these Applesoft statements.

I approached this journey through Applesoft as I have done countless numbers of times when I have explored, learned, and modified the software that has been written by other programmers or other software engineers. I am a professional software engineer because I have made my living at developing software products for several aerospace companies. And, I have written all of the software for those software products in various assembly languages, Fortran, or in C language. I did restrict myself to only utilizing ANSI C language because most of my software products were required to process data in real time. Some of the computer platforms that I have utilized for my software products include the BBN Butterfly, SEL Encore computers, SUN Microsystems workstations, and the SGI Origin 2000 and 3000 series of mainframes. Several of my software products as well as an Origin 2000 were installed on the Raytheon Multi-Program Testbed, or, the RMT which is a Boeing 727 that is used to fly on various sorties along with various government sponsors. I consider myself more than well equipped to understand the 6502 assembly language that was used to write the Applesoft interpreter. I am more than able to discern errors in the use of the 6502 assembly language and in algorithm logic, and I can discern illogical software structure, order, and format. Furthermore, I am fully capable of explaining the function of an Applesoft software algorithm.

I begin my journey through Applesoft with several goals in mind. First and foremost, I want to ensure that each Applesoft mathematical function can produce the most reliable solution having at least ten digits of accuracy even though only nine digits can be presented at any given time. In order to accomplish this goal, I place a strong emphasis on utilizing guard bytes in every possible calculation and in every series of calculations. I want to remove the rounding of a floating-point variable at all times during consecutive calculations until that variable must be rounded before it is written to memory and presented to the user. I want to add additional statements to the Applesoft language repertoire in order to increase the precision of the Applesoft language. And, above all, I want to repair or to replace the ill-informed as well as the uninformed software decisions that I found laced throughout the Applesoft language that contributes to the generation of mathematical errors. Of course, the C0:FF ROM contains only so much real estate for changes, for additions, and for improvements to its software routines. When any duplicated software logic is uncovered, the possibilities to insert software changes, additions, and improvements into the Applesoft language become more probable. That is when I become driven by excitement and eagerness to install new modifications in order to deliver an Applesoft language that has far more accuracy. Without having any explanations for the fabricated, concocted, and aberrant polynomials that are used for the processing of Applesoft transcendental functions, these functions are particularly problematic even when many other mathematical modifications are utilized. Perhaps better solutions can be found by others after having read and studied the details of my journey through Applesoft.

The Applesoft Statements

The Applesoft language is designed around 107 commands which I designate as Applesoft statements. These statements can be categorized into three main groups. The first category of Applesoft statements are those that perform a particular function, like END or FOR or HCOLOR= or HIMEM:. These statements do not

require evaluating an expression that is enclosed by parenthesis. This first category of statements is called the BASIC statements and I preface a B in front of each of their software labels. There are sixty-four BASIC statements. The list of sixty-four two-byte fully qualified addresses for the software labels of the BASIC statements begins the Applesoft source code at 0xD000. The token numbers or identification numbers for all Applesoft statements begin with the number 0x80, that is, with numbers that have their most significant bit set. All other ASCII data in the form of numbers and strings in an Applesoft program have their most significant bit clear. In order to calculate the token number that is associated with each BASIC statement, Applesoft takes the least significant byte of the BASIC address where the two-byte label address is found where that statement is processed in memory, divides that BASIC address byte by two, and then adds 0x80. For example, the Applesoft BASIC statement POKE is processed at 0xE77A. That address, 0xE77A, is found in the list of two-byte BASIC addresses at 0xD072. Thus, the token number for POKE is $0x72 / 2 + 0x80 = 0xB9$ or 185. The token numbers for the BASIC statements begin with 0x80 and they end with 0xBF.

The next category of statements is called the FUNCTION1 statements and I preface an F in front of each of their software labels. There are twenty-two FUNCTION1 statements. The FUNCTION1 statements are those Applesoft statements which contain an expression that has a numerical value and that expression is enclosed by parenthesis. That numerical expression must be evaluated before Applesoft can process the statement. The list of twenty-two two-byte fully qualified addresses for the software labels of the FUNCTION1 statements begins at 0xD080 which follows the two-byte addresses for the BASIC statements. Examples of FUNCTION1 statements are SGN(), PDL(), and ASC(). In order to calculate the token number that is associated with each FUNCTION1 statement, Applesoft takes the least significant byte of the FUNCTION1 address where the two-byte label address is found where the statement is processed in memory, divides that address byte by two, and then adds 0x92. For example, the software label address for the Applesoft FUNCTION1 statement PEEK is 0xE764. That address, 0xE764, is found in the list of two-byte addresses at 0xD0A0 and the token number for PEEK is $0xA0 / 2 + 0x92 = 0xE2$ or 226. The token numbers for the FUNCTION1 statements begin with 0xD2 and they end with 0xE7.

The last three statements are the FUNCTION2 statements and they include the LEFT\$, RIGHT\$, and MID\$ statements. These statements also contain an expression that is enclosed by parenthesis which must be evaluated before Applesoft can process the statement. However, the expression for these statements are more complex than the FUNCTION1 statements because these expressions contain two or three string variables rather than a single numerical variable as found in the FUNCTION1 statements. The token number for FUNCTION2 statements is calculated in the same way as token number is calculated for FUNCTION1 statements. The token numbers for the FUNCTION2 statements begin with 0xE8 and they end with 0xEA. Applesoft token parsing happens to end with the FUNCTION2 statements. The modified Applesoft, however, contains two additional statements that follow the FUNCTION2 statements and these two statements are processed like FUNCTION1 statements. These two statements include the LN statement and the PI statement.

The Applesoft statement SCRN(is syntactically not a FUNCTION1 statement because it contains two numerical expressions that are separated by a comma rather than a single numerical expression. And, the SCRN(statement is syntactically not a FUNCTION2 statement because its two expressions contain numerical variables rather than string variables. The UNARY function at 0xDF0C processes all FUNCTION1 and FUNCTION2 Applesoft statements, but UNARY extracts the SCRN(statement first with no further processing in the unmodified Applesoft.

The last category of statements is called the Operator TAG statements and I preface an O in front of each of their software labels. There are ten TAG statements. The TAG statements are those statements which perform a mathematical operation or some sort of mathematical comparison. The list of their precedence codes and their two-byte fully qualified addresses for the software labels of the TAG statements begins at

0xD0B6 in the modified Applesoft which follows the two-byte addresses for the FUNCTION statements. Examples of TAG statements are +, AND, and <. The precedence code is the value that Applesoft uses in order to determine the processing order for various variables and TAGS when Applesoft evaluates a complex mathematical expression. The precedence codes range from 0x46 for OR to 0x7F for =. Applesoft extracts the precedence code first and then it extracts the address bytes for the selected routine that processes that TAG statement. The token numbers for the TAG statements begin with 0xC8 and they end with 0xD1.

The remaining eight Applesoft statements use the token numbers from 0xC0 to 0xC7. These statements appear to be a group of catch-all statements that include FUNCTION-like statements and statements that are ancillary to other Applesoft statements. For example, the statements TABC and SPCC look very much like FUNCTION1 statements, FN can only be used after it has been defined by DEF, and TO, THEN, AT, NOT, and STEP must be used ancillary to other Applesoft statements. These Applesoft statements are not listed with a two-byte fully qualified address because these statements are parsed while the Applesoft interpreter is processing other Applesoft statements. These Applesoft statements are usually identified when and if Applesoft would logically find their occurrence, utilize their occurrence, or require their occurrence.

I have previously stated that I have added two additional statements to the Applesoft language repertoire which includes the PI statement with token number 0xEB and the LN statement with token number 0xEC. The first statement that I added to Applesoft is the PI statement, and this statement simply loads the FAC floating-point register as well as its guard byte FACGUARD with the 48-bit value of π . The second statement that I added to Applesoft is the LN statement. The unmodified Applesoft calculates the natural logarithm for the Applesoft LOG statement which is somewhat of a misnomer. In engineering, especially in Electrical Engineering, the logarithm of a number is the exponent to which a base must be raised in order to produce that number. Therefore, in order to differentiate the natural logarithm which is based on e and the logarithm which is based on 10, LN is used for the natural logarithm and LOG is used for the base-10 logarithm. In the modified Applesoft, the LN statement produces the same values that the LOG statement produces for the same arguments in the unmodified Applesoft. In the modified Applesoft, the LOG statement multiplies what the LN statement produces by the floating-point variable base-10 LOG (e). I added both of the two-byte fully qualified label addresses for the new PI and LN Applesoft statements at 0xD0B2 and 0xD0B4, respectively, which comes after the address for MID\$ and before the Operator TAG addresses. Adding these two addresses changes the location for BASNAME by four bytes which is where all of the Applesoft statement names are listed in DCI format.

All Applesoft statements are reserved words for the Applesoft interpreter. For example, if one should define a variable such as GR = 32, the Applesoft interpreter will first initiate LORES graphics and then issue a Syntax Error in the offending line where the GR statement is incorrectly used for the name of a variable. I find it interesting that the Applesoft language developers thought that it was necessary to add particular suffixes to some Applesoft statements, suffixes like =, :, and C. I can understand, perhaps, why \$ was added to the string processing statements in order to identify their intention and ultimate purpose. I suppose not having to evaluate the = TAG followed by a parameter does reduce some little processing, so why not use = rather than : for HIMEM: and LOMEM:? The CHKOPNP routine at 0xDEBB that checks for an open parenthesis is already in place, so why add the open parenthesis to TABC, SPCC, and SCRN C and not to all of the FUNCTION statements or none of the FUNCTION statements? I imagine that each of the many Applesoft language developers had their own reasoning and their own rationale for the ownership of the various routines that processed statements that were under their direction, and even control over the statement name itself. There might have even existed a statement playbook that went so far as to already spell out the characters that are used for each Applesoft statement. Certainly, how much thought was given to how these naming peculiarities might affect the overall development of the Applesoft interpreter? I do find it interesting when I conjure up possible explanations for some of these very awkward statement names.

Disabling an Applesoft statement does not remove the address requirement for some sort of alternate processing that must always be assigned to that token number. Applesoft programs that were previously developed have already been tokenized using an unmodified Applesoft, that is, the Applesoft statements have already been converted to their assigned token numbers. These programs are intended to execute in their tokenized form at any time and on any Apple][computer. Thus, disabling an Applesoft token number that is assigned to an Applesoft statement consists of two tasks: substituting another name for the Applesoft statement in BASNAME in order to prevent the use of that token number and substituting a different software routine for that token number if it should be parsed in an Applesoft program. Clearly, listing an Applesoft program using a modified Applesoft that has disabled certain token numbers would not produce a meaningful version of that program. And, when executed, that program would certainly not produce many of the intended results. It is simply not possible to incorporate new functionality into a space that has a fixed size without discarding something that is less desirable. I have chosen to discard all of the cassette recorder Applesoft write statements and some of the cassette recorder Applesoft read statements. The list of Applesoft statements that I have discarded include SHLOAD, RECALL, STORE, and SAVE, and their respective token numbers are 0x9A, 0xA7, 0xA8, and 0xB6. I have chosen to only retain the cassette recorder Applesoft read statements LOAD and RUN in order to support Insta-Disk and c2t processing. These two magnificent routines, Insta-Disk and c2t processing, were both designed and developed by Egan Ford. Refer to my book *DOS 4.5 Volume and File Disk Management System Second Edition* for a detailed discussion on Insta-Disk software, Insta-Disk disk images, and the c2t C language software that creates the Insta-Disk disk images. The SHLOAD statement has become a DOS 4.5.08H command and this command also includes several options. The DOS SHLOAD command provides the ability to load the data of a SHAPE table from a diskette into memory for the purpose of drawing HIRES shapes. I have also developed the companion SHSAVE command for DOS 4.5.08H, and this command provides the ability to save the data of a SHAPE table onto a diskette. See Appendix D for using the DOS SHSAVE and the DOS SHLOAD commands.

BASNAME begins at 0xD0D4 in the modified Applesoft. This is where the Applesoft interpreter begins its search for every ASCII string that is not enclosed within double quotes when the interpreter is tokenizing an input line of Applesoft. When an ASCII string is found within this table, that string is replaced by the number that resides in a statement counter before that line of Applesoft is inserted into the Applesoft program by virtue of its given line number. Certainly, in order to accomplish any of the goals set forth in this Applesoft journey, additional Applesoft space must be found or created. In concert with finding this space is the desire to disable specific Applesoft statements. The GR command happens to occur early in BASNAME and this command has only two characters. It is the perfect replacement statement for SHLOAD, then RECALL and STORE, and finally SAVE. While the Applesoft interpreter is scanning BASNAME, these four statements will never be found and tokenized and, therefore, they are disabled in the modified Applesoft. If an Applesoft program is found to contain any of the token numbers for these disabled statements, the GR statement will be processed using the software routine that is assigned to that token number. In the modified Applesoft, the address for IORTS or 0xFF58 is used to replace the Applesoft processing and label addresses for SHLOAD at 0xD034, RECALL at 0xD04E, STORE at 0xD050, and SAVE at 0xD06E. When the Applesoft interpreter encounters the token number for these Applesoft statements, Applesoft will simply process an RTS instruction and return to the Applesoft program to fetch the next Applesoft statement or issue a Syntax Error in <nn> if an expression is included with the GR statement.

BASNAME also includes the names of the two new Applesoft statements that I added in DCI format at 0xD256 and 0xD258, respectively. BASNAME now ends at 0xD25A with a terminating NULL byte. In the unmodified Applesoft, BASNAME ends at 0xD260. With these five extra bytes and changing “REDIM'D ARRAY” to “Redefined Array”, all contractions can be removed in the list of error messages that begins at 0xD25B with MSG01 and ends with MSG20 and still leave three extra bytes remaining. I have also changed the

error messages to include some lower case characters that assist in making the error messages far more readable in my opinion since the Apple //e can display lower case characters. I have never liked the ? prompt character that Applesoft uses when Applesoft is requesting input data or when Applesoft is printing an error message. This is my opportunity to change the prompt character to > in OUTPROMT at 0xDB56. I also modified MSG21 at 0xDCDF and MSG22 at 0xDCEF to print the > character rather than the ? character for those two error messages. These are the only three locations that need to be changed in order to utilize a less-offensive Applesoft prompt character. To further assist in making error messages more readable, I have modified locations PRLINUM at 0xD431 and PRTMSG19 at 0xED0A to print additional carriage returns.

The Applesoft Interpreter

The Applesoft interpreter is a collection of Applesoft statements, their routines, and other functions that manage an Applesoft program. The interpreter is used to construct an Applesoft program in memory, initialize that program, and execute Applesoft statements, ROM Monitor routines, and external assembly language routines. The various Applesoft statements that assist the interpreter to manage Applesoft processing and Applesoft program flow are part of the Applesoft interpreter. Applesoft is only generally divided into its collection of statements that assist the Applesoft interpreter, statements that manage string and numeric variables, statements that perform floating-point arithmetic operations, statements that perform transcendental arithmetic operations, Applesoft initialization and miscellaneous functions, and statements that manage the various LORES and HIRES graphic routines. The following is a collection of Applesoft statements and their routines that assist the processing and the capabilities of the Applesoft interpreter.

The Applesoft interpreter begins with the GTFORPNT routine three bytes earlier at 0xD352 than in the unmodified Applesoft. This routine is utilized by the FOR and NEXT Applesoft statements. Each FOR/NEXT construction is called a frame and GTFORPNT scans through the STACK for the frame whose address for its current iteration number matches the address that is currently in the FORPNT variable. If the addresses do not match, GTFORPNT resets its pointer to check the previous frame which is twenty bytes higher on the STACK. If the MSB in FORPNT is zero, then the address for the current iteration number is simply copied to FORPNT. This logic takes care of the two cases when NEXT specifies a variable name or not, respectively. In the unmodified Applesoft, when NEXT does not specify a variable name, the logic passes from the first half of GTFORPNT into the second half of GTFORPNT simply to set the processor Z status to TRUE. Of course, the code is more condensed when using this logic, but the processing time is needlessly lengthened for every single iteration of every single FOR/NEXT frame. Those three extra bytes up front to GTFORPNT are used to stop the logic of the first half of GTFORPNT needlessly passing into the logic of the second half of GTFORPNT, and it forces the processor to set the Z status to TRUE and immediately return to the caller.

The next routine is the BLTU routine or Block Transfer Utility and it begins at 0xD393 which is the same address for BLTU in the unmodified Applesoft. BLTU is designed to be a negative copy routine that copies data from the end of a program to some higher location in memory. This method of copying data is necessary when a line of Applesoft is inserted somewhere in the middle of a program. The end of the program needs to be copied higher in memory and backwards, that is, from a higher address to a lower address such that the copy process does not overwrite the program. As in all negative copy routines, even the routine I designed for the DOS 4.5 CHAIN command, the routine can appear unwieldy and cumbersome.

A FOR/NEXT frame consists of twenty bytes as previously mentioned, and Applesoft must always verify that the STACK has enough room to add another frame. This verification check is performed by CKSTKSIZ at

0xD3D6. A value, 0x0A for FOR, 0x03 for GOSUB, or 0x01 for an expression evaluation is doubled by CKSTKSIZ and added to 0x36, and that sum is compared to the current stack pointer. The ASL instruction that is used to double the entry value to CKSTKSIZ also serves the need to ensure that the C-flag is clear before the addition is performed. It seems that 0x36 is a rather generous value for STACK headroom and some may consider reducing this value in order to allow a deeper nesting of FOR/NEXT loops or nested GOSUBs. Another verification check routine follows CKSTKSIZ at 0xD3E3 that ensures that the end of the Applesoft array variable descriptors given by STREND do not overflow into the beginning of the Character String Pool given by FRETOP. If the CKSTRSIZ routine detects this situation, it protects and copies FRETOP and the page-zero locations 0x94:9C to the STACK, calls GARBAG, and then restores FRETOP and the 0x94:9C page-zero locations. If GARBAG is unsuccessful in separating STREND and FRETOP sufficiently, the Out of Memory error is posted as it is for CKSTKSIZ when the STACK has insufficient memory.

All error messages in Applesoft are printed by the PRTERr routine at 0xD412. Applesoft is aware of only two modes of operation: Direct mode and Running mode. Obviously, if an Applesoft program is processing Applesoft statements, then Running mode is in operation and this mode can easily be detected by looking at the MSB of the variable CURLIN+1, or the current line number that is being processed. Otherwise, Applesoft ensures that CURLIN+1 is always initialized to 0xFF when Applesoft is not processing Applesoft statements and Direct mode is in operation. Another flag that is maintained by Applesoft is ERRFLG. DOS 4.5.08H knows about this flag as the ASONERR flag. When an Applesoft program uses the Applesoft ONERR statement, Applesoft sets the MSB in ERRFLG and performs other housekeeping chores. If the MSB in ERRFLG is set, then PRTERr transfers the error management to HANDLERr at 0xF2E9. Otherwise, PRTERr happily prints the appointed error message. If CURLIN+1 is not equal to 0xFF, then PRTERr uses PRTMSG19 at 0xED0A to print the line number in which the error occurred because Applesoft is in Running mode. PRTERr concludes by printing an additional carriage return. I have modified the PRTERr routine along with developing a new routine called PRTMSG19 which resides just before LINEPRT at 0xED18. The purpose of these modifications is to clearly show the Applesoft error message by using carriage returns before and after the error message. PRTERr falls directly into the Applesoft RESTART routine.

DOS 4.5.08H initializes its WARMADR pointer to the address for ASROMWRM which is also the RESTART routine in Applesoft at 0xD43C. This is the routine where the prompt character] is printed to the screen and the page-zero pointers that are shown in Figure 1 are initialized. Applesoft programs are also developed using this routine along with the resources of BLTU. Immediately following RESTART is the ASENTER routine at 0xD4F2 which is the entry location for the real and true Applesoft Interpreter. DOS 4.5.08H initializes its RESETADR pointer to the address for ASROMRST which is also the address for ASENTER. ASENTER clears all variables and recalculates all of the page-zero pointer addresses that are shown in Figure 1.

The INLIN routine at 0xD52C removes the PROMPT character, reads the Applesoft command line, stores the input characters into INPUT, and clears the MSB of all entered data. INLIN also terminates data input after 0xEF or 239 characters have been entered. I was able to easily accelerate this routine and reduce it by two bytes. I also removed the next routine, the INCHR routine which is completely unnecessary. INCHR was only used by the ISCNTLC routine at 0xD858. Eight free bytes are now available at 0xD551. The PARSINPT routine at 0xD559 parses and tokenizes the data in INPUT for RESTART. This is a very lengthy and complex routine, and PARSINPT shows much of the brilliance of the Applesoft developers. PARSINPT initializes a pointer with the base address of BASNAME and I was surprised that TOKNCNTR was initialized with 0x00 at 0xD59A and not with 0x80 which is the first Applesoft token number. Several ambiguities in the Applesoft lexicon are worked out at 0xD5B8 using tedious character comparisons when the token number is initially thought to be 0xC5 for the Applesoft AT statement. Parsing continues until End of Line, a NULL byte, or End of Statement, that is, a : byte, is found.

A line of Applesoft begins with a line number followed by an Applesoft statement which may, for example, be followed by one or more Applesoft statements, variables, expressions, equations, and ASCII text that is between quotation marks. Every line of Applesoft is precisely constructed using an exact format and syntax. That syntax is used to build every line of Applesoft using two bytes for the fully qualified address in low/high byte order for the next line of Applesoft, two bytes for the line number in low/high byte order, up to 239 bytes for the Applesoft, and a terminating NULL byte. This format is crucial to the FNDLIN routine at 0xD61A when it searches an Applesoft program for a particular line number. The routines RESTART, LIST, ASROMSET, and DEL use LINNUM when they call FNDLIN in order to search an Applesoft program for that specific line number. FNDLIN utilizes this syntax for every line of Applesoft it processes and it can easily skip through an Applesoft program while searching for a specific line number. Because the syntax of a line of Applesoft does not include the two-byte address for the previous line of Applesoft, Applesoft can only search forward and never backward through an Applesoft program.

The routine for the first Applesoft statement to be processed by the Applesoft interpreter is for NEW at 0xD649. Many of the BASIC statements like NEW begin with checking the state of the Z-flag. If the Z-flag is clear, it indicates that some stray character is included with the NEW statement, then NEW simply does nothing, yet the Applesoft interpreter will issue a Syntax error for that stray character, whatever it is. NEW as well as the Applesoft COLDSTRT initialization routine at 0xF127 both utilize the SCRTCH routine at 0xD64B in order to initialize the page-zero pointers that are shown in Figure 1. Whether NEW is issued on the Apple Command Line or if NEW appears in an Applesoft program, NEW simply falls into SCRTCH. SCRTCH falls immediately into the SETPTRS routine or ASROMCLR at 0xD665 in order to finish page-zero pointer initialization. DOS 4.5.08H calls ASROMCLR after DOS has moved all of the Applesoft variable and array descriptor addresses to their new memory location on behalf of DOS CHAIN, and DOS calls ASROMCLR before DOS enters ASROMNEW processing at 0xD7D2 on behalf of DOS RUN. SETPTRS bypasses the Applesoft CLEAR statement entry and falls into the CLEARC routine at 0xD66C, and CLEARC falls into the STKINIT routine. The Applesoft CLEAR statement entry at 0xD66A, like the NEW statement entry, begins its processing by checking the state of the Z-flag, and if the Z-flag is clear, then CLEAR does nothing and it lets the Applesoft interpreter handle whatever stray characters it finds. Otherwise, CLEAR simply falls into CLEARC in order to initialize FRETOP, ARYTAB, and STREND.

I have already noted that Applesoft is heavily dependent on page-zero variables, yet page-zero is hardly enough memory for the processing demands of Applesoft. Thus, Applesoft is also heavily dependent on STACK memory and Applesoft routinely pushes floating point variables, addresses, pointers, and data onto the STACK and pulls those variables and addresses from the STACK for every iteration of a FOR/NEXT frame or for a defined function as two examples. In order to keep Applesoft from losing complete control of the STACK with possible overruns or overflows, STKINIT at 0xD683 is where Applesoft controls the start of the STACK and initializes its pointer to 0xF8. This pointer initialization, of course, plays havoc with DOS and DOS simply cannot call ASROMCLR and expect to find its return address still available on the STACK. Obviously, I have written into DOS 4.5.08H an adequate solution that presets the STACK pointer to 0xFA in order to circumvent this unfortunate situation specifically for the DOS CHAIN command. Once SETPTRS processing is complete, an Applesoft program is nearly ready to begin processing its Applesoft statements. The small routine that follows STKINIT is the STXTPTR routine at 0xD697 and STXTPTR is called by SETPTRS to simply copy the address that is in PRGTAB to TTXTPTR decremented.

The Applesoft LIST statement begins at 0xD6A5 and it is a lengthy and somewhat complex routine. The six assembly language commands that begin LIST processing allow for the use of the – and , delimiters to control some of the behavior of LIST. When Applesoft tokenizes an input line of Applesoft and before adding that tokenized line into an Applesoft program, the Applesoft interpreter removes all unnecessary space characters during that process in order to condense the total size of the Applesoft program. However,

when LIST displays those program lines of Applesoft to the screen or to any output device such as a printer, the Applesoft LIST statement inserts a variety number of space characters according to its own processing decisions. Some of those decisions can be controlled by adjusting WNDWDTH. If a CTRL-C character is typed while LIST is displaying data, LIST will be interrupted and the Break message will be printed to the screen. I modified LIST and removed the useless NOP command at 0xD708 and I replaced the printing of the 0x0D character at 0xD724 with a call to PRTCR at 0xDB50, a new routine which I added to OUTCHR at 0xDB58. LIST is an excellent example of a routine which embeds another routine within its processing. Whether the GETCHR routine is utilized by other routines or not does not condone this programming style in my opinion. This practice appears rampant throughout the Applesoft interpreter and I find it very disturbing. There is no reason whatsoever why GETCHR cannot be placed outside of LIST processing at 0xD758.

The Applesoft FOR statement processing begins at 0xD766 and this routine initializes the SUBFLG flag with the value of 0x80 in order to disable the use of array variables with the FOR statement. This routine pushes twenty bytes onto the STACK after calling CKSTKSIZ in order to verify that the STACK still has enough room for another FOR/NEXT frame. Those twenty bytes consist of two bytes for the TXTPTR of the next Applesoft statement (TXTPTR pushed first), two bytes for the current line number (CURLIN+1 pushed first), six bytes for the initial or the current value of the FOR variable as a floating-point number (FACUARD pushed first), one byte for the sign of the STEP value whether or not STEP is included, six bytes for the value of the STEP variable as a floating-point number whether or not STEP is included (FACUARD pushed first), two bytes from the FORPNT variable that is the address of the FOR variable that is stored within VARTAB (FORPNT+1 pushed first), and one byte for the token identification number of the FOR statement or 0x81 in order for GTFORPNT to easily identify this FOR/NEXT frame on the STACK. The unmodified Applesoft pushes eighteen bytes onto the STACK because FACGUARD is not included with the floating-point value of the FOR variable and for the floating-point value of the STEP variable. FOR processing utilizes the FRMSTAK3 routine at 0xDE23 in order to push the floating-point value of the FOR variable onto the STACK and to automatically enter STEP processing by means of an indirect jump. STEP processing utilizes the FRMSTAK2 routine at 0xDE15 in order to push the sign of the STEP value as well as the floating-point value of the STEP variable onto the STACK. The Applesoft STEP statement processing immediately follows the FOR statement processing at 0xD7AF. Both FOR processing and STEP processing along with the processing of FRMSTAK3 and FRMSTAK2 are absolutely beautiful implementations of two Applesoft statements and their ancillary routines. I am astounded in how clever all of these routines are designed. STEP processing falls into the NEWSTT routine.

Both the DOS RUN and the DOS CHAIN commands in DOS 4.5.08H enter the Applesoft ASROMNEW or NEWSTT routine at 0xD7D2. This routine is also used to process a FOR/NEXT frame, an input line of Applesoft in Direct mode after the interpreter has parsed and tokenize the Applesoft statements in that Applesoft line at 0xD569, LIST at 0xD726, TRACE at 0xD823, GOSUB at 0xD93B, NEXT at 0xDD49, HANDLERR at 0xF315, and RESUME at 0xF32B. NEWSTT falls into the DOTRACE routine at 0xD805 which prints a # followed by the line number of the Applesoft statement as that statement is processed when the MSB of the TRACEFLG flag is set. DOTRACE returns to the top of the NEWSTT routine in order to process the next Applesoft statement. Thus, the Applesoft interpreter loops using these two routines while checking for a CTRL-C in NEWSTT and processing each Applesoft statement using the DOSTAMT routine in DOTRACE.

The Applesoft DOSTAMT routine at 0xD828 follows DOTRACE. DOSTAMT only processes the sixty-four BASIC statements using the token number of the BASIC statement as an index to BASADDR for the address of the routine that processes that BASIC statement. The routine address is always found decremented and DOSTAMT pushes that address onto the STACK and jumps to the CHRGET routine. As soon as the CHRGET routine issues its RTS instruction, the routine for the designated statement will be entered immediately. The CHRGET routine looks ahead one character and it clears the C-flag if that character is a number, otherwise CHRGET sets the C-flag. What is more interesting is that CHRGOT sets the Z-flag if that character is a

colon :, that is, an End of Statement marker. The Applesoft RESTORE statement at 0xD849 follows DOSTAMT. RESTORE statement processing simply sets DATPTR to the beginning of the Applesoft program, that is, whatever address that is found in PRGTAB is used to initialize DATPTR. The ISCNTLC routine at 0xD858 follows the RESTORE routine, and I modified this routine in order to remove its dependence on the INCHR routine which I removed as completely unnecessary. If the ISCNTLC routine captures a control-C input, the routine preloads the X-register with ERROR.2 or 0xFF in case ERRFLG is found to be TRUE in the ASROMERR routine. The ASROMERR routine at 0xD865 follows ISCNTLC and the address of this routine is used by DOS 4.5.08H in order to initialize its ERRORADR vector. When the DOS ASONERR flag or Applesoft ERRFLG flag is armed, that is, when its MSB is set and a DOS error occurs, DOS enters ASROMERR by means of ERRORADR. I modified the ASROMERR routine since INCHR is no longer available to strip the MSB from whatever keyboard character is captured by ISCNTLC. I moved and placed the jump to HANDLERR at 0xD8B0 if ERRFLG (or ASONERR) is armed so that the Applesoft STOP statement can still reside at 0xD86E and the Applesoft END statement can follow at 0xD870. The END statement processing uses the C-flag to either jump to RESTART if the flag is clear or it prints the Break error message if the flag is set. The Applesoft CONT statement follows the END statement processing at 0xD896, and it simply restores TXTPTR and CURLIN, the current line number, from TEXTPTR and OLDLIN, respectively.

The Applesoft SAVE statement is found at 0xD8B0 in the unmodified Applesoft, and SAVE uses twenty-five bytes of memory. I removed the SAVE statement as I previously explained and I installed the DOHANDLR jump instruction for HANDLERR processing, the PULL3A routine at 0xD8B3 for Applesoft POP statement processing, and the RDBYTE routine at 0xD8BB in this available space. The Applesoft LOAD statement is still found at 0xD8C9 and I use that Applesoft statement primarily for *c2t* processing. This statement uses the CXREAD routine to read an Applesoft program into memory at PRGTAB for LINNUM number of bytes. If the RUNFLAG flag is armed, LOAD statement processing transfers to the Applesoft RUN statement processing in order to enter the SETPTRS routine. Otherwise, LOAD statement processing enters the ASENTER routine, that is, the Applesoft interpreter. LOAD processing now incorporates the VARTIO routine directly and it discards the PROGIO routine as in the unmodified Applesoft since the Applesoft SAVE statement is removed. This allows the addition of the RD2BIT routine at 0xD8FF that reads two transitions of the audio waveform for the RDBYTE routine and for the CXREAD routine. In operation, the RDBYTE routine requires eight calls to the RD2BIT routine in order to read a full 8-bit byte of audio data. The Applesoft RUN statement follows RD2BIT at 0xD912 and RUN statement processing runs the Applesoft program that is currently in memory either at the top of the program or at a particular line number using the GOSUB2 entry point at 0xD935.

The Applesoft GOSUB statement is processed at 0xD921 and it pushes seven bytes onto the STACK. These seven bytes include two bytes for the NEWSTT return address, two bytes for TXTPTR, two bytes for CURLIN, and one byte for the GOSUB token identification number 0xB0. GOSUB uses the GOTO statement processing to setup the TXTPTR from LINNUM and then GOSUB enters the NEWSTT routine in order to process this statement using the values that have been pushed onto the STACK. The Applesoft GOTO statement processing is conveniently placed after GOSUB processing at 0xD93E. The second half of GOTO processing is called ASROMSET and this routine resides at 0xD955. DOS 4.5.08H uses ASROMSET in order to establish the starting line number for the DOS RUN or for the DOS CHAIN command as long as that line number exists in the Applesoft program that currently resides in memory. An Undefined Statement error is written to the screen if that line number cannot be found in the resident Applesoft program.

The Applesoft POP statement and the Applesoft RETURN statement are both processed at 0xD96B. A glaring Applesoft bug occurs in this processing when Applesoft initializes FORPNT with #NEGONE in the unmodified Applesoft. Obviously, FORPNT+1 must be initialized with #NEGONE prior to the call to GTFORPNT otherwise POP would not be able to cancel a FOR/NEXT frame whose data has been pushed onto the STACK. POP

processing falls into the processing for the Applesoft DATA statement at 0xD995 which skips to the next Applesoft colon : or the End of Line which is demarcated by a NULL byte in Applesoft. The routine that the DATA statement uses to skip ahead to the next Applesoft colon : or End of Line is the DATSCAN routine at 0xD9A3, and DATSCAN directly follows DATA processing. The Applesoft IF statement is processed next at 0xD9C9 and after its expression is evaluated, the IF statement processes a GOTO if that statement is found next or the IF statement syntactically checks for an Applesoft THEN statement before it processes the GOTO. Or, the IF statement simply processes the next Applesoft statement as part of the Applesoft REM statement processing at 0xD9DC. The REM statement also uses DATSCAN in order to scan ahead to the next Applesoft colon : or End of Line. The following Applesoft statement ON at 0xD9EC operates in many similar ways to the IF statement by checking for a following GOSUB statement or a following GOTO statement.

The Applesoft LINGET routine at 0xDA0C utilizes some rather dangerous logic that can cause a potential catastrophic jump to 0xD922 whenever LINNUM contains a value that is between 437,760 and 440,319 or 0xAB00 and 0xABFF. LINGET converts an Applesoft program line number into a 16-bit integer. The maximum Applesoft line number is 25599 and LINGET tests for any line number value that is greater than 25600 or 0x6400. However, if the most significant byte that is in LINNUM+1 happens to be exactly 0xAB, the unmodified Applesoft will compare that value to the value of the GOTO token number which happens to be also 0xAB, find that they are equal, and begin felonious processing at 0xD9F8 on behalf of LINGET. The Applesoft language programmer that coded the LINGET routine was far too lazy to extend the branch at 0xDA1E to 0xD981 where the branch should have been directed to in the first place. That branch is 0x9D bytes in size and it is not possible for that branch to take place, of course. In the modified Applesoft, I moved the PULL3A routine from 0xD9C5 to 0xD8B3 and I moved the SY.ERR2 jump instruction from 0xD981 to 0xD9C5. Now, the branch instruction at 0xDA1E for the comparison of LINNUM+1 and /6400 is only 0x5B bytes away and well within the reach of the SY.ERR2 label. The *Wikipedia 100,000* entry is now solved and no longer an issue in the modified Applesoft.

The Applesoft LET statement at 0xDA46 allows one to assign an Applesoft expression to an Applesoft variable whether that variable is a real variable, an integer variable, or a string variable. If the variable is an integer, the variable is rounded, converted to an integer, and saved using FORPNT. If the variable is a real variable, the FAC floating-point register is copied using FORPNT. And, if the variable is a string variable, the LET statement processing falls into the PUTSTR string routine at 0xDA7B in order to create and install a string descriptor at the address that is in FACMANT+2 and FACMANT+3. The COPYSTR routine at 0xDAB7 discards any temporary string descriptor and it copies the string from its current location in memory into the Character String Pool for safe keeping.

The Applesoft PRINT statement at 0xDAD5 follows the processing for COPYSTR. The PRINT statement handles the Applesoft TAB and SPC statements as well as evaluating all expressions and converting numerical values into ASCII text strings. I did slightly modify the PRINT statement processing by removing the useless CLC instruction at 0xDAE4, I changed the maximum line length from 24 to 32 at 0xDAFF, and I branched to a newly added error handler for a Syntax error. The LINEOUT routine at 0xDB38 calls the FPOUT routine in order to convert a floating-point number into a printable numerical string. The STROUT routine at 0xDB3B uses the STRLIT routine to build a temporary string descriptor for the string that is pointed to by (A,Y) and it terminates the temporary string by looking for a quotation mark or a NULL byte. That temporary string is output to the screen by the following routine STRPRT at 0xDB3E. STRPRT contains three useless lines of assembly instructions and the routine is coded illogically. This routine requires 26 bytes in the unmodified Applesoft and I only require 18 bytes for its complete, correct, and now beautiful implementation. In the unmodified Applesoft, a space character, the Applesoft prompt character, and any other characters that need to be output to the screen is handled by the routines OUTSPC at 0xDB57, OUTPROMT at 0xDB5A, and OUTCHR at 0xDB5C, respectively. Throughout the interpreter, I found that the carriage return

was output via OUTCHR at least twice. Since I still had a few available bytes after modifying STRPRT, I added the PRTCR routine at 0xDB50 that outputs a carriage return and I modified those instances where I could substitute PRTCR in place of the previous instructions. Now, the OUTSPC routine resides at 0xDB53, the OUTPROMT routine resides at 0xDB56, and the OUTCHR routine resides at 0xDB58. The OUTCHR routine ORs the value in FLASHBYT to all ASCII characters that are greater than 0xA0 in order to flash characters if FLASHBYT contains the value of 0x40, otherwise FLASHBYT contains 0x00 for normal characters. More importantly, the OUTCHR routine implements a call to WAIT at 0xFCA8 using the value that is found in SPEEDBYT. In the unmodified Applesoft, the OUTCHR routine *always* calls WAIT. If the value in SPEEDBYT is equal to 0x01, the smallest value possible in the unmodified Applesoft, *each* displayed character is delayed by 29 clock cycles which is around 28 microseconds. The equation for calculating the delay that is based on the SPEEDBYT value in the A-register by WAIT is as follows:

$$\text{Delay} = 2.5 * A^2 + 13.5 * A + 13 \text{ cycles}$$

The clock in the Apple II series of computers is set to provide an average rate of 1,020,484 cycles/second. The calculations that arrive at this value are found in my book *DOS 4.5 Volume and File Disk Management System Second Edition*. In the modified Applesoft, I changed the logic in OUTCHR to bypass the call to WAIT if the value in SPEEDBYT is equal to 0x00, otherwise OUTCHR calls WAIT with the value that is loaded into the A-register from SPEEDBYT. SPEEDBYT is equal to the value that is evaluated from the Applesoft SPEED= statement and exclusively-ORed with 0xFF. Therefore, if SPEED= contains the value of 255, the default speed, SPEEDBYT is set to 0x00. If SPEED= contains the value of 254, SPEEDBYT is set to 0x01.

The INPUTERR routine at 0xDB6F must determine whether an illegal character comes from an INPUT source, a READ source, or a GET source when an illegal character is found somewhere within a numerical field. The READERR routine at 0xDB79 handles READ errors, the ERRLINN routine at 0xDB7D handles GET errors, and the RESPERR routine at 0xDB87 handles INPUT errors. I modified the INPUTERR routine and created another handler for Syntax errors at 0xDB81. Doing so simplifies PRINT error processing and INPUTERR error processing while saving a couple of bytes. RESPERR issues the Reenter request message to handle an INPUT error. The Applesoft GET statement at 0xDBA0 follows the various error processing routines. The GET statement is one of the few statements that can only be used in Running mode. The INPTLIST routine that the GET statement utilizes in order to obtain its input data requires (X/Y) to point to an input buffer which is INPUT+1 and the A-register must be set to the GET command code which is 0x40 for INPUTFLG. I modified the GET routine and changed a JSR/RTS construction to a JMP construction which saves processing time and one byte. The Applesoft INPUT statement at 0xDBB2 follows the GET statement processing. The INPUT statement is another statement that can only be used in Running mode. If the INPUT statement is supplied with a string that prefaces the desired input data, that string is printed by STRPRT, and because I modified STRPRT so nicely, a branch rather than a jump instruction can be used after calling this routine. Otherwise, the INPUT statement prints the prompt character and requests the desired input data. In either case, INLIN obtains the input data and it sets (X/Y) to INPUT-1. The INPUT statement branches to set the A-register to the INPUT command code which is 0x00 and it falls into INPTLIST. Finally, the Applesoft READ statement at 0xDBE2 sets (X/Y) to DATPTR, the A-register to the READ command code which is 0x98, and it falls into INPTLIST. Sandwiched between INPUT and READ is the HEXTIN routine at 0xDBDC. The HEXTIN routine prints the prompt character and jumps directly to INLIN on behalf of INPTLIST. These are certainly well-imagined routines.

The INPTLIST routine at 0xDBEB is the first of six intertwined routines that obtain the requested data on behalf of the GET, the INPUT, or the READ Applesoft statements. The other five routines that bring data into the Apple computer are the INPTITEM routine at 0xDBF1, the INSTART routine at 0xDC2B, the INPTFLG routine at 0xDC99, the FINDATA routine at 0xDCA0, and the INPTDONE routine at 0xDCC7. MSG21 at

0xDCDF prints the >Extra Ignored message and MSG22 at 0xDCEF prints the >Reenter message. INPTLIST simply saves the current value in the A-register to INPUTFLG and it saves the registers (X/Y) to SRCPTR. INPTITEM gets the address of the input variable, sets TXTPTR to point to the selected input buffer, and calls RDKEY at 0xFD0C in the ROM Monitor to get a character for GET, or it branches to FINDATA for READ, or it falls into INSTART for INPUT. The call to RDKEY makes a jump to RDKEY2 at 0xFD13 in the ROM Monitor so it would be faster to simply call RDKEY2 only if this version of the Applesoft interpreter is used in conjunction with the Apple //e ROM Monitor. Leaving this call for RDKEY processing is the safer option. INSTART gets the next input character to build either a floating-point number or a character string variable. Initially, a NULL character or a quote character is used for the string terminator, but if the quote character is not an input character, then the NULL, colon, or comma character can be used for the string terminator. The call to STRLIT2 at 0xDC57 builds a character string starting at (A/Y) and the call to GETINT at 0xDC6A uses TXTPTR to get a floating-point number. INPTFLG uses INPUTFLG to simply direct READ and INPUT data operations. FINDATA is used by READ as indicated above. This routine needs to check for a colon :, an End of Line NULL, or an End of Program NULL. The final routine in this set of data input routines is INPTDONE and this routine restores the (A/Y) registers from SRCPTR and the X-register from INPUTFLG. If a READ is requested, then (A/Y) is saved to DATPTR. Otherwise, an INPUT is requested and if the character at SRCPTR is not a NULL character, then MSG21 is printed. I have made several modifications to FINDATA and to INPTDONE that slightly accelerate the processing of these two routines.

The Applesoft NEXT statement at 0xDCF9 directly follows the various data input routines. Processing for the NEXT statement is involved, complex, and a little difficult to follow. Each 20-byte FOR/NEXT frame is stored in the STACK and NEXT statement processing must retrieve variables and addresses that are contained in that frame in order to control the flow of FOR/NEXT loop processing either back to the FOR statement or to the statement that follows the NEXT statement. I found this routine to be a great opportunity to condense it, to accelerate it, and to utilize the guard bytes that are now included in all floating-point variables that are pushed onto the STACK on behalf of FOR/NEXT processing. The NEXT statement may optionally include the incrementing variable of its companion FOR statement. Slightly faster overall processing can be achieved if that variable is not included with the NEXT statement. The only difference in timing amounts to a call to PTRGET. Regardless, the address of that incrementing variable is either verified or located by GTFORPNT. NEXT locates the STEP value, the END value, and the sign of the STEP value in the frame for this FOR/NEXT loop. The STEP value is added to the incrementing FOR variable, its value saved to VARTAB, and its value compared to the END value. That result, ironically, is used in a calculation to determine if the FOR/NEXT loop has expired. If the FOR/NEXT loop has remaining iterations, both CURLIN and TXTPTR are extracted from the frame and NEXT jumps to NEWSTT processing in order to run the next FOR/NEXT iteration. Otherwise, NEXT enters some very interesting processing which allows NEXT to include all incrementing and comma-separated variables if there exists at least one nested FOR/NEXT loop. A bit of recursive processing is used to handle that particular Applesoft programming construction. I extracted a total of five bytes of unnecessary logic and I accelerated the beginning of this routine. I have also made the size of the FOR/NEXT frame dynamic, so if FACGUARD is not pushed onto the STACK on behalf of FOR, the FOR/NEXT frame size would revert back to eighteen bytes since FACSIZ would be equal to only five bytes.

The FRMNUM routine at 0xDD64, the CHKNUM routine at 0xDD67, the CHKSTR routine at 0xDD69, and the CHKVAL routine at 0xDD6A all follow the processing for the NEXT statement. If NEXT implements its recursive processing, NEXT literally falls into FRMNUM in order to begin processing the next comma delineated NEXT statement in exactly the same way the FOR statement calls FRMNUM at 0xD799. This is truly insightful programming. These four evaluation test routines confirm that the variable under evaluation is a string variable when VALTYP is equal to 0xFF and the C-flag is set or the variable is a numeric variable when VALTYP is equal to 0x00 and the C-flag is clear. To further decode numeric variables, a numeric variable is a floating-point variable when VALTYP+1 is equal to 0x00 or a numeric variable is an integer

variable when VALTYP+1 is equal to 0x80. If the variable under evaluation and VALTYP in concert with the C-flag do not match these specifications, the Type Mismatch error message is issued and the Applesoft interpreter terminates any further program processing. I wonder (rhetorically) why a single value-type variable could not be utilized having the values of 0x00, 0x40, or 0x80 that could easily be tested using the BIT instruction regardless of the C-flag in order to more quickly evaluate variable type for a string variable, for a floating-point variable, or for an integer variable? The FRMEVAL routine at 0xDD7B follows these variable evaluation tests. FRMEVAL utilizes TXTPTR in order to evaluate the expression that resides at that location in memory. Whatever value FRMEVAL extracts from that expression is transferred into the FAC floating-point register. Also, FRMEVAL can be used to evaluate both string and numeric expressions. This routine fully utilizes the precedence code found in the Operator TAG statements in order to properly evaluate the expression using relational operators and/or mathematical operators. When string variables are evaluated for addition, FRMEVAL simply concatenates the strings. Some operations are pushed onto the STACK and evaluated as if they were functions by utilizing the recursive FRMRECUR routine. As in the processing for the FOR and the STEP statements, the SAVOP routine at 0xDDD7 calls FRMRECUR at 0xDDFD to utilize the FRMSTAK routine in order to push the FAC floating-point register onto the STACK.

The FRMSTAK routine at 0xDE10 follows FRMRECUR. FRMSTAK uses a protocol that is different from the DOSTAMT routine which pushes a decremented address onto the STACK in order to engage the processing of the routine for the selected statement by means of CHRGET. Rather, FRMSTAK pulls the address of the calling routine from the STACK, increments that address, and saves that address in INDEX. Once FRMSTAK has pushed the FAC floating-point register onto the STACK, it simply jumps indirectly to the address in INDEX. There is a 1 in 256 chance, perhaps, of finding an address on the STACK that is off a 256-byte page boundary by one byte, or 0xnnFF, where nn is some page value. In the unmodified Applesoft, FRMSTAK assumes that this occurrence will never happen and, therefore, it only increments the LSB of the address without even checking the MSB of the address. Only two routines utilize FRMSTAK, so it is reasonable to smartly position those two routines away from a page boundary in order to prevent such a miscalculation. The necessary logic to modify this routine and not be concerned with page boundary issues amounts to adding three bytes of additional code. Those three bytes as well as an additional three bytes for a modification to the NOTMATH routine are obtained by offloading some of the logic that pushes the FAC floating-point register onto the STACK to another memory location in the modified Applesoft. When FRMEVAL finds no mathematical operations to perform while evaluating an expression, it branches to the NOTMATH routine at 0xDE32 in order to setup its exit by loading FACEXP into the A-register. However, if NOTMATH finds that there is an operation that has been pushed onto the STACK, that floating-point value is pulled from the STACK into the ARG floating-point register in preparation for mathematical processing. Those additional three bytes are now used to pull FACGUARD from the STACK into ARGGUARD.

The FRMELMNT routine at 0xDE60 follows NOTMATH. FRMELMNT processes an array element and it either extracts a numerical value at TXTPTR or it uses TXTPTR that points to a string descriptor and it falls into STRTXT at 0xDE81 in order to initialize (A/Y) to point to the first character in that string for the STRLIT routine. The following NOTFUNC routine at 0xDE90 checks for an Applesoft NOT statement, one of those catch-all statements in the C0:C7 token number range, initializes the Y-register with the EQU tag, and lets the EQUFUNC routine handle further processing. The Applesoft = statement is processed by the Applesoft EQUAL statement at 0xDE98 in order to initialize the Y-register with 0x01 if FACEXP is 0, otherwise, it initializes the Y-register with 0 and it creates an integer from either value. Processing on behalf of the Applesoft FN statement and on behalf of the Applesoft SGN statement is offset by three bytes in the modified Applesoft. The FNFUNC routine at 0xDEA7 evaluates the FN token and jumps to CALLFNC whereas the SGNFUNC routine at 0xDEAE evaluates the SGN token and falls into PARENCHK when that evaluation fails with any token number less than 0xD2. In the unmodified Applesoft, SGNFUNC branches to PARENCHK and

jumps to UNARY for further consideration. The modified Applesoft simply branches to UNARY and saves three bytes of Applesoft space, but more importantly, the modified Applesoft has accelerated FUNCTION statement processing.

Applesoft expressions are typically enclosed in parentheses. The PARENCHK routine at 0xDEB2 calls the CHKOPNP routine to check for an open parenthesis, then calls the FRMEVAL routine and falls into the CHKCLSP routine. The CHKCLSP routine at 0xDEB8 checks for a closed parenthesis using the SYNTAXCHK routine. The CHKOPNP routine at 0xDEBB checks for an open parenthesis. Both parenthesis checks use the SYNTAXCHK routine. The CHKCOM routine at 0xDEBE checks for a comma and falls into the SYNTAXCHK routine. The SYNTAXCHK routine at 0xDEC0 compares the ASCII character that resides in the A-register to whatever the TXTPTR is currently pointing to. That ASCII character is expected to be where TXTPTR is pointing, so if that character is not found at that memory location, the Applesoft interpreter will issue a Syntax error.

The MINUFUNC routine at 0xDECE follows the parenthesis and comma evaluation routines and this routine handles the minus function, it initializes the Y-register with the NEG tag, and the routine falls into the EQUFUNC routine at 0xDEDD that also processes the NOTFUNC routine discussed above. The EQUFUNC routine simply pops the STACK pointer twice and it jumps to the SAVOP routine in order to push that processing onto the STACK. When the FRMELMNT routine finds an ASCII letter A-Z, it branches to the GETIVAL routine at 0xDEDD5. GETIVAL locates the address for this array element. If this array element is a floating-point number, that number is loaded into the FAC floating-point register using INDEX. If this array element is an integer, that integer is converted into a floating-point number. Otherwise, this routine returns with the address of the string array element in VARPTR. The Applesoft SCRN(statement is processed next by means of the SCREEN routine at 0xDEF9 which is one of those odd FUNCTION statements that is syntactically not a FUNCTION1 statement because it contains two numerical expressions that are separated by a comma rather than a single numerical expression. And, the SCRN(statement is syntactically not a FUNCTION2 statement because its two expressions contain numerical variables rather than string variables. Thus, PLOTFS is used to extract those numerical variables for the screen location that the ROM Monitor SCRN function at 0xF871 requires in order to provide its 4-bit color value.

I have had the pleasure to evaluate many complex Applesoft routines, but the UNARY routine at 0xDF09 in the modified Applesoft is one of the most interesting routines which I found necessary to modify in order to add two statements to the Applesoft language repertoire. The UNARY routine manages all of the FUNCTION1 and FUNCTION2 statements from SGN to MID\$, and UNARY even uses its capabilities to process the SCRN(statement and utilize its call to CHRGET on behalf of SCREEN processing. Adding a new Applesoft statement requires Applesoft space for its address and Applesoft space for its statement name in DCI format. Since all token values from 0x80 to 0xEA are already assigned, the next available token number is 0xEB. The address for any new Applesoft statement must follow the address of MID\$ if that statement is to be processed by UNARY. And, new instructions are required to sort out the new token numbers at the beginning of the UNARY routine. In the unmodified Applesoft, the address that is associated with the SCRN(statement at 0xD08A uses the address for PRterr and not for SCREEN at 0xDEF9. In other words, the UNARY routine will never point to the address for processing the SCRN(statement since UNARY already captures its token number. The modified Applesoft requires 0xD08A to contain the address for SCREEN in order for UNARY to process the SCRN(statement. Just allocating the additional space for a new statement address and for a new statement name at the beginning of the Applesoft interpreter is a difficult, first hurdle. Modifying the UNARY routine in order to incorporate a new Applesoft statement is the final hurdle. In UNARY processing, all of the variables that are pushed onto the STACK as well as their order is critical so that FN processing is fully supported. For this single reason, there are no shortcuts that can be implemented when attempting to modify UNARY. First, all FUNCTION1 statements, that is, statements that are not string functions like LEFT\$, RIGHT\$, or MID\$, must branch to 0xDF3A for PARENCHK processing except for SCRN(which branches to

0xDF3D just after PARENCHK processing. Second, if the new Applesoft token number is equal to 0xEB, it must also branch to 0xDF3D since the Applesoft PI statement does not require its expression to be evaluated by FRMEVAL. Third, if the new Applesoft token number is greater than 0xEB, it must branch to PARENCHK processing since the Applesoft LN statement does require expression processing. And fourth, space must be made available in order to accommodate these specific token number branches such that the instructions that form the jump address for the target routine remain at 0xDF3F. I found that I could relocate the final three processing bytes for string FUNCTION2 statements elsewhere in Applesoft space for the three bytes that are needed in order to implement the branches for the Applesoft SCRNC, PI, and LN statements. Once the relocated instructions complete their processing, a jump is made directly to 0xDF3F. I have only added three cycles to string FUNCTION2 statement processing for those relocated instructions. Rather than pushing a decremented address onto the STACK or saving the target address in INDEX and indirectly jumping to INDEX, the UNARY routine saves the target address to JMPADRS+1 and JMPADRS+2 and uses JMPADRS as a subroutine call. Of course, if the byte at JMPADRS is ever clobbered and no longer equal to the absolute address JMP instruction 0x4C, all hopes of Applesoft interpreter recovery would be dismal if not impossible. Should JMPADRS always be refreshed with 0x4C? I wonder. I think it should. But how?

String and Numeric Variables

Applesoft is only generally divided into its collection of statements and routines that assist the management of string variables, floating-point variables, and integer variables. These variables are managed by the use of descriptors whose format depends on the type of variable that it describes. The following is a collection of Applesoft statements and routines that manage string and numeric variables.

The simple binary routines for the Applesoft OR statement at 0xDF4F, the Applesoft AND statement at 0xDF55, the Applesoft FALSE routine at 0xDF5D, and the Applesoft TRUE routine at 0xDF60 all follow UNARY processing. These four routines only need to operate on FACEXP and ARGEXP in order to generate an integer response. The Applesoft LT statement processing at 0xDF65 follows TRUE. LT processing performs relational operations by comparing the FAC floating-point register with the ARG floating-point register and floating the result. Staying on point, the Applesoft STRCMP routine at 0xDF7D compares two string variables. Both the FAC floating-point register and the ARG floating-point register are utilized in order to make comparisons of the content of the two string variables. Like in LT processing, STRCMP processing uses the NUMCMP routine at 0xDFB0 in order to float the results of its string comparison and NUMCMP floats the results of LT comparisons. The Applesoft PDL statement at 0xDFCD converts its expression into an integer in order to call the ROM Monitor routine PREAD at 0xFB1E. Unfortunately, PDL processing will accept an expression that produces any input numerical value from 0 to 255. Though PREAD does not test or mask the X-register for valid input values, PREAD is intended to provide only four paddle read values for paddles 0:3. Applesoft promulgates this nonsense that users will only use Applesoft statements correctly and within their defined ranges whereas I believe users must be informed when they utilize an Applesoft statement incorrectly or beyond the range of useability for that statement. PDL processing does not perform its task well enough in the unmodified Applesoft and corrective action is required. I supplied additional instructions to ensure that the argument that is supplied with the Applesoft PDL statement is within the range of 0:3 or the Illegal Quantity error message is given in response rather than an erroneous numerical result from PREAD. The Applesoft DIM statement at 0xDFD9 follows PDL processing. DIM processing simply allocates the memory for the descriptor of an array variable and its elements. Applesoft automatically provides memory for up to eleven elements for any array variable from 0:10. If an array variable contains more than eleven elements, that array variable must be dimensioned using the

DIM statement or the Bad Subscript error will be issued when a dimension greater than the number 10 is ever utilized. Even though an array descriptor allows for up to 255 dimensions, Applesoft limits the number of dimensions for an array to eighty-eight, that is, DIM A(0,0,...0) can only specify up to eighty-eight zeros. DIM processing uses successive calls to PTRGET to allocate the memory for an array followed by a call to CHKCOM in order to process the next following array element variable.

The Applesoft PTRGET routine at 0xDFE3 is nearly a page in length at 0xEF bytes and it wraps around the Applesoft cold start and warm start entry points at 0xE000 and 0xE003, respectively. PTRGET is an important external routine as well, so its legacy entry address must definitely be conserved. PTRGET is somewhat controlled by the DIMFLG flag and by the SUBFLG flag as mentioned earlier in the processing for the Applesoft FOR statement. All calls to PTRGET initialize DIMFLG to 0 except for the call from Applesoft DIM statement processing which sets DIMFLG to a non-zero value. SUBFLG is initialized to 0 by the STKINIT routine which is part of SETPTRS (or ASTROMCLR) and utilized by DOS 4.5.08H in order to RUN or to CHAIN an Applesoft program. STKINIT is also called by PRTERr whenever an error message is displayed on the screen as a result of a processing error that causes the Applesoft interpreter to terminate further program processing. The GETARYPT routine in the unmodified Applesoft initializes SUBFLG to 0x40 before calling PTRGET and then GETARYPT returns SUBFLG back to 0 before reading from or writing to the cassette recorder. In the modified Applesoft, the GETARYPT routine is removed and all SUBFLG logic is no longer required to test for 0x40. Therefore, the six bytes of SUBFLG logic at 0xE048 is no longer necessary. Like the FOR statement, the Applesoft DEF statement processing initializes the SUBFLG to 0x80 in order to restrict its variables to only simple variables and never array variables. GETFNC processing sets the SUBFLG to any of the token values from 0xC0 to 0xDB. This tells PTRGET precisely which Applesoft statement is requesting the information for the specified variable. PTRGET is a general variable scan routine for the variable name that is found at TXTPTR, and PTRGET searches VARTAB and ARYTAB for that variable name. If PTRGET is unable to locate the variable name, PTRGET creates the appropriate type variable in either VARTAB or in ARYTAB. PTRGET names the descriptor, clears the descriptor, and inserts the address of the value into the descriptor for that variable. PTRGET returns with the address of the variable in VARPNT as well as in (A/Y) for the external user. For some reason that I am unable to fathom, the Applesoft language developers occasionally insert a short routine within a lengthy routine that has absolutely nothing in common with the lengthy routine. The CHKASCII routine is one such routine that I moved to 0xE0DA from the middle of PTRGET to the end of PTRGET. I also moved the value for integer zero from the middle of PTRGET to 0xE105 which is just before the floating-point value for 32768 at 0xE107. CHKASCII is a very short routine that sets the C-flag if the A-register contains an ASCII character from A to Z, otherwise, CHKASCII clears the C-flag. I not only shortened this routine by one byte from the version that is found in the unmodified Applesoft, but I also accelerated this routine as well. The PNTARVAL routine at 0xE0E3 points to the first array value by calculating the size of the descriptor for that array variable which depends on multiplying the value of its dimensions by two and adding in the size of its descriptor #AHADRLen.

The STRSETUP routine at 0xE0FF follows PNTARVAL and the short continuation of the UNARY routine so that the PI and the LN statements can be include in the modified Applesoft. STRSETUP is a 3-byte patch that checks for a closed parenthesis before continuing the processing at 0xE6BC. This software patch is used by LEFT\$, RIGHT\$, and MID\$ statement processing. As mentioned above, the IVALZERO zero value and the FP8000 32768 value follow STRSETUP. The MAKINT routine at 0xE10C evaluates the numeric expression that is currently pointed to by TXTPTR. MAKINT falls into the AYPOSINT routine at 0xE112 in order to test FACSIGN and verify that the evaluation result is positive, or AYPOSINT issues an Illegal Quantity error. That positive result is submitted to the following routine AYINT at 0xE116 which converts the FAC floating-point register into an integer having a maximum value of 32767. AYINT uses the floating-point verification value of 32768 in order to test if the value in the FAC floating-point register is equal to or greater than the verification value. Unfortunately, the Applesoft language developers failed to include the

fourth byte of the mantissa for this verification value. The correct verification value for FP8000 is utilized in the modified Applesoft.

The ARRAY routine at 0xE128 follows AYINT and this routine locates an array element or this routine creates an array element. ARRAY is an extraordinarily lengthy routine of 0x1B6 bytes and ARRAY utilizes the STACK heavily. This routine is one of the most poorly designed and implemented routines in all of Applesoft. Even the Y-register that serves as the dimension counter is pushed onto the STACK. My first modification to ARRAY is to utilize NUMDIM at the very beginning of the routine rather than utilize the Y-register for the dimension counter and simply eliminate some of the STACK complexities. VALTYP is initially pushed onto the STACK before processing the first array dimension and then VALTYP is pulled from the STACK in order to search for this array name once that processing is complete. However, because the Applesoft interpreter overloads its page-zero variables to such a gross extent, ARRAY must push VARNAM onto the STACK before it can utilize MAKINT in order to evaluate the expression at TXTPTR. Once the expression has been evaluated, ARRAY can restore VARNAM from the STACK. I utilize the X-register and the Y-register for different purposes in this first part of ARRAY than how these registers are utilized in the unmodified Applesoft and, as a result, I have tremendously accelerated this part of ARRAY. When ARRAY must create a new array, ARRAY continues its processing at 0xE1BE that first determines if there even exists enough memory for a new array. That new array can be a string array, a floating-point array, or an integer array with a corresponding descriptor size provided. The default array size is set by #DFLTDIM whose value is 0x0B unless DIMFLG indicates that a dimension value is provided. Once the address for the end of the array is computed at 0xE20B, STREND can be evaluated in order to verify that sufficient memory exists for all requested array elements. When ARRAY must locate a specific array element, ARRAY continues its processing at 0xE24B by pulling subscripts from the STACK and comparing those subscripts to the desired element number. The FINDELE label is used at 0xE24B for this processing. This part of ARRAY processing even goes as far as having to multiply subscripts using the MULSUBS routine. Of course, ARRAY must utilize VARNAM to discriminate between string, floating-point, and integer type arrays to correctly calculate the address for the first array element in order to determine the address of the specified element. ARRAY returns with the address of the specified element in VARPNT as well as in (A/Y) for the external user.

The MULSUBS routine at 0xE2AD is a 16-bit integer multiply routine that ARRAY uses in order to multiply two subscript values. The subscript that is found in LOWTR is the multiplicand and the subscript that is found at STRING2 is the multiplier. If the product should ever exceed 32767, the Out of Memory error is issued and ARRAY is unable to create this array within ARYTAB. It is rather a shame that the Applesoft language developers could not have developed a 32-bit integer multiply routine and use that routine for ARRAY as well as for correctly processing the Applesoft RND statement. I have no doubt that MULSUBS and my 32-bit integer multiply routine can be merged. MULSUBS is followed by the processing for the Applesoft FRE statement at 0xE2DE. If a temporary variable exists and it is a string variable, its descriptor is released before FRE calls the GARBAG routine. Thus, the FRE statement forces the call to GARBAG. After GARBAG processing, the number of bytes of Free Space between FRETOP and STREND is calculated, saved as an integer, floated as a floating-point value, and presented to the caller. FRE does evaluate its expression, so its expression must be something legal, but FRE does not utilize the value that the Applesoft interpreter obtains from that expression which might be something useful. The Applesoft POS statement at 0xE2FF follows the FRE statement and this statement extracts the value in CH, the current horizontal screen cursor position relative to the left hand margin of the TEXT window, and presents that value to the SNGFLT routine at 0xE301 in order to generate a single byte integer that is floated to a floating-point value between 0 and 255. The Applesoft interpreter evaluates the expression for POS but POS does not utilize the value that is obtained from its expression similar to the processing for the FRE statement, another wasted opportunity. As in the CH variable, the first character at the left hand margin of the TEXT window on any line has a value of 0. The short Applesoft routine CHKIFDIR at 0xE305 follows SNGFLT, and CHKIFDIR issues the Illegal

Direct error whenever CURLIN+1 is still equal to 0xFF, that is, when the Applesoft interpreter is still processing in Direct Mode and not in Running Mode. I removed a SEC instruction at the beginning of SNGFLT that serves absolutely no purpose whatsoever in order to accelerate interpreter processing.

The Applesoft DEF statement at 0xE313 follows the Illegal Direct and the Undefined Function error messages at 0xE30B and 0xE30E, respectively. DEF processing uses the GETFNC routine to parse and to verify that the next program token number is the Applesoft FN token number and GETFNC obtains the address of that function name in order to initialize FUNCNAM. DEF now expects the Applesoft interpreter to be in Running Mode before it evaluates the FN expression for its required dummy variable. PTRGET initializes VARPNT with the address of this simple variable. To complete the required syntax of this statement, the token number for the Applesoft EQUAL statement is verified last. The first token number after the EQUAL statement, that is, the Applesoft statement that FN is about to process, VARPNT content, and TXTPTR content are all pushed onto the STACK, processing for the Applesoft DATA statement is performed that swaps out the dummy variable, and DEF statement processing continues in the FNCDATA routine. The GETFNC routine at 0xE341 is utilized by DEF processing as described above and by the CALLFNC routine at 0xE354 which follows GETFNC. Once processing returns from GETFNC, CALLFNC pushes FUNCNAM onto the STACK in order to protect its value in case of a nested FN statement before it calls PARENCHK in order to evaluate its numerical expression, and then CALLFNC can restore FUNCNAM from the STACK. CALLFNC restores VARPNT from FUNCNAM and pushes all five bytes of its floating-point value onto the STACK as well as loading the FAC floating-point register with that same value. CALLFNC now pushes TXTPTR and VARPNT onto the STACK, evaluates the FN statement expression with the actual replacement value for the dummy variable, restores the address of the dummy variable from VARPNT into FUNCNAM, and restores TXTPTR before falling into FNCDATA for further processing, just like in DEF statement processing. The FNCDATA routine at 0xE3AF, on behalf of DEF, restores from the STACK the TXTPTR, VARPNT, and the Applesoft statement that is being processed by FN. Otherwise, on behalf of CALLFNC, FNCDATA restores from the STACK the original value for the dummy variable that was specified in the expression of the FN statement. Why the original value that is found in the dummy variable is pushed onto the STACK at 0xE378 in a register loop and pulled from the STACK byte by byte in FNCDATA is very perplexing. Is it because the FNCDATA routine processes data twice as often as the routine at 0xE378? I have no doubt that the Applesoft language developers utilized a number of caulk boards in order to design the DEF FN statement pairing routines and the FN statement processing routines in conjunction with the existing BASIC interpreter routines and their specific capabilities. FN statement processing is a substantial endeavor, but it does heavily compromise STACK resources and it limits the nesting for the utilization of many FOR/NEXT loops or many nested GOSUBs. I wonder if many Applesoft users have even utilized a DEF FN statement or even many nested FOR/NEXT loops or many nested GOSUBs in their Applesoft programs? Having to support the DEF FN statement is the driver for having to push all of those parameters and variables onto the STACK in UNARY.

I would very much like to understand why the Applesoft language developers differentiated between a NULL terminated ASCII string when that string is found at STACK-1, at STACK, or at INPUT. The Applesoft STR statement at 0xE3C5 follows FNCDATA processing. The expression for the STR statement must be a numeric string whose numeric ASCII values are written to and begin at STACK-1 or 0x00FF when STR processing branches to the STRLIT routine with STACK-1 in (A/Y). Whereas FRMEVAL, for example, evaluates an expression for the Applesoft PRINT statement and finds its string values at the beginning of the STACK or 0x0100 when it calls the STRLIT routine with STACK in (A/Y). As previously noted, the STRTXT routine calls the STRLIT routine when its string values are found in the INPUT buffer at 0x0200, *all in the unmodified Applesoft*. This string differentiation plays out in the STRLIT processing where NULL terminated strings that begin on Page 0x00 or that begin on Page 0x02, that is, at STACK-1 or in the INPUT buffer, respectively, are fully processed *and* moved into memory. NULL terminated strings that are found

on Page 1 are *not* fully processed and they are *not* moved into memory. I can certainly understand why STR strings and Page 2 strings are processed differently than Page 1 strings. I simply cannot grasp why so much effort is expended in order for the FPOUT routine to begin NULL terminated strings at two different memory locations at or about the STACK. The STRINI routine at 0xE3D0 follows the STR routine and this routine takes the string address at FACMANT+2 in order to create a descriptor for that string by the following routine STRSPA at 0xE3D8. STRSPA is only utilized by the CHR\$ and LEFT\$/RIGHT\$/MID\$ processing, and STRSPA uses the GETSSPC routine to find space for that string at FRETOP. All of my fuss over the STRLIT routine is focused at 0xE3E2 and STRLIT follows STRSPA. STRLIT builds a descriptor for the string variable at (A/Y) that is either NULL terminated or surrounded by quotation marks. STRLIT may utilize STRINI in order to move that string variable into memory as noted above, but STRLIT always creates a temporary descriptor in page-zero. As will be presented in FPOUT, when FPOUT processes the FAC floating-point register in the modified Applesoft, FPOUT *always* writes the numeric ASCII values for that floating-point value to the *beginning* of the STACK in *all* situations. Therefore, STRLIT must utilize a new strategy in order to differentiate the various string variable sources and what processing to implement. Only half of the bytes that implement the old strategy in the unmodified Applesoft are utilized for the new strategy in the modified Applesoft, and STRLIT is, of course, accelerated.

The PUTNEW routine at 0xE426 follows STRLIT and this routine verifies that there are no more than three 3-byte temporary string descriptors in page-zero, otherwise the Formula too Complex error is issued. Perhaps only overly complex expressions use more than three temporary string descriptors? I have yet to witness this error message. PUTNEW proceeds to copy the temporary string descriptor that is currently in DSCTMP to TEMPST indexed by 0, 3, or 6. No two Applesoft string descriptors will ever point to the same memory location. Even if the ASCII content of two string variables are identical, that string data content will be found at two different memory locations and their string descriptors will contain one or the other memory address. The GETSSPC routine at 0xE454 returns with the address of a memory location within the Character String Pool at FRETOP for the number of bytes specified in the A-register. GETSSPC is only utilized by STRSPA and it returns with the allocated space in (X/Y). If GETSSPC pushes FRETOP down to or past STREND, GETSSPC will issue the Out of Memory error, set the MSB of the GARFLG flag, and it will attempt to recover some Character String Pool memory by calling the GARBAG routine and then repeat the exercise one more time. The GARBAG routine at 0xE484 directly follows GETSSPC.

The GARBAG routine utilizes an algorithm similar in concept to a basic bubble sort algorithm in order to remove all of the unreferenced character string data from the Character String Pool. Thus, GARBAG attempts to compact the Character String Pool contents for GETSSPC or before the DOS 4.5.08H CHAIN command can relocate the SAVs in memory. The processing time for GARBAG to extract all of the little bits and pieces of unreferenced character strings and string characters is proportional to the square of the number of character strings that are currently in use. So, if there are one hundred active character strings it will take four times longer to process those character strings than if there are only fifty active character strings. Many Garbage Collection algorithms have been previously published that accomplish the same results as GARBAG in far less time, but there can be a number of caveats when using some of these other algorithms. For instance, normal Applesoft programs save all character string data in **lower** ASCII where the MSB is clear for each character byte in the string. Furthermore, normal Applesoft programs never allow more than one character string descriptor to point to the same character string data in memory. Multiple character string variable and array element descriptors may each point to identical character string data sets, but these identical sets of character string data must reside at different memory locations. Some Garbage Collection algorithms depend upon these constraints. If either constraint is not found to be true, a catastrophe will result during the course of subsequent Applesoft processing! Of course, if the character string data of an Applesoft program is kept normal and these constraints are observed, there will be no subsequent processing problems. If assembly language routines, possible appendages to the Applesoft program, or other code

segments perform exotic manipulations to the character string descriptors or to the content of the Character String Pool, these constraints might very well be violated. As described above, the Applesoft FRE statement forces a call to GARBAG and the number of bytes of Free Space between FRETOP and STREND is calculated, saved as an integer, floated as a floating-point value, and presented to the caller.

Cornelis Bongers of Erasmus University in Rotterdam, Netherlands, published a brilliant Garbage Collector specification for Applesoft character strings in *Micro*, August, 1982, many, many years ago. According to an article in *Apple Assembly Line*, March, 1984, the speed of Mr. Bongers' algorithm was incredible when compared to the GARBAG algorithm that was designed by the Applesoft language developers. And, the processing time for this algorithm was directly proportional to the number of active character strings rather than to the number of active character strings squared. The only problem with this algorithm was that the magazine that published the algorithm also owned Mr. Bongers' specification. Worse yet, the algorithm was tied to a program called Ampersoft, marketed by Microsparc, then publishers of *Nibble* magazine. It was reported that a license to use Bongers' algorithm was prohibitively expensive at that time.

From the Applesoft Variables section, Table 1 shows the definition of a simple character string variable descriptor as it is found in the Simple Variables memory area. From that same section, Table 2 shows the definition of a character string array variable descriptor as it is found in the Array Variables memory area. Bongers' specification introduced the idea of *marking* active character strings that are located in the Character String Pool. During the first pass through the Simple Variable and the Array Variable descriptors storage areas in memory and through the Character String Pool, Bongers set the third byte in the character string data to its upper ASCII value and he swapped in the address of its character string descriptor in place of the first two bytes of the character string data. He saved those first two bytes of the character string data safely in the address field of its descriptor or of its character string element. The address that was previously in the address field of the descriptor would most likely be changed anyway after all of the character strings are moved up in memory to their final destination. During the second pass through the Simple Variable and the Array Variable descriptors storage areas in memory and through the Character String Pool, he moved all of the active character strings up in memory as far as possible, he unmarked the third character string data byte, he retrieved the first two characters from storage in its descriptor or in its character string element, and he updated the address field to the new memory location where that string now resides in the Character String Pool.

Bongers' algorithm is most efficient when the active character strings are a least three bytes in length, so one- and two-character strings require slightly different handling in his specification. During the first pass through the Simple Variable and the Array Variable descriptors storage areas in memory and through the Character String Pool, he saved the first byte of character string data pointed to by these *short* descriptors into the character string length byte of its descriptor. If the character string length is two, he stored the second data byte into the low address byte of its descriptor. For single byte character strings, he flagged the low address byte with the value of 0xFF. He flagged the high address byte in all *short* descriptors with the value of 0xFF since no character string will ever have a memory address that is equal to or greater than 0xFF00. If he found *short* character strings during the first pass, he set a *short* descriptor's flag and if that flag was found to be set after the second pass was completed, his specification initiated a third pass where he returned the *short* character strings to the Character String Pool with their descriptors updated to their new memory location. *Short* character strings do slow down Bongers' algorithm a little. However, the processing time is still directly proportional to the number of active character strings, and not to the number of active character strings squared. Table 6 illustrate Bongers' specification during the first pass through the Simple Variable and Table 7 through the Array Variable descriptors storage areas in memory and through the Character String Pool.

String Descriptor Before Pass 1							⇒	String Descriptor After Pass 1						
+AS	-AS	1	LSB	MSB	0	0		+AS	-AS	41	FF	FF	0	0
Character String Pool Before Pass 1							⇒	Character String Pool After Pass 1						
41								41						
String Descriptor Before Pass 1							⇒	String Descriptor After Pass 1						
+AS	-AS	2	LSB	MSB	0	0		+AS	-AS	41	42	FF	0	0
Character String Pool Before Pass 1							⇒	Character String Pool After Pass 1						
41	42							41	42					
String Descriptor at ADL/ADH Before Pass 1							⇒	String Descriptor at ADL/ADH After Pass 1						
+AS	-AS	LEN	LSB	MSB	0	0		+AS	-AS	LEN	41	42	0	0
Character String Pool Before Pass 1							⇒	Character String Pool After Pass 1						
41	42	43	44	45	46	47		ADL	ADH	C3	44	45	46	47

Table 6. Bongers Simple Variable Descriptor Processing in Pass 1

String Element Before Pass 1				⇒	String Element After Pass 1									
1	LSB		MSB		41	FF	FF							
Character String Pool Before Pass 1				⇒	Character String Pool After Pass 1									
41					41									
String Element Before Pass 1				⇒	String Element After Pass 1									
2	LSB		MSB		41	42	FF							
Character String Pool Before Pass 1				⇒	Character String Pool After Pass 1									
41		42			41	42								
String Element at ADL/ADH Before Pass 1				⇒	String Element at ADL/ADH After Pass 1									
LEN		LSB			LEN	41	42							
Character String Pool Before Pass 1							⇒	Character String Pool After Pass 1						
41	42	43	44	45	46	47		ADL	ADH	C3	44	45	46	47

Table 7. Bongers Array Variable Element Processing in Pass 1

Pass two in Bongers' specification uses only the information that is in the Character String Pool data in order to move all currently active character string variables up in Character String Pool memory as far as possible. This is accomplished by initializing a string pool pointer and a character string pointer beginning at HIMEM and then searching down in memory to FRETOP for any upper ASCII character bytes. Once an

upper ASCII character byte is found, its character string descriptor is located and retrieved at the memory location that is two bytes prior to the upper ASCII character byte. That character string descriptor contains the length of the character string and the first two ASCII characters that were copied from the data of that character string. Those two characters may be safely copied back into its character string data and the upper ASCII character byte that *marked* this string data can be changed back to its lower ASCII value. The character string length can now be subtracted from the current character string pointer address, the new character string address can be copied to the second and the third bytes in its character string descriptor, and the character string data can be copied to its new Character String Pool location. However, the character string data must be copied from its last character to its first character rather than from its first character to its last character in order to prevent possibly overwriting part of the character string data. Once the string pool pointer reaches the original address in FRETOP, the current character string pointer address becomes the new address for FRETOP if the *short* descriptors flag is not set. If the *short* descriptors flag is set, then a third pass must be made through the Simple Variable and the Array Variable descriptors storage areas in memory and through the Character String Pool according to Bongers' specification. A memory pointer is initialized to VARTAB and a search is made for the 0xFF byte in either the fifth byte of a Simple Variable descriptor or in the third byte of an Array Variable element. If the prior byte also contains an 0xFF byte, then the descriptor is for a single byte character string, otherwise the descriptor is for a two byte character string. The current character string pointer is adjusted for one or for two characters, the character string data is copied from its descriptor to the Character String Pool, and the character string pointer address is copied to its character string descriptor. Once the memory pointer reaches STREND, the current character string pointer address becomes the new address for FRETOP.

I must again emphasize that Bongers' specification depends upon two very important caveats: *normal Applesoft programs* save all character string data to memory in **lower** ASCII, that is, with the MSB of each character byte cleared, and *normal Applesoft programs* never allow more than one character string descriptor to point to the same character string data in memory. Bongers' algorithm will **fail** if a user should program something like `A$ = CHR$(193)` rather than `A$ = CHR$(65)`. Bongers' algorithm will **fail** if an assembly language routine should modify two character string descriptors to point to the same character string data in the Character String Pool. Therefore, reasonable care must be given when creating Applesoft programs and/or assembly language routines that take the above caveats seriously in order to exact the stupendous benefits in using a garbage collector routine that is based on Bongers' specification. Armed with only these limited and published details of Bongers' specification that I just presented, my analysis of those details, and my complete understanding of Tables 1, 2, and 3 as well as my generation of Tables 6 and 7, my attempt to recreate Bongers' algorithm resulted in an assembly language routine that was 0x200 bytes in size. This necessitated creating a suitable Applesoft test program that would verify the accuracy of my implementation of Bongers' specification and to confirm to my satisfaction that no character string was altered in length, modified in content, or destroyed during VARTAB, ARYTAB, or Character String Pool processing. My ultimate goal would be to replace GARBAG in the Applesoft interpreter with my version of Bongers' specification. In the unmodified Applesoft, GARBAG occupies 0x113 bytes of space and there is 0x70 bytes of additional space available in the CX ROM area at 0xC600:C66F just prior to where I placed the *SWEET16* Metaprocessor code at 0xC670. If the CX ROM space is used, then CX ROM memory management must also be incorporated into the new garbage routine. When all available memory is totaled, my garbage routine must fit within 0x183 bytes if it is to replace GARBAG.

Certain decisions must be made that, hopefully, do not cause the introduction of more processor cycles than absolutely necessary in order to compact an assembly language routine. Example strategies would be to limit subroutine calls in the inner-most loops and to limit the pushing and popping of variables onto the STACK. Sometimes, simply reorganizing the order of a number of processing loops can greatly simplify the routine and eliminate having to re-initialize registers. Keeping the MSB address of a variable in a register

when addresses are often compared or manipulated can help simplify and even accelerate the routine as well. I have no doubt that Mr. Bongers could have condensed my initial attempt in programming his specification down from 0x200 bytes to 0x183 bytes where six of those bytes are required for CX ROM memory management. My initial attempt to condense my garbage routine could not meet the goal of 0x183 bytes unless I removed the *short* descriptors flag that signaled whether a third pass was necessary, so I always made a third pass. Many times, it is helpful to just take a break from any difficult programming task, walk away, and work on something that is demanding in other ways. Thus, when I returned to my garbage routine, I took another fresh look and I found several additional strategies that could condense my routine even further, and even allow the use of the *short* descriptors flag. Hurray! I was able to place one segment of the routine into the 0x70 bytes that are located in the CX ROM area and the other segment into the 0x113 bytes where GARBAG normally resides. All that was left for me to do was the testing, the timing, and the verification of the routine once I fully installed the routine into the Applesoft interpreter.

The CAT2STR routine at 0xE597 directly follows GARBAG and CAT2STR concatenates two string variables. The string variable address of the first string must be pushed onto the STACK so that the address and length of the second string variable can be evaluated. If the sum of these two strings exceeds 256 bytes, CAT2STR issues the String too Long error message and processing for the Applesoft program terminates. I moved this error message to the end of the CAT2STR routine in order to accelerate its processing. The MOVINS routine at 0xE5D4 follows CAT2STR, and this routine extracts the length and the address of a string variable from its string descriptor and transfers those values into the three registers. MOVINS falls into the MOVSTR routine at 0xE5E2. MOVSTR uses the A-register for the length and (X/Y) for the address of the string variable in order to move the string data from its current memory location to the destination address in FRESPC. The variable FRESPC is incremented with the string length. The routine following MOVSTR is the FRESTR routine at 0xE5FD and FRESTR is used only by the GETSTRLN routine in order to call CHKSTR and fall into the FREFAC routine at 0xE600. FREFAC is used by STRPRT and STRCMP and by the FRE statement routine to simply obtain the address of a descriptor. It seems that GETSTRLN could have called CHKSTR and then call FREFAC and save yet another programming symbol from the Symbol Table. FREFAC falls into the FRETMP routine at 0xE604 in order to release a single page-zero temporary string and reduce the value of temporary strings in LASTPT. FRETMP is called with the address of a descriptor in (A/Y) in order to extract the length of the string variable into the A-register and the address of the string variable into (X/Y) and initialize INDEX with that same address. The FRETMS routine at 0xE635 follows FRETMP. FRETMS is called having values in (A/Y) that are compared to LASTPT. If they are equal, LASTPT is reduced by 3.

The Applesoft CHR\$ statement at 0xE646 follows FRETMS. CHR\$ processing converts its input expression into a single byte integer and uses STRSPA to obtain space at FRETOP for a single-byte string variable where the input variable can be stored. The Applesoft LEFT\$ statement at 0xE65A follows CHR\$ processing. The LEFT\$ statement processing provides several common routines that both RIGHT\$ and MID\$ processing can utilize. All three string manipulation routines use the STRSETUP routine in order to process the input expression and evaluate that expression for its first parameter, a string variable. LEFT\$ processing uses STRSPA to reserve space at FRETOP for the non-zero number of characters that it extracts from the input string variable. The Applesoft RIGHT\$ statement at 0xE686 follows LEFT\$ processing and it utilizes much of the LEFT\$ routine after subtracting the non-zero number of characters to extract from the input string variable. How clever is that? The Applesoft MID\$ statement at 0xE691 follows RIGHT\$ processing and far more processing effort is utilized in MID\$ in order to evaluate all components of its input expression. Either one or two numerical parameters follow the input string variable in the MID\$ expression, and the value of those numerical parameters can range from 1 to 255 according to official Applesoft documentation. Since STRSETUP handles the first parameter as in LEFT\$ and RIGHT\$ processing, that routine correctly detects an error if the first numerical parameter is zero. Unfortunately, the official Applesoft documentation errors concerning the legal range of the second numerical parameter if it is given in the MID\$ expression. The

unmodified Applesoft apparently allows the second numerical parameter to equal zero and not issue an error message. Of course, setting the second numerical parameter to zero in the MID\$ expression does not provide any data output whether that is intended or not. However, setting this second numerical parameter to zero should cause the Applesoft interpreter to issue the Illegal Quantity error message and terminate any further processing just like STRSETUP does when it finds the first numerical parameter equal to zero. In order to manage this glaring deficiency, the output of the call to the GETBYT routine at 0xE69F must not return a value of zero in the X-register. Three bytes of memory are required in order to test the X-register and branch if its value is equal to zero. Those three bytes of memory can be derived in several ways by offloading some of the instructions in MID\$ processing or in the following routine so as not to perturb the entry address for the next Applesoft statement. The cleanest modification that would insert the least processing delay would be to move the entry of STRSETUP elsewhere and jump back to finish the STRSETUP routine. Using this strategy, STRSETUP is actually entered at 0xE0FF as already noted, and at 0xE102 the routine returns to its normal processing location at 0xE6BC and this patch costs this routine only three additional processor cycles. The STRSETUP routine extracts its return address from the STACK and saves that address, it pops and discards the return address to the UNARY routine, it pops the integer byte value of the second numerical parameter in the expression, and it pops the address of the descriptor for the string variable in the expression. Lastly, STRSETUP pushes the return address it previously saved and tests the integer byte value of the first numerical parameter for zero. If that integer byte value is zero, STRSETUP issues the Illegal Quantity error message and it terminates any further processing.

The Applesoft LEN statement at 0xE6D6 follows STRSETUP. LEN processing utilizes the GETSTRLN routine to evaluate the LEN statement expression in order to extract the length of the string variable that is found in its descriptor, it sets VALTYP to numeric, and it returns with the length of the string variable in the Y-register. That string variable length is floated and returned to the user as an integer value. The GETSTRLN routine is at 0xE6DC and it follows LEN processing. As I indicated above and in my opinion, the GETSTRLN routine wastes a symbol in using FRESTR to call CHKSTR which falls immediately into the FREFAC routine. Perhaps the Applesoft language developers had additional plans to utilize the FRESTR routine for other purposes? The Applesoft ASC statement at 0xE6E5 also utilizes the GETSTRLN routine in order to evaluate its expression for the ASCII value of the first character in a string variable. If GETSTRLN should ever return zero, the ASC statement routine will issue the Illegal Quantity error message and terminate further Applesoft processing. Official Applesoft documentation warns that the ASC statement will generate a Syntax error message if the statement should attempt to process a control-@ as in ASC(CHR\$(0)). I found that no such error is generated and that the value of zero is returned as expected. Inaccurate documentation does complicate these simple routines unnecessarily. The routine GETBYTC at 0xE6F5 follows the ASC statement routine and GETBYTC reads the next character at TXTPTR and falls into the GETBYT routine at 0xE6F8. GETBYT evaluates the expression at TXTPTR, returns with its byte value in the X-register, and falls into the CONVINT routine at 0xE6FB. CONVINT converts the value that is found in the X-register to a positive single byte integer in the FAC floating-point register.

The Applesoft VAL statement at 0xE707 follows CONVINT processing and VAL processing is the last user of GETSTRLN. If GETSTRLN returns with a value of zero, a floating-point value of 0 is returned in the FAC floating-point register to the caller. Otherwise, VAL processing collects all of the numeric values in the expression while ignoring all space characters up until the first non-numeric character. The resulting value is returned as either a floating-point number or as an integer number. GETSTRLN returns with the number of characters that comprise the numeric content in the VAL expression including the space characters. TXTPTR, which points to the first character after the) in the VAL expression, is copied to STRING2 for safe keeping and INDEX, which points to the beginning of the VAL expression, is copied to TXTPTR in order to make use of the powerful CHRGET routine. The value from GETSTRLN is added to INDEX and saved to DEST. DEST now points to the end of the VAL expression, copies its last character to the STACK for safe keeping,

and replaces that character with a NULL character. This effectively allows the GETINT routine to evaluate this entire expression for its numeric value including any +, -, ., and E characters as well as + and - characters that might be associated with the E character if scientific notation is encountered within this expression. After GETINT processing, DEST is used again in order to restore that last character and TXTPTR is restored from STRING2 by the STRCOPY routine at 0xE73D. The GETINT routine performs its task as intended unless the VAL expression happens to contain a string variable that resides at the very top of the Character String Pool. DEST may indeed contain the address of 0xC000 and GETINT may not even encounter the NULL character until somewhere in the 0xC0 page, though unlikely. This situation will always be potentially problematic for GETINT whenever HIMEM is initialized to 0xC000. Fortunately, DOS 4.5.08H utilizes main memory from 0xBE00 to 0xBFFF, and HIMEM is always initialized to 0xBE00.

The GETASNUM routine at 0xE746 follows STRCOPY. GETASNUM and the following routine COMBYTE at 0xE74C, together, evaluate two comma-separated expressions where the first expression provides a 16-bit integer value in LINNUM and the second expression provides an 8-bit integer value in the X-register. The GETADDR routine at 0xE752 follows COMBYTE and GETADDR converts the 16-bit integer value in LINNUM to a floating-point number in the FAC floating-point register. The Applesoft PEEK statement at 0xE764 utilizes GETADDR in order to return the value that resides at the memory address that is obtained when GETADDR evaluates the PEEK statement expression. Following PEEK processing is the Applesoft POKE statement at 0xE77B. POKE processing utilizes GETASNUM in order to save the value that resides in the X-register to the memory address that is found in LINNUM. The Applesoft WAIT statement at 0xE784 is the final user of GETASNUM in order to obtain a 16-bit memory address in LINNUM and an 8-bit value that is saved to FORPNT. If a third variable is used with the WAIT statement, that value is obtained by means of COMBYTE, and that value is saved to FORPNT+1, otherwise FORPNT+1 is initialized with zero. WAIT statement processing occurs in a very simple loop where the value at (LINNUM) is exclusively-ORed with FORPNT+1 and ANDed with FORPNT. If that processing produces a zero result, the processing loop is repeated. As soon as the processing produces a non-zero result, the loop processing is terminated and no values are returned to the user. The Applesoft WAIT statement can provide a means to pause Applesoft processing until very specific or very precise conditions are met, either by internal values or by external values such as the keyboard, the annunciators, or the RND variable that is incremented by XKEYIN or by CXKEYIN in the CXROM.

Floating-Point Arithmetic Operations

Applesoft is only generally divided into its collection of statements and routines that perform floating-point arithmetic operations. These arithmetic operations include subtraction, addition, multiplication, division, and the square root function. The following is a collection of Applesoft statements and routines that perform floating-point arithmetic operations.

The Applesoft SUB statement at 0xE7A7 begins the Applesoft floating-point arithmetic operations. SUB processing follows WAIT processing and the converted floating-point value for the PI variable at 0xE7A1 and its guard byte FPIGUARD at 0xE7A6. I was able to modify the Applesoft floating-point arithmetic operations, the floating-point register to memory routines, and the floating-point register to register routines in order to utilize guard bytes in far more arithmetic operations once I had sufficient space for these improvements. SUB processing begins like the other three arithmetic operations to load the ARG floating-point register with the floating-point value that is pointed to by INDEX. The FAC floating-point register already comes pre-loaded with the first numeric variable after the Applesoft interpreter has evaluated the expression for the SUB statement. FACSIGN is inverted and exclusively-ORed with ARGSIGN and saved to

XORSIGN, the two numeric values are added, and their sum is returned in the FAC floating-point register. The Applesoft ADD statement at 0xE7BE follows SUB processing. As in SUB processing, ADD processing begins by calling LOADARG to load the ARG floating-point register with the floating-point value that is pointed to by INDEX. The FAC floating-point register already contains the value of the first variable in the expression for the ADD statement. The processing that is common to both SUB processing and ADD processing begins by messing around with their respective guard bytes. Guard bytes are left unmodified in the modified Applesoft. Period! In order for two floating-point values to be added when these values each utilize an exponent and a mantissa, the exponents must be equal in order for their mantissas to be added properly, and any carry bit that results from this addition is added to the common exponent. FACEXP and ARGEXP are subtracted and the mantissa of whichever exponent is smaller is adjusted to the right that many bits. If the number of bits is greater than seven, the mantissa is shifted to the right by one byte for every 8 bits. The mantissa is further shifted to the right for any remaining bits. Depending on the value of XORSIGN, the two mantissas are subtracted or added, and the resulting floating-point value is normalized. Floating-point normalization is always performed so that the floating-point value can utilize all of its mantissa and guard bits to their greater efficiency. The mantissa checks its MSB and the normalization routine shifts the mantissa and the guard bits left until that MSB is set in order to create the implicit high-order one bit to yield a full 40-bit significand. Every time the mantissa is shifted left, the exponent is decremented. The MSB is swapped for the sign bit from XORSIGN to complete the normalization routine. Besides the normalization routine, SUB processing and ADD processing utilize other routines to zero the FAC exponent, to compliment the FAC mantissa, to increment the FAC mantissa, and to shift the FAC mantissa up a variable number of bits. FACGUARD plays a prominent and decisive role in all of these normalization routines.

I have no idea why the Applesoft language developers decided to follow the SUB and the ADD routines with the Applesoft LN statement at 0xE941. But, more importantly, why those developers called this routine their LOG statement routine rather than their LN statement routine since this routine calculates the *natural* logarithm rather than the base-10 logarithm of a positive floating-point number. 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 . Electrical engineers and tactical radar engineers, for example, all utilize this logarithmic nomenclature to ensure that their equations are using the correct logarithmic values in their calculations. Many floating-point variables are explicitly included as well as the polynomials that are utilized to calculate the natural logarithm before Applesoft tackles LN statement processing. These variables begin at 0xE913 and they end at 0xE940, and they follow the continuation of the Applesoft PDL processing at 0xE908. There exists a number of methods to calculate the natural logarithm of any positive floating-point number.

The LN statement routine factors out and saves n , the powers of 2 from the exponent of the input argument minus EXPBIAS or 0x80, and it 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 only 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 LN routine saves and replaces the exponent of the input argument with EXPBIAS, the routine adds the square root of 0.5 to the argument that resides in the FAC floating-point register and that register

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.

$$1 - \frac{\sqrt{2.0}}{x + \sqrt{0.5}} = \frac{x - \sqrt{0.5}}{x + \sqrt{0.5}} = \frac{x\sqrt{2.0} - 1}{x\sqrt{2.0} + 1}$$

Polynomial	Index	Applesoft Value	Base-10 Value	True Value	Base-10 Value
Entries - 1	0x00	0x03	3	0x03	3
(2/7)(√2)*x^7	0x01	0x7F 3E56CB79	0.43425594	0x7F 4EE115F3	0.404061018
(2/5)(√2)*x^5	0x06	0x80 139B0B64	0.57658454	0x80 10D0C292	0.565685425
(2/3)(√2)*x^3	0x0B	0x80 76389316	0.96180075	0x80 715BEEF0	0.942809042
2(√2)*x	0x10	0x82 38AA3B20	2.88539007	0x82 3504F336	2.828427125

Table 8. Applesoft Natural Log Routine Polynomials

The FAC floating-point register is now in the correct format for Taylor series expansion, and the Applesoft language developers decided to utilize only four polynomials to calculate the natural logarithm. However, those developers modified these four polynomials from their theoretical value. All of the polynomials have been preprocessed, -0.5 is added to the FAC floating-point register, 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. Table 8 shows the four modified values that are used for the Taylor series expansion polynomials that process the transformed input argument to calculate its natural logarithm. The modified values deviate, always higher, than the theoretical Taylor series expansion values. However, even if the theoretical values shown in Table 8 are used in conjunction with the theoretical values for the fifth and sixth polynomials, the resulting logarithmic value is still not even close to the logarithmic value that is obtained from using the modified polynomials values. Obviously, I have no idea how these modified polynomial values were calculated nor do I know the mathematical rationale that was utilized for these calculations.

LN processing is completed when it cleverly falls into the Applesoft MULT statement at 0xE97F in order to multiply the FAC floating-point register with the natural logarithm of two. As in SUB and ADD processing, MULT processing begins by calling LOADARG to load the ARG floating-point register with the multiplicand, a floating-point value that is pointed to by INDEX. The FAC floating-point register already contains the multiplier, the value of the first variable in the expression for the MULT statement. The MULMANT floating-point register contains the mantissa product. Both the MULT routine and the DIV routine utilize a common PROCEXP routine that processes the FACEXP and ARGEXP exponents. If ARGEXP is 0, both FACEXP and FACSIGN are cleared to zero and the arithmetic operation is terminated. Otherwise, ARGEXP is added to FACEXP. If their sum is less than 0x80 and the carry flag is clear, then both FACEXP and FACSIGN are cleared to zero and the arithmetic operation is terminated as above. On the other hand, if their sum is greater than 0x80 and if the carry flag is set, an Overflow error message is generated and the arithmetic operation is terminated. Otherwise, when the carry flag is clear, their sum is added to EXPBIAS and that sum is stored at FACEXP. If FACEXP is not equal to 0, the value that is stored at XORSIGN is copied to FACSIGN. However, if FACEXP is equal to 0 in the unmodified Applesoft, then 0 is stored at FACSIGN. This final logic cannot be more wrong! This logic makes the quotient of a very small number positive without regard to the sign of the divisor or the sign of the dividend. The modified Applesoft fixes this glaring error.

At the conclusion of MULT or DIV processing, the mantissa is always copied from the MULMANT floating-point register to the FAC floating-point register and the exponent in the FAC floating-point register is finalized with that mantissa. As in all floating-point and integer multiply routines, the product register, that is, MULMANT, is first cleared to zero. In order to facilitate the MULT routine, each byte of the multiplier participates in the routine individually such that if its value is zero, then the product register is simply shifted right by one byte. Otherwise, that byte is shifted to denote when to add the multiplicand to the product register. FACGUARD is the first multiplier byte, and then it is immediately cleared to zero so that it can participate in multiplicand and product register addition. Both FACGUARD and ARGGUARD did not participate in multiplicand and product register addition in the unmodified Applesoft.

The LOADARG routine at 0xE9E3 follows MULT processing. LOADARG uses (A/Y) to initialize INDEX and copy the floating-point number at that address into the ARG floating-point register. LOADARG also manages ARGSIGN, and with FACSIGN, LOADARG produces a value for XORSIGN. LOADARG initializes ARGGUARD to 0 and exits with the Y-register equal to zero and the A-register equal to FACEXP, a value that each of the four arithmetic operations check for zero after calling LOADARG. Following LOADARG is the PROCEXP routine at 0xEA10 which was explained above. Following PROCEXP is the ZEROFERR routine at 0xEA2E that checks for a zero or overflow error, the Overflow error message at 0xEA32, and the Division by Zero error message at 0xEA35. Following the error messages are two shortcut routines that multiply or divide the FAC floating-point register by ten, respectively. The MULFAC10 routine at 0xEA3A utilizes COPYF2A in order to copy the FAC floating-point register to the ARG floating-point register without using any roundup facilities as done in the unmodified Applesoft. COPYF2A also copies FACGUARD to ARGGUARD without modification. MULFAC10 adds the value of 2.0 to FACEXP effectively multiplying the FAC floating-point register by four, adds the ARG floating-point register to the FAC floating-point register, and increments FACEXP again, effectively multiplying the FAC floating-point register by two. In effect, MULFAC10 processing produces $(4 + 1) * 2 = 10$. The DIVFAC10 routine at 0xEA55 follows MULFAC10. The DIVFAC10 routine also uses COPYF2A, it uses LOADFAC to load the floating-point parameter value of 10.0, and then it uses DIV to produce the desired quotient. DIVFAC10 is followed by the Applesoft DIV statement at 0xEA66. DIV processing begins by calling LOADARG to load the ARG floating-point register with the dividend, a floating-point value that is pointed to by INDEX. The FAC floating-point register already contains the divisor, the value of the first variable when the expression of the DIV statement is evaluated. The DIV routine subtracts FACEXP from zero before it calls PROCEXP like in MULT processing as I explain above. Unlike in MULT processing, however, DIV increments FACEXP and tests FACEXP for zero. I believe the MULT routine is beautifully written and I could not have offered a better organization for its various processing units. On the other hand, the DIV routine is not beautifully written and its processing units are not well organized. Besides the disorganization within the DIV routine, both FACGUARD and ARGGUARD do not participate in dividend and divisor addition in the unmodified Applesoft. Along with the quotient in the MULMANT register, FACGUARD provides only two valid upper bits, the most bare minimum of information for any useful floating-point normalization once the MULMANT register is copied to the FAC floating-point register and normalized. It is pointless to describe DIV processing in the unmodified Applesoft.

DIV processing in the modified Applesoft initializes the X-register with 0xFB as a five iteration counter and pointer because DIV processing intends to produce four mantissa bytes and one guard byte. The A-register is initialized with 0x01 so that that register can simultaneously act as an eight bit counter as it gathers bits for a quotient byte value. At the top of the DIV processing loop, the Y-register is solely utilized in order to make a preliminary comparison of the dividend and the divisor mantissas along with their guard bytes, and capture the state of the carry flag before it is shifted into the LSB of the A-register. If the carry flag is set, DIV processing performs the actual subtraction where the dividend becomes the minuend and the difference is stored as the new dividend. Whether the dividend is replaced or not, ARGMANT and its guard byte are shifted one bit to the left and its MSB is shifted into the carry flag. If that carry flag

is set or if ARGMENT is positive determine the two conditions that will occur if, indeed, the dividend is still smaller than the divisor, and another preliminary comparison of the two mantissas at the top of the processing loop is unnecessary. Only when the carry flag is clear and when ARGMENT is negative does DIV processing make another preliminary comparison of the dividend and the divisor mantissas. This strategy is brilliant and it confines all of the DIV processing to what is only necessary in order to efficiently progress to the next processing iteration. In capturing the state of the carry flag that is shifted into the A-register from either the preliminary comparison of dividend and divisor or from the shift of ARGMENT one bit to the left as the A-register is also shifted one bit to the left, and as long as the MSB of the A-register, which shifts into the carry flag, is clear, the next processing iteration will continue for that quotient byte. Similar in how the MSB of the A-register is set in MULT processing in order to provide an eight bit counter as the A-register is shifted right, the A-register provides an eight bit counter as the A-register is shifted left in DIV processing. As soon as this eight bit counter expires, the X-register is incremented and that register is used as the index in order to save the quotient byte value that currently resides in the A-register to MULMANT as the most significant or the next significant byte in the mantissa of the developing quotient. The X-register is also used to determine if DIV processing should continue to calculate the next quotient byte value and again initialize the A-register with its counter value of 0x01 or to conclude DIV processing entirely and save the last quotient byte value to FACGUARD, copy MULMANT into FACMANT, and normalize the FAC floating-point register. I modified DIV processing and reordered some of its processing units that optimize its processing flow. Even though I added full FACGUARD and ARGGUARD participation in DIV processing, I am able to extract twelve processing bytes for other uses. Rather than produce a FACGUARD with only two valid bits as in the unmodified Applesoft, FACGUARD now contains eight valid bits prior to FAC floating-point register normalization. DIV processing is far more meticulous and this additional processing time helps to reduce Applesoft mathematical irregularities in subsequent floating-point numerical values for those Applesoft arguments where arithmetic division is necessary.

DIV processing falls directly into the COPYM2F routine at 0xEAE6. The COPYM2F routine is only used by MULT and by DIV processing, and this routine jumps directly to floating-point mantissa normalization in order to finalize the floating-point exponent. Following the COPYM2F routine is the LOADFAC routine at 0xEAF9, and LOADFAC uses (A/Y) to initialize INDEX and copy the floating-point number at that address into the FAC floating-point register. LOADFAC initializes FACSIGN and it exits with the Y-register equal to zero, FACGUARD equal to zero, and the A-register equal to FACEXP. Following the LOADFAC routine is the COPYFAC routine at 0xEB2B and its additional entry points for COPYF2T2 at 0xEB1E, COPYF2T1 at 0xEB21, and COPYF2FR at 0xEB27. The COPYFAC routine is the only register copy routine that calls the RNDUP routine in order to process FACGUARD and possibly roundup the FAC floating-point register before COPYFAC saves (X/Y) to INDEX. Before COPYFAC copies the content of the FAC floating-point register to memory that is pointed to by INDEX, COPYFAC loads the X-register from FACGUARD rather than destroying FACGUARD and initializing FACGUARD with 0 as in the unmodified Applesoft. Following COPYFAC is the COPYA2F routine at 0xEB53 which is written as a register loop routine that saves memory at the expense of processing time. This routine is only used by ADD processing when LOADARG returns FACEXP equal to 0 and by exponentiation processing. To better serve exponentiation, I extended the COPYA2F routine and included the copy of ARGGUARD to FACGUARD rather than initializing FACGUARD with 0 as in the unmodified Applesoft. The COPYF2A routine at 0xEB63 immediately follows COPYA2F. The COPYF2A routine is also written as a register loop routine in the unmodified Applesoft and it requires very little space for this very popular routine that is used throughout Applesoft. For that reason alone, I removed its register loop in the modified Applesoft and extended this routine into a free area of Applesoft space. Even though I added three cycles for a JMP instruction, I have tremendously accelerated this routine and I have copied FACGUARD to ARGGUARD rather than initializing FACGUARD with 0 as in the unmodified Applesoft. In fact, the COPYF2A routine in the unmodified Applesoft does not even touch or update the value in ARGGUARD. All of the routines that utilize COPYF2A now benefit from having a valid 40-bit

mantissa in the ARG floating-point register. The infamous RNDUP routine at 0xEB70 follows COPYF2A. The utilization of this routine alone has been at the center of many of the Applesoft mathematical irregularities when a floating-point numerical value is passed to subsequent arithmetic operations having only a 32-bit mantissa when it could easily have a complete and valid 40-bit mantissa.

The SIGNCHK routine at 0xEB82 follows RNDUP and SIGNCHK tests FACEXP for having a zero value or FACSIGN for having a negative or a positive value. If FACSIGN is negative, the A-register is returned containing 0xFF and the C-flag is set. If FACEXP is 0, the A-register is returned containing zero and the C-flag is indeterminate. If FACSIGN is positive, the A-register is returned containing 0x01 and the C-flag is clear. The value in the A-register and/or the setting of the C-flag are far easier to test for these simple conditions of the FAC floating-point register. The Applesoft SGN statement at 0xEB90 follows SIGNCHK, and the SGN routine simply calls SIGNCHK and falls directly into the FLOAT routine at 0xEB93 in order to create a floating-point value of the SIGNCHK result using an exponent for a single byte value. SGN exits through floating-point normalization. The Applesoft ABS statement at 0xEBAF simply shifts FACSIGN one bit to the right, thus shifting a zero bit into its MSB position. If the input argument is negative, for example, that argument will now be processed as a positive argument. The FPCOMP routine at 0xEBB2 follows ABS processing. The FPCOMP routine saves (A/Y) to DEST and compares the floating-point value at DEST to the value that currently resides in the FAC floating-point register. In order to utilize this routine more profoundly, I modified the beginning of FPCOMP to initialize the X-register from either ARGGUARD or from FACGUARD. The X-register is now compared to FACGUARD rather than comparing 0x7F to FACGUARD as in the unmodified Applesoft. Constructing the FPCOMP routine in this fashion allows this routine to compare the X-register to other values. I also modified the final few instructions of FPCOMP in order to provide the extra space for X-register initialization, yet these replacement instructions provide the same result to FPCOMP.

The FP2INT routine at 0xEBF2 follows FPCOMP. The FP2INT routine quickly converts the value in the FAC floating-point register to an integer value by shifting FACMANT right with sign extension until all of the fractional bits have been shifted out. The FP2INT routine always assumes that FACEXP is less than thirty-two, otherwise FP2INT utilizes the two's complement of FACMANT before any shifting begins. Directly following the FP2INT routine is the Applesoft INT statement at 0xEC23 and INT converts the FAC floating-point register into a 16-bit integer and then refloats that integer into a floating-point value using floating-point normalization. It seems that a faster approach would be to simply clear the lower fractional bits in FACMANT to zero. INT processing also assumes that FACEXP is less than thirty-two. The INT routine is still on my list of routines for further study in order to implement a clear means to accelerate this routine. There are ties to the exponential and power routines that must be considered if any modifications are made to the current implementation. The CLRMANT routine at 0xEC40 follows INT processing and CLRMANT clears FACMANT to 0. This short and simple routine is followed by the very lengthy and complex routine GETINT at 0xEC4A. The GETINT routine is 172 bytes in length and it is comprised of six processing units that converts a string variable into a floating-point value in the FAC floating-point register. GETINT is called by INPTLIST, FRMELMNT, and VAL with the first character of the string variable already scanned and in the A-register. GETINT clears its working area in page-zero from 0x99 to 0xA3 in order to evaluate a string variable for its numeric value including +, -, ., and E, as well as + and - that might be associated with E if scientific notation is encountered in this string variable. The GETINT routine performs as intended unless the string variable resides at the top of the Character String Pool. In that situation, GETINT may not encounter a NULL character until somewhere, hopefully, in the 0xC0 page. This situation will always be problematic for GETINT whenever HIMEM is initialized to 0xC000. Fortunately, DOS 4.5.08H utilizes main memory from 0xBE00 to 0xBFFF, and HIMEM is always initialized to 0xBE00. The GETINT routine uses CHRGET, DIVFAC10, MULFAC10, NEGFAC, and ADD2FAC for its external resources in order to perform this string variable to floating-point value conversion. The ADD2FAC routine at 0xECF6 follows GETINT. Here

is, yet again, another instance where the Applesoft language developers inserted a complete routine in the middle of another complete routine. The ADD2FAC routine is used by LN and GETINT, yet ADD2FAC is found at 0xECD5 in the unmodified Applesoft. If there was some advantage for incorporating a routine in the middle of another routine, I am all ears. So far, I have not heard one rational reason for this programming style, and I am using the programming description generously. I moved ADD2FAC out of and after GETINT processing. ADD2FAC copies the floating-point value in the FAC floating-point register into the ARG floating-point register, floats the value that is in the A-register putting that value into the FAC floating-point register, and calls ADD to add the two registers leaving their sum in the FAC floating-point register.

I did a substantial reorganization of the floating-point values and the routines that come after ADD2FAC and come before FPOUT. The floating-point values are only used by FPOUT, so they have no business being positioned anywhere in Applesoft except where I have placed them just before FPOUT. The PRTMSG19 routine, the LINEPRT routine, and MSG19 can be all nicely positioned immediately following ADD2FAC. I wanted to insert another carriage return in the PRLINUM routine just before the RESTART routine. DOS 4.5.08H is programmed with my sensibilities to insert an extra carriage return before the next DOS command line. I have become quite accustomed to how nice and uncluttered this makes the presentation of the entire monitor display. And, I want to extend that presentation style to Applesoft. Some of the PRLINUM processing is done by PRTMSG19 at 0xED0A in order to provide the necessary space for that additional usage of PRTCR. I have modified PRTMSG19 to complete PRLINUM processing and then print MSG19 as it always has and provide CURLIN, the current line number, in (X/A) to LINEPRT at 0xED18. LINEPRT floats the value received in (X/A) and it uses FPOUT by means of LINEOUT to print that floating-point number. Because I have the space here at 0xED25, I moved MSG19 to this location in order to provide additional space for the other error messages and for GTFORPNT. The floating-point values that are needed for FPOUT processing begin at 0xED2A and FPOUT processing begins at its normal location at 0xED34.

I have already made it abundantly clear why it was unnecessary for the Applesoft language developers to differentiate between STR statement processing and FPOUT processing. The modified Applesoft does not differentiate between these two processing routines for the utilization of the STACK. FPOUT can, therefore, make so much better use of its substantial processing space of 332 bytes. This is one of the more exciting routines in all of Applesoft, and I thoroughly enjoyed tearing this routine apart and finding ways to not only accelerate its processing, but to add more capabilities to its processing. FPOUT makes extensive use of the registers and I found it foolish to maintain the Y-register as the STACK pointer since numeric values as well as other ASCII characters are saved onto the STACK as the processing unfolds. I devised a subroutine to manage a true STACK pointer while items are saved onto the STACK. Applesoft is designed to provide up to nine base-10 digits for display. The Applesoft floating-point variable having eight bits for its exponent and thirty-two bits for its mantissa is designed intentionally to provide this number of accurate digits. The FPOUT routine must ensure that it can present those nine digits without mistake by modifying the value in the FAC floating-point register to reside in a specific numerical range using the MULFAC10 and DIVFAC10 routines. FPOUT processing can then extract the necessary values to place the sign character if necessary, the number of whole digits, the decimal point, the number of decimal digits, and if scientific notation is required, an E character, a sign character, and an exponent value. Once the FAC floating-point register is normalized with an exponent that is equal to 0xA0, those various counters for the number of whole digits and the number of decimal digits can be initialized. FPOUT processing utilizes an incredibly complex processing loop to determine each and every digit of the floating-point value and in which iteration to place the decimal point if at all. Certainly, if the number does not contain a fractional component, a decimal point should not appear in the final integer display. The FPOUT routine in the unmodified Applesoft does not include a 0 placeholder character to the left of the decimal point as in .1234. I find this annoying and very unattractive when presenting numerical information. Is it so difficult to present such a value in the format of 0.1234? During which numerical iteration should this 0 be written? What are the conditions? I found

an easy solution once I understood all of the mechanisms that FPOUT utilizes for making all of its other decisions. What I found most interesting was how FPOUT ping-pongs between subtracting a base-10 value in order to ascertain the first numerical value to adding a base-10 value in order to ascertain the second numerical value and so forth. And, by design, the FPDECTBL that FPOUT needs for this iteration loop contains all nine values from 100 million to 1 in order to present up to nine digits for display. Once FPOUT has written the ASCII data NULL termination byte, FPOUT exits with the address of the STACK in (A/Y).

The Applesoft SQR statement at 0xEE8D follows the FPDECTBL table that is used by FPOUT. FPDECTBL begins at 0xEE5C and it ends at 0xEE7F. So, there are thirteen bytes from 0xEE80 to 0xEE8D for use by SQR processing. The unmodified Applesoft leverages off of the power operator function and uses a value of 0.5 in order to calculate the square root of the input argument to the SQR statement. According to the *Basic Programming Reference Manual for Applesoft II*, the Applesoft SQR statement processing is a *special implementation* that is said to execute more quickly than $x^{0.5}$. However, according to my analysis of SQR processing, the *Reference Manual* could not be more incorrect: the FAC floating-point register is initialized with the value of 0.5 and it falls directly into the power operator function to complete its processing. This processing is *exactly* like $x^{0.5}$. As will be presented in the next section, the power operator function utilizes the exponentiation of the product from the natural logarithm of the input argument and the value of the power argument. The power operator function depends on processing two Taylor series expansion routines and this processing is very expensive in terms of execution time. There does exist an alternative method to calculate the square root of any positive floating-point number. Isaac Newton, the father of the Industrial Revolution, was the first person to recognize and to develop an easy to implement root-finding algorithm which produces successively better approximations to the roots of a real-valued function. When Newton's method is applied to finding the square root of a positive floating-point number, it turns out that this specific algorithm is centuries old dating at least to the ancient Babylonians.

$$x_{n+1} = \frac{1}{2} \left(x_n + \frac{A}{x_n} \right)$$

In this equation, A is given as the input argument to the SQR expression and x_n is given as the first guess or initial value. Selecting an appropriate initial value is the most problematic decision that must be made in order to significantly reduce the number of iterations to the minimum number possible. In all of my reading on the Newton-Raphson iteration method, I found no useful recommendations for the initial value. Even an acquaintance of mine who has a PhD in mathematics could not provide a useful recommendation except to say that *any positive value that is not zero would work just fine for the initial value*. After I examined a few floating-point numbers, both fractional numbers and numbers that are greater than one, I observed that the exponent of its square root value is typically around half of its given value after EXPBIAS is removed. And, that is precisely why the Applesoft language developers utilized the natural logarithm and the exponentiation routines. If the natural logarithm of a number is multiplied by 2, for example, when e is raised to that power, the original number is now squared. Because I had sufficient space within Applesoft for the design and the implementation of an algorithm that utilizes the Newton-Raphson iteration method, my new SQR algorithm begins with the utilization of the SIGNCHK routine. If the input argument to the SQR function is zero, there is nothing further to do. If the input argument to the SQR function is negative, I issue the Illegal Quantity error message and I terminate further processing. Otherwise, I copy the FAC floating-point register to the T1 floating-point register and process FACEXP which is already in the A-register. This simple processing provides me with a very good first approximation for x_n . Because this iterative method is so efficient in calculating a square root value, I setup a loop counter for a maximum of seven iterations. I save the first approximation as well as every successive approximation to the T3 floating-point register. The T3 floating-point register is the only other floating-point register besides the FAC and the ARG floating point registers that includes a guard byte. The T3 floating-point register and its guard byte

is my design; it is not the design of the Applesoft language developers. All of the T3 register copy routines include the copy of the appropriate guard byte. Of course, utilizing the T3 floating-point register with the DIV routine and followed by the ADD routine provides far more arithmetic accuracy. Furthermore, I take advantage of my modified FPCOMP routine for a more accurate comparison of the new approximation in the FAC floating-point register with the previous approximation that was saved in the T3 floating-point register. If FPCOMP finds that the two floating-point values are equal or if the iteration counter expires, SQR processing exits into the RNDUP routine in order to return the finalized floating-point square root value.

Transcendental Arithmetic Operations

Applesoft is only generally divided into its collection of statements and routines that perform transcendental arithmetic operations. These arithmetic operations include power, exponential, random number, cosine, sine, tangent, and arctangent calculations. The following is a collection of Applesoft statements and routines that perform transcendental arithmetic operations.

The Applesoft \wedge statement at 0xEE97 is the power operator function in Applesoft and it directly follows SQR processing. Originally, SQR processing utilized the Applesoft power function and SQR processing simply fell into the power function with the FAC floating-point register initialized with the parameter value of 0.5. The power function utilizes the following expression in calculating its power function value into the FAC floating-point register:

$$\text{FAC} = \text{EXP}[\text{LN}(\text{ARG}) * \text{FAC}]$$

The floating-point value that is being raised to some power is found in the ARG floating-point register and the floating-point value that is equivalent to the power is found in the FAC floating-point register when the power expression is evaluated. Both FACEXP and ARGEXP are tested for zero and processing terminates if either exponent is zero. The FAC floating-point register is copied to the T3 floating-point register and ARGSIGN determines whether to process the FAC floating-point register as an integer and compare it to the T3 floating-point register or simply note that the value in the ARG floating-point register is positive. If the ARG floating-point register is positive, the FAC floating-point register may contain either a positive or a negative floating-point value. If the ARG floating-point register is negative, the FAC floating-point register may only contain any positive or negative *integer* value. In this case, if the FAC floating-point register does contain a floating-point value, the Illegal Quantity error message is issued and further processing terminates. The ARG floating-point register is copied to the FAC floating-point register and its natural logarithm is calculated. That natural logarithm in the FAC floating-point register and the T3 floating-point register are multiplied, and the exponential of their product is calculated. Both the T3 floating-point register and the FAC floating-point register provide a full 40-bit multiplicand and multiplier, respectively. If the sign that was initially determined is negative, the power function falls into the NEGFAC routine, otherwise, the power function exits with its power function value in the FAC floating-point register. The Applesoft > statement at 0xEED0 is the greater than operator function in Applesoft or the NEGFAC routine, and it directly follows the \wedge processing. The NEGFAC routine exits if FACEXP is zero, otherwise the routine exclusively-ORs FACSIGN with 0xFF.

The eight Taylor series exponential polynomials at 0xEEEE0 follow the NEGFAC routine. These polynomials are used to service and conclude exponential processing. Following the eight exponential polynomials is the Applesoft EXP statement at 0xEF09. EXP processing calculates e to the power of the input value that

currently resides in the FAC floating-point register when the EXP expression is evaluated. EXP processing exits with its result in the FAC floating-point register. EXP processing 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 $\ln(e)/\ln(2) = 1/\ln(2) = 1.442695041$. The FAC floating-point register is processed by RNDUP and then copied to the ARG floating-point register. FACEXP must be less than 0x88 in order to continue processing, otherwise an Overflow error message is generated and processing terminates. The FAC floating-point register is converted into an integer in order to generate a new exponent for the final fractional value. I have absolutely no idea why this routine copies the ARG floating-point register into the FAC floating-point register using exactly the same procedure as that used in the COPY2F routine, subtracts the two registers, and then negates the FAC floating-point register. Only an adolescent, perhaps a developer, did not realize that $-(\text{ARG} - \text{FAC}) = (\text{FAC} - \text{ARG})$. This bit of nonsense is deleted in the modified Applesoft and only the registers as they are found are subtracted. A modified Taylor series expansion is utilized in order to process the value that is obtained from the floating-point register difference. The saved exponent is added to the final value that is obtained after 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. The Applesoft language developers modified the exponential polynomials as shown in Table 9 from their theoretical 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 polynomial values that are shown in Table 9 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 - 1	0x00	0x07	7	0x07	7
$(\ln(2)^7)/7! * x^7$	0x01	0x71 34583E56	2.14987637E-05	0x70 7FE5FE2B	1.525273380E-05
$(\ln(2)^6)/6! * x^6$	0x06	0x74 167EB31B	1.43523140E-04	0x74 2184897B	1.540353039E-04
$(\ln(2)^5)/5! * x^5$	0x0B	0x77 2FEEE385	1.34226348E-03	0x77 2EC3FF3E	1.333355815E-03
$(\ln(2)^4)/4! * x^4$	0x10	0x7A 1D841C2A	9.61401701E-03	0x7A 1D955B7E	9.618129108E-03
$(\ln(2)^3)/3! * x^3$	0x15	0x7C 6359580A	0.0555051269	0x7C 635846B8	0.05550410867
$(\ln(2)^2)/2! * x^2$	0x1A	0x7E 75FDE7C6	0.240226385	0x7E 75FDEFFD	0.2402265070
$\ln(2)/1! * x$	0x1F	0x80 31721810	0.693147186	0x80 317217F8	0.6931471806
1.0	0x24	0x81 00000000	1.0	0x81 00000000	1.0

Table 9. Applesoft Exponential Function Polynomials

Two short Applesoft statement routines directly follow the EXP statement routine which includes the Applesoft LOG statement at 0xEF3E and the Applesoft PI statement at 0xEF48, both made possible by the smart elimination of the register copy and the register negation routines in EXP processing. The LOG routine converts the natural logarithm of the input argument that is evaluated from the LOG expression to the base-10 logarithm simply by multiplying the value of the natural logarithm and the value of base-10 LOG (e). The PI routine uses LOADFAC in order to initialize the FAC floating-point register with the floating-point

parameter value of PI, initialize FACSIGN with 0, and initialize FACGUARD with FPIGUARD, or the last eight bits of the 40-bit PI mantissa. Having the value of PI readily available as an Applesoft statement prevents having to fumble around for that value or some close approximation to the value of PI and have it already in floating-point format with a full 40-bit mantissa. How excellent is that?

Applesoft polynomial processing comes in two flavors: sequential or normal polynomial processing as in exponential polynomial processing and odd polynomial processing as in natural logarithm polynomial processing. Odd polynomial processing depends on normal polynomial processing after odd polynomial processing has prepared the FAC floating-point register with its x^2 value rather than with its x value. Because there is sufficient space in the modified Applesoft, the sine function can use the POLYSIN entrance at 0xEF57 to initialize (A/Y) for COEFPTR at 0xEF5B rather than during sine processing. Otherwise, POLYPROC is the general entrance at 0xEF5B for odd polynomial processing. POLYPROC uses COPYF2T1 in order to copy the FAC floating-point register to the T1 floating-point register, the registers are multiplied, and the POLYNOM routine is used to process x^2 with the polynomials whose address is already found at COEFPTR. Once POLYNOM processing is complete, the POLYPROC routine finishes by multiplying the FAC floating-point register with the value that was saved in the T1 floating-point register. The POLYNOM routine at 0xEF71 directly follows POLYPROC and POLYNOM is normal polynomial processing. This routine uses the COPYF2T2 routine in order to save the FAC floating-point register to the T2 floating-point register. I modified the general purpose COPYFAC routine to always load the X-register with the value in FACGUARD so that FACGUARD is readily available whenever it might be required. The POLYNOM routine makes use of this COPYFAC feature and it saves the X-register to T2GUARD, a new variable that I added to the modified Applesoft. POLYNOM extracts the number of coefficients from COEFPTR, increments COEFPTR, and points (A/Y) to the first coefficient in the list of polynomials. Coefficient processing is a very dense processing loop that begins by loading the ARG floating-point register from either a coefficient address or from a register address into (A/Y) and saves the value that is in the X-register to ARGGUARD which is new to POLYNOM processing. In effect, this is my attempt to make use of 40-bit mantissas in very dense processing loops such as POLYNOM processing. The FAC and ARG floating-point registers are multiplied, COEFPTR is modified to point to the next coefficient, the new coefficient is added to the FAC floating-point register, (A/Y) now points to the T2 floating-point register, the X-register is loaded from T2GUARD, the number of coefficients is decremented, and if the number of coefficients is not zero, processing continues at the beginning of the POLYNOM processing loop. No matter how many times I review this dense processing loop, I am filled with awe at how incredible the Applesoft language developers utilized the fundamental Applesoft floating-point arithmetic operations to compute a beautiful Taylor series expansion.

I can only chuckle at the level of disgust that I have for the Applesoft language developers when it comes to the routine that directly follows their brilliant POLYPROC and POLYNOM routines. Surely, did the same developers produce all of the incredible Taylor series expansion routines also produce the Applesoft RND statement at 0xEFAE? I am not sure if that is even possible. The two integer values at 0xEFA6 and 0xEFAA that precede the RND statement have given previous Applesoft reviewers issues in understanding why these variables are not five bytes in length. Are they not floating-point variables? They are used as floating-point variables by floating-point arithmetic operations. What is going on in the RND routine?

The random number generator that is utilized in the unmodified Applesoft is faulty, and an article *RND is Fatally Flawed* was submitted to Call A.P.P.L.E. and printed in the January, 1983, issue on pages 29-34. This article also presents an alternative routine. Applesoft initialization only copies the first four bytes of the five-byte variable that is utilized as the *seed* for the next random number iteration. This *seed* is utilized in the random number generator as a floating-point number rather than as an integer. The random number generator is conflicted in that it attempts to implement a Linear Congruential Generator, or LCG equation using floating-point variables. The Applesoft generator even resorts to byte swapping the first and the third

bytes of the final mantissa, a technique that is said to be of last resort even for a lousy implementation of a random number generator. The two four-byte variables that come before the RND routine are used as floating-point variables. Applesoft floating-point variables must be five bytes in size with one byte for the exponent and four bytes for the mantissa. The assumed exponent in these four-byte variables, 0x98 for the first and used as a multiplier and 0x68 for the second and used as an addend, differ by 0x30. Any exponent difference that is greater than 0x20 **cannot** be accommodated by an Applesoft normalization routine. Are these two numbers indeed floating-point variables or are they truly 32-bit integers? What Mr. Sander-Cederlof does not explain in his article *Random Numbers for Applesoft* in the May, 1984, magazine *Apple Assembly Line*, is *why* the Applesoft RND routine fails to generate more than a few thousand random numbers before the full period of its sequence is reached. He does offer three useable routines that are *better* algorithms according to Donald Knuth in his series of books *The Art of Computer Programming*. In Volume 2 *Seminumerical Algorithms*, pages 155 to 157, Knuth discusses using a standard LCG in order to easily generate random numbers. The Applesoft RND routine is written as if it is trying to implement an LCG using floating-point variables. The equation for the standard LCG is given as follows:

$$X_{n+1} = (X_n * A + C) \bmod(M)$$

An LCG is an algorithm that yields a sequence of pseudo-randomized numbers that are calculated with a discontinuous piecewise linear equation. The method represents one of the oldest and best-known pseudo-random number generator algorithms. The values for A, C, and M are **integer constants**. Historically, poor choices for A have led to ineffective implementations of LCGs. Choosing M to be a power of two such as 2³² often produces a particularly efficient LCG. Correctly choosing the constants A and C will allow a sequence period equal to M. This will occur if and only if 1) M and C are coprime, 2) A-1 is divisible by all prime factors of M, and 3) A-1 is divisible by four if M is divisible by four. Typically, LCGs are fast and require minimal memory. This makes them valuable for simulating multiple independent streams. LCGs are **not** intended, and must never be used for cryptographic applications. In practice, LCGs are not suitable for large-scale Monte Carlo simulations.

Knuth specifies M to be 2³² when A and C are 32-bits in size, so four-byte integer variables are used for A and C in the above equation. Based on the above three rules that Knuth describes in his book, he specifies that A should equal 0x12B9B0A5 and C should equal 0x361962EA. These two values are quite different from the values that are found in the unmodified Applesoft. Applesoft uses 0x9835447A for A and 0x6828B146 for C. Where Applesoft goes terribly wrong in implementing the LCG equation shown above is that Applesoft uses these two variables as floating-point arguments and processes them with floating-point arithmetic operations. Applesoft multiplies the seed at IRAND with its value of A and adds to that product its value of C. Applesoft then implements a modulo 2³² by changing the resulting exponent to 0x80 before it normalizes the floating-point value with the final mantissa. Simply stated, floating-point numerical operations are designed to **preserve** the most significant bits and **discard** the least significant bits during the implementation of those arithmetic operations. This is **not** what is intended for the design of an LCG that requires a modulo. Specifically, a modulo dictates that the **least** significant bits are to be preserved and the most significant bits are to be discarded. A Peasant integer multiply routine will easily provide the necessary computation. Mr. Sander-Cederlof provides his 32-bit integer multiply routine claiming that it is tricky and that it uses a minimum of variable and program space. I do agree that the multiply routine that Mr. Sander-Cederlof presents is vastly tricky, yet it is not extraordinary by any means. I have great respect for Mr. Sander-Cederlof and he has written a vast amount of revolutionary software. However, in this particular instance, the simple Peasant integer multiply routine that I have chosen to use in my random number generator is smaller in size and faster in overall computation.

Every culture throughout history teaches their children the method or the algorithm that that culture uses in order to multiply two integer numbers by hand. Some cultures emphasize learning multiplication tables whereas other cultures emphasize learning how to quickly divide by two and multiply by two. The later method is known as the Peasant integer multiply routine. The multiplier is checked for even or oddness and then it is halved, any remainder is tossed, and the new value is written below. The multiplicand is scratched out if the multiplier is even, then it is doubled, and the new value is written below. All of the retained multiplicand values are added in order to form the product. That is precisely how the Peasant integer multiply routine operates. The multiplier resides in the MULMANT register and it contains the four-byte variable A. The multiplicand resides in ARGMANT and it contains the four-byte seed IRAND. The four-byte variable C is copied into FACMANT which serves as the product register. After the multiplier in MULMANT is shifted right and if the C-flag is set noting an odd number, the multiplicand in ARGMANT is added to the product in FACMANT. Whether an addition occurs or not, the multiplicand is shifted left thus doubling its value. Any MSB bit that is shifted into the C-flag by ARGMANT is discarded. Using four bytes in each of these registers ensure that modulo 2^{32} remains in force throughout the required thirty-two iterations.

My RND routine is engineered somewhat similar to how Mr. Sander-Cederlof designed his RND routine which he linked to the Apple USR function. If a negative integer argument is provided to the RND routine as in `RND(-1234)`, for example, RND saves that value to IRAND as a positive 32-bit integer which will be the seed for the next random number iteration. If a zero argument is provided to RND as in `RND(0)`, RND returns the value that is saved in IRAND as a positive integer value that has a range from zero to $2^{31} - 1$, or `0x00000000` to `0x7FFFFFFF`. If a positive integer argument that is equal to 1 is provided to RND as in `RND(1)`, RND returns a fractional value that has a range from zero to less than 1 which is simply the integer value that is saved in IRAND divided by 2^{32} . Finally, if a positive integer argument that is equal to a value that is greater than 1 is provided to RND as in `RND(192)` or `RND(280)`, for example, RND returns an integer value that has a range from zero to the supplied integer value minus one. My RND routine captures the integer value of the argument that is provided to RND when the RND expression is evaluated, and if that value is greater than zero, that value is saved to the T1 floating-point register as a Range which is a normalized floating-point number. Once the processing of the LCG equation that is shown above is complete, an exponent of `0x80` is stored in FACEXP and that 32-bit product integer is normalized as a floating-point number using `NORMFAC1`. If a Range of 1 is supplied to RND, the normalized floating-point fraction is returned unaltered to the user. Otherwise, that floating-point fraction is multiplied by the value that is stored in the T1 floating-point register using `MULT`, its product is converted to an integer value by `INT`, and the result is returned to the user as a random number integer value.

The Applesoft COS statement at `0xEFEA` takes the value that resides in the FAC floating-point register, the value in radians that is derived from evaluating the COS expression, and adds to it $\pi/2$. COS processing then falls directly into the Applesoft SIN statement at `0xEFF1`. The Applesoft language developers are not implementing a trigonometric identity between COS and SIN, but Applesoft computes $\text{COS}(x) = \text{SIN}(x + \pi/2)$ simply to extract the sign of the generated numerical value since the COS function lags the SIN function by $\pi/2$. SIN processing spends over 60 bytes of processing instructions transforming the input argument to reside entirely in Quadrant 1 while taking note of its sign as if the input argument actually resides in its intended quadrant. I have no doubt that there are far easier transformation methods, but the floating-point parameters that are required by this processing already exist in Applesoft. Once quadrant transformation is complete, the input argument is processed by a Taylor series of SIN polynomials. The six pre-calculated polynomials in the unmodified Applesoft are modified from the normal Taylor series polynomials, yet these polynomials do yield precise values particularly for angles that are not near the limits of this function, that is, near zero and near $\pi/2$. The Taylor series that is utilized by the Applesoft SIN statement 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 six polynomials sufficient for Applesoft to provide a minimum of nine digits of accuracy for the SIN statement, for the COS statement, and for the TAN statement where both COS and TAN processing depend on SIN processing?

Polynomial	Index	Applesoft Value	Base-10 Value	True Value	Base-10 Value
Entries - 1	0x00	0x05	5	0x05	5
$-(2\pi)^{11}/11! * x^{11}$	0x01	0x84 E61A2D1B	-14.3813907	0x84 F183A7EF	-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 99696673	-76.70585975
$(2\pi)^5/5! * x^5$	0x10	0x87 2335DFE1	81.6052237	0x87 2335E33C	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 10. Applesoft Sine Function Polynomials

Polynomial	Index	Real Value	Base-10 Value
Entries - 1	0x00	0x0A	10
$(2\pi)^{21}/21! * x^{21}$	0x01	0x77 143B8107	0.001130924
$(2\pi)^{19}/19! * x^{19}$	0x06	0x7A C5202109	0.012031586
$(2\pi)^{17}/17! * x^{17}$	0x0B	0x7D 55761958	0.104229162
$-(2\pi)^{15}/15! * x^{15}$	0x10	0x80 B7D6DCF9	-0.718122302
$(2\pi)^{13}/13! * x^{13}$	0x15	0x82 747A1A68	3.819952585
$-(2\pi)^{11}/11! * x^{11}$	0x1A	0x84 F183A7EF	-15.09464258
$(2\pi)^9/9! * x^9$	0x1F	0x86 283C1A44	42.05869395
$-(2\pi)^7/7! * x^7$	0x24	0x87 99696673	-76.70585975
$(2\pi)^5/5! * x^5$	0x29	0x87 2335E33C	81.60524928
$-(2\pi)^3/3! * x^3$	0x2E	0x86 A55DE731	-41.34170224
$(2\pi) * x$	0x33	0x83 490FDAA2	6.283185307

Table 11. Expanded Applesoft Sine Function Polynomials

Table 10 lists the six pre-calculated polynomials that are used for the unmodified Applesoft SIN statement. Table 10 also includes the theoretical values for these six pre-calculated polynomials which exposes a degree of mathematical manipulation to these SIN polynomials. I determined that if the theoretical pre-calculated values for these six polynomials are utilized in Applesoft instead, sufficient calculation differences are obtained from the Applesoft SIN function that are *not* trivial. There are ten unreferenced bytes at 0xF094 that can be eliminated. These bytes, a little example of narcissism, when exclusively-ORed

with 0x87 produce the unusable backward ASCII string MICROSOFT! Also, moving the initialization of (A/Y) to POLYSIN and modifying SIN processing to support TAN processing provide sufficient space to add six more pre-calculated SIN polynomials as shown in Table 11, and a modified Applesoft image can be generated. I have yet to find any difference in the output of the unmodified Applesoft versus the modified Applesoft when eleven pre-calculated polynomials are used for SIN processing. I have observed that the five pre-calculated polynomials that the Applesoft language developers mathematically modified for the unmodified Applesoft produce the same results as if eleven accurately pre-calculated polynomials are utilized for SIN processing. 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 SIN function using the eleven accurately pre-calculated polynomials that are shown in Table 11. At this time, sufficient space is currently available for those additional pre-calculated polynomials. I will, of course, relinquish those 30 bytes of space when and if I absolutely require other functionality.

The Applesoft TAN statement at 0xF03A utilizes SIN processing twice since $\text{TAN}(x) = \text{SIN}(x) / \text{COS}(x)$. The SIN function has been designed to provide the necessary signal to the TAN function when the input argument resides in another quadrant other than Quadrant 1. This signal provides proper sign management for the final value. How SIN implements this signal is inefficient, it forces SIN to manage the STACK, and it forces TAN to enter SIN processing indirectly by means of a weird JSR/JMP construction. I was able to unravel this entire programming mess simply by introducing a page-zero value called SIGNFLG that uses the unused fifth random number seed byte at 0xCD. These modifications have reduced both SIN and TAN processing and have accelerated both routines. The TAN routine saves the 40-bit SIN mantissa value in the T3 floating-point register, obtains the 40-bit COS mantissa value in the FAC floating-point register, copies the T3 floating-point register to the ARG floating-point register, and divides those two registers while utilizing both FACGUARD and ARGGUARD. TAN utilizes the most accurate floating-point arithmetic operations that are possible in the modified Applesoft, and it maintains 40-bit mantissas throughout the POLYPROC and POLYNOM polynomial processing.

The Applesoft ATAN statement at 0xF09E directly follows the eleven SIN polynomials which begin at 0xF066 and end at 0xF09D. In order to compute the arctangent, the input argument after evaluating the ATAN expression must be folded into the range of $[-1,1]$ in order to utilize a Taylor series expansion. If the argument is greater than 1, its reciprocal is used and noted along with its sign, thus reducing its effective range to $[0,1]$. I did make a slight modification to the ATAN routine to help accelerate its processing. This Taylor series expansion converges slowly, particularly for an argument that is close to one. Hence, ATAN polynomial expansion is very inefficient. The final output value of the ATAN routine is always in radians. The Taylor series expansion for the ATAN function for any input value x is given as follows:

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

The final value is subtracted from 2π after Taylor series expansion if the input argument is inverted. If the input argument is negative, the final value is complimented by NEGFACTOR. As stated above, this Taylor series expansion is very slow to converge. Isaac Newton suggested a means to accelerate this convergence that was later published by Leonhard Euler. The Applesoft language developers modified the polynomials that are shown in Table 12 from their theoretical values. These polynomials begin at 0xF0CC, they end at 0xF108, and they grow substantially smaller as the denominator increases in value. I have found that the sample test data that I utilized in order to compare the angle values that the Applesoft ATAN routine generates versus the angle values that a modern day computer 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 12 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 - 1	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 4CCCC91C7	0.19999912	0x7E 4CCCCCCD	0.2
$-(1/3)*x^3$	0x33	0x7F AAAAAA13	-0.333333316	0x7F AAAAAAAB	-0.333333333
$1.0*x$	0x38	0x81 00000000	1.0	0x81 00000000	1.0

Table 12. Applesoft Arctangent Function Polynomials

Applesoft Initialization & Miscellaneous Statements

Applesoft is only generally divided into its collection of statements and routines that manage the initialization of Applesoft. This section of Applesoft also includes several miscellaneous statements. The following is a collection of Applesoft statements and routines that initialize Applesoft and conclude Applesoft I that was purchased from Microsoft.

The architecture of the 6502-microprocessor, and later the 65C02-microprocessor, addresses three vectors at the very top of its 16-bit address capability. These vectors include the non-maskable interrupt vector, or NMI vector at 0xFFFFA:0xFFFFB, the RESET vector at 0xFFFFC:0xFFFFD, and the maskable interrupt vector, or IRQ/BRK vector at 0xFFFFE:0xFFFFF. A non-maskable interrupt cannot be disabled or ignored using either processor instructions or software masks. On the other hand, a maskable interrupt can be disabled or ignored and then re-enabled. The 6502 or the 65C02 instruction set contains the SEI instruction to disable a maskable interrupt and the CLI instruction is used to enable a maskable interrupt. When the Apple II computer is powered on, the 6502-microprocessor automatically loads the RESET vector at 0xFFFFC:0xFFFFD into the program counter and it continues to fetch instructions beginning from the address that is stored at that location. The ROM RESET handler address that is stored in the RESET vector is memory address 0xFA62. After the ROM RESET handler has initialized the annunciators, the window specifications, its CSWL and KSWL interface pointers, and XMODE for the Apple II/e, the handler directly enters the NEWMON routine at memory address 0xFA81. After ringing the bell at 0xFF3A, the NEWMON routine calculates its own PWRSTATE value and compares that calculation to the value it finds at 0x3F4. If the comparison fails, the hardware is powering up and the NEWMON routine branches to the PWRUP routine at memory address 0xFAA6. Otherwise, the routine falls into the NEWMON2 routine that initializes AUTORSET at 0x3F2 with a non-zero

value of 0x03 and jumps directly to BASIC which is at 0xE000. The 0xE000 location as well as the 0xE003 location for a jump directly to the Applesoft RESTART routine, occur in the middle of the Applesoft PTRGET routine. The 0xE000 location contains a jump directly to the COLDSTRT routine at 0xF125 in the modified Applesoft. The COLDSTRT entry address is 0xF128 in the unmodified Applesoft. That three-byte difference comes from accelerating ATAN processing and removing the fifth byte of the random number generator *seed* which is unnecessary.

Prior to the COLDSTRT routine and just after the ATAN polynomials are the infamous CHRGET and CHRGOT routines as well as the four-byte random number generator *seed*. There have been reports that Cornelis Bongers devised a shorter and accelerated CHRGOT routine. Since this routine utilizes 24 bytes of page-zero memory at 0xB1:C8, that would have been an awesome accomplishment. The CHRGOT routine absorbs the SPACE character 0x20, it clears the C-flag for a numeric value 0x30:39, and it sets the C-flag for all other ASCII values. What is more interesting is that CHRGOT sets the Z-flag if that input character happens to be a colon :. The random number generator *seed* resides at 0xC9:CC following CHRGOT. Many of the initializations that are performed by the COLDSTRT routine are ridiculous and they serve no purpose or benefit. I have removed all of these particular initializations in the modified Applesoft. The Applesoft COLDSTRT routine sets the Direct Mode flag and the STACK pointer, creates four jump vectors, copies CHRGET, CHRGOT, and the random number generator *seed* to page-zero, initializes some flags and a pointer, and determines the end of RAM memory. The end of RAM memory is the location where MEMSIZE and FRETOP are initialized. Memory location 0x0800 is initialized to zero and PRGTAB is initialized to 0x0801, the beginning address for an Applesoft program. DOS 4.5.08H provides the capability to load and to run an Applesoft program at any selected address by initializing PRGTAB to an address that is greater than or equal to 0x0801. I did add the initialization of HRSCALE to COLDSTRT processing because there was sufficient space. The Applesoft COLDSTRT routine calls CKSTRSIZ in order to check the amount of memory between arrays and strings, which is of rather dubious value since FRETOP was just initialized to its maximum value possible. The call to SCRTCH initializes RUNFLAG, VARTAB, and PRGEND and falls into SETPTRS as previously described. Finally, the Applesoft COLDSTRT routine completes the initialization of two USER vectors and jumps directly to the RESTART routine at 0xD43C. I have used the reclaimed space from the Applesoft COLDSTRT routine from 0xF1B1 to F1D4 for additional FRMSTAK4 processing at 0xF1B1, the new Applesoft COPYF2T3 routine at 0xF1BA, a new Applesoft routine that increments the coefficient pointer for polynomial processing at 0xF1C5, and a new Applesoft routine that initializes the MULMANT register to zero at 0xF1CC. The COPYF2T3 routine first copies FACGUARD to T3GUARD, a new page-zero variable, and then this routine initializes the X-register and the Y-register with the address of the T3 floating-point register. The COPYF2T3 routine uses the COPYFAC2 entry point in order to copy the floating-point number from the FAC floating-point register into the T3 floating-point register.

The Applesoft CALL statement at 0xF1D5 follows the CLEARMUL routine that clears the MULMANT floating-point register to zero for the MULT routine. CALL processing evaluates its input expression and converts that value into a 16-bit integer value that is saved in LINNUM. CALL processing simply jumps indirectly to that LINNUM address. The Applesoft IN statement at 0xF1DE follows CALL processing. IN processing evaluates its input expression and converts that value into an 8-bit integer value and leaves that value in the FAC floating-point register. The CONVINT routine copies the least significant byte of FACMANT into the X-register and returns that value to IN. IN copies the value that is in the X-register into the A-register so that that value can be utilized by the INPORT routine at 0xFE8B in the ROM Monitor. The Applesoft PR statement at 0xF1E5 follows IN processing and PR processing evaluates its input expression in exactly the same fashion as IN processing. The value that is in the X-register is copied into the A-register so that that value can be utilized by the OUTPORT routine at 0xFE95 in the ROM Monitor.

Management of LORES and HIRES Graphics

Applesoft is only generally divided into its collection of statements and routines that manage the various LORES and HIRES graphics. These graphic routines include PLOT, POSN, HRPLOT, DRAW, XDRAW, and HLIN. The following is a collection of Applesoft statements and routines that manage the various LORES and HIRES graphics.

The Applesoft PLOTFNS routine at 0xF1EC follows PR processing. The PLOTFNS routine is used by the SCRN(statement, the LINC00R routine, and the Applesoft PLOT statement. This routine evaluates an expression in order to extract the first coordinate value and the second coordinate value. The first coordinate value is saved to FIRST and the second coordinate value is saved to H2 and V2. Each comma separated value is verified to be less than 48, otherwise Applesoft issues the Illegal Quantity error message. The Applesoft LINC00R routine at 0xF209 follows the PLOTFNS routine. The LINC00R routine is used by the Applesoft HLIN statement and by the Applesoft VLIN statement for LORES graphics. The LINC00R routine utilizes the PLOTFNS routine in order to obtain the start and the end screen coordinates, and it swaps those coordinates if the start coordinate is larger than the end coordinate. Next, the LINC00R routine verifies the presence of the Applesoft AT statement, and it continues to evaluate the expression for a third screen coordinate value. The third screen coordinate value is verified to be less than 48, otherwise Applesoft issues the Illegal Quantity error message. The Applesoft PLOT statement at 0xF225 follows the LINC00R routine. The PLOT routine utilizes the PLOTFNS routine to obtain the first and the second coordinate values and verifies that the first coordinate value is less than 40, otherwise Applesoft issues the Illegal Quantity error message. Applesoft PLOT utilizes the services of ROM Monitor PLOT at 0xF800. The Applesoft HLIN statement at 0xF232 follows PLOT processing. HLIN utilizes LINC00R to evaluate its expression for three coordinate values and verifies that the second coordinate value is less than 40, otherwise Applesoft issues the Illegal Quantity error message. HLIN also utilizes the services of the HLINE routine at 0xF819 in the ROM Monitor. The Applesoft VLIN statement at 0xF241 follows HLIN processing. VLIN utilizes LINC00R to evaluate its expression for three coordinate values and verifies that the third coordinate value is less than 40, otherwise Applesoft issues the Illegal Quantity error message. VLIN utilizes the services of the VLINE routine at 0xF828 in the ROM Monitor.

The Applesoft COLOR statement at 0xF24F follows VLIN processing. COLOR evaluates its input expression and converts that value into an 8-bit integer in the X-register, copies the value into the A-register, and sets the LORES graphic COLOR variable using the SETCOL routine at 0xF864 in the ROM Monitor. The Applesoft VTAB statement at 0xF256 follows COLOR processing. VTAB evaluates its input expression and converts that value into an 8-bit integer in the X-register, decrements that register, and verifies that the final value is less than 24. The VTAB statement accepts an input range of 1:24 and converts that range to 0:23 in order to utilize the TABV routine at 0xFB5B in the ROM Monitor. The Applesoft SPEED statement at 0xF262 follows VTAB processing. SPEED evaluates its input expression and converts that value into an 8-bit integer in the X-register, copies the value into the A-register, exclusively-ORs that value with 0xFF, and saves that final value to SPEEDBYT in the modified Applesoft. In the unmodified Applesoft, the final value is incremented so that the fastest Applesoft speed is 0x01 and the slowest Applesoft speed is 0x00 which is based on the call to WAIT in OUTCHR. I found that adding any unnecessary wait to Applesoft was outrageous and unacceptable. The modified Applesoft accepts the default input speed of 255, converts that to 0, and the modified OUTCHR routine bypasses the call to WAIT when SPEEDBYT is zero. The Applesoft TRACE statement at 0xF26D follows SPEED processing and the Applesoft NOTRACE statement at 0xF26F follows TRACE processing. TRACE sets the C-flag and NOTRACE clears the C-flag so that the C-flag can be used to set or clear the MSB of the TRACEFLG flag.

Applesoft utilizes three statements in order to control how ASCII characters are displayed as they are written to the TEXT screen. The ROM Monitor INVFLG flag controls the MSB, or bit-7 of the ASCII character and the Applesoft FLASHBYT variable controls bit-6 of the ASCII character. The proper setting of these two variables, INVFLG and FLASHBYT, is handled by the Applesoft NORMAL statement at 0xF273 to set bit-7 of INVFLG flag and to clear bit-6 of FLASHBYT. The Applesoft INVERSE statement at 0xF277 clears bit-7 of INVFLG flag and clears bit-6 of FLASHBYT. The Applesoft FLASH statement at 0xF280 clears bit-7 of INVFLG flag and sets bit-6 of FLASHBYT. The value in FLASHBYT is OR'd with all ASCII characters in the OUTCHR routine and the value in INVFLG flag is AND'd with all ASCII characters by the COUT routine that is utilized in the OUTCHR routine.

The Applesoft HIMEM statement at 0xF286 follows FLASH processing. HIMEM evaluates its input expression and converts that value into a 16-bit integer that is saved to LINNUM. LINNUM is compared to STREND in order to verify that the new HIMEM address is above all current variables and arrays, otherwise Applesoft issues the Out of Memory error message. The new HIMEM address is saved to MEMSIZE and FRETOP. The HIMEM routine is another example where it is possible to accelerate the processing by pulling the error message jump out of the routine. Instead of branching around the error message jump, I reversed the branch logic in order to branch to the error message jump only if there exists an error. Perhaps it is a matter of programming style as some might proclaim. I perceive it as the rational option and I let faster throughput code assist and guide my programming style. The Applesoft LOMEM statement at 0xF2A6 follows HIMEM processing. LOMEM evaluates its input expression and converts that value into a 16-bit integer that is saved to LINNUM. LINNUM is compared to MEMSIZE, and if LINNUM is greater, Applesoft issues the Out of Memory error message. LINNUM is then compared to VARTAB, and if LINNUM is smaller, Applesoft issues the Out of Memory error message. Otherwise, LINNUM is saved to VARTAB and LOMEM processing jumps to CLEARC in order to initialize ARYTAB, STREND, and the STACK pointer for this new environment.

The Applesoft ONERR statement at 0xF2CB follows LOMEM processing. ONERR processing verifies that this Applesoft statement is followed by the Applesoft GOTO statement in order for ONERR processing to continue. The current TXTPTR value is saved, the ERRFLG is enabled, and the current CURLIN value is saved. All Applesoft statements and commands that are on the same program line and precede the Applesoft ONERR GOTO statements are processed normally. And, whether the DATSCAN routine is called or not, all Applesoft statements and commands that are on the same program line and come after the Applesoft ONERR GOTO statements are not processed, that is, they are fully ignored and discarded. ONERR processing jumps to DATA2 in order to continue Applesoft program processing. The Applesoft HANDLERR routine at 0xF2E9 follows ONERR processing. HANDLERR is called by PRterr, ASROMERR, or RESPERR whenever the ERRFLG has been enabled. ONERR processing is another example where the ERRFLG is enabled. The RESTART or ASTROMRM routine is one example where the ERRFLG is disabled. The HANDLERR routine saves the REMSTK value, the CURLIN value, and the TEXTPTR value, and restores the saved TXTPTRSV value to TXTPTR and the saved CURLINSV value to CURLIN. The routine then enters the line number that was provided with the Applesoft ONERR GOTO statement and initiates normal Applesoft processing by means of the NEWSTT routine. The Applesoft RESUME statement at 0xF318 follows the HANDLERR routine. RESUME processing restores the saved ERRLIN value to CURLIN, the saved ERRPOS value to TXTPTR, and the saved ERRSTK value to the STACK pointer essentially restoring the line number and the text pointer to the very same line where Applesoft previously detected an error. Once the Applesoft error is corrected by the statements that reside on the ONERR GOTO program line number, the RESUME statement can be issue in order to retry the offending Applesoft program line number. The Applesoft manual on page 82 suggests utilizing a very short routine that will execute at any address and will augment an error-handling routine. This routine is exactly what the RESUME statement accomplishes once the software problem is managed. For example, an ONERR GOTO can be setup to protect a DOS CATALOG command for various volume numbers. Making

volume number a variable that can be managed by Applesoft would certainly allow an Applesoft RESUME statement to repeat the DOS CATALOG command until a valid volume number is utilized that does not cause a DOS error in an Applesoft program.

The Applesoft DEL statement at 0xF331 follows RESUME processing. This statement deletes an Applesoft program line number or a range of Applesoft program line numbers in both the immediate-execution mode and in the deferred-execution mode. However, if the DEL statement is utilized in the deferred-execution mode, the Applesoft program line number or line numbers would be deleted certainly, but the Applesoft program would halt in its execution. There is no workaround such as using the Applesoft CONT statement in order to resume Applesoft processing at the next available statement because that capability was simply not incorporated into the design of DEL or CONT processing. DEL processing expects to evaluate at least one numerical value, otherwise Applesoft issues a Syntax error message. DEL processing removes that specific program line number if it exists, otherwise Applesoft issues a Syntax error message. However, if DEL processing evaluates a second numerical value that is separated from the first numerical value by a comma, a range of program line numbers is removed from an Applesoft program. This range of program line numbers would be from the first numerical value or greater to the second numerical value or lesser. DEL processing tolerates some ignoramus entries such as line 0, negative line numbers, or a range of program line numbers from a larger number to a smaller number. Once DEL processing has removed the target program line numbers, the processing jumps to the beginning of the Applesoft interpreter at AENTER or ASROMRST. DEL processing was simply not designed to resume Applesoft processing at any particular line number that might be just prior to or just after the deleted range of line numbers. I accelerated DEL processing slightly by moving an RTS instruction from 0xF364 and I redirected the branch instruction to an RTS instruction at 0xF38F which is at the end of the routine. Actually, any RTS instruction in the vicinity would have sufficed only if doing so provided other advantages. The Applesoft GR statement at 0xF390 follows DEL processing. GR processing establishes LORES graphics by disabling HIRES graphics, it enables TEXT and mixed graphics, and it utilizes the SETGR routine at 0xFB40 in the ROM Monitor. The SETGR routine disables TEXT and it reiterates enabling TEXT and mixed graphics, it clears the top 40 LORES graphic lines, and it sets WNDTOP to 20 so that only four TEXT lines are displayed at the bottom of the screen. I accelerated GR processing in the modified Applesoft by eliminating the duplicate TEXT and mixed graphics switch that is already found in SETGR processing. The Applesoft TEXT statement at 0xF399 follows GR processing. I modified the TEXT processing to simply jump to the INIT2 routine at 0xFB33 rather than to SETEXT routine at 0xFB39 in the ROM Monitor. The INIT2 routine disables HIRES graphics, enables PAGE1, enables TEXT, and sets WNDTOP to 0 so that all 24 TEXT lines are displayed on the screen.

I have removed the Applesoft STORE statement at 0xF39F and the Applesoft RECALL statements at 0xF3BC in the modified Applesoft since these routines depend on reading and writing data to and from the cassette ports that are no longer useful to the Apple][user. More specifically, I have also removed the routines that the STORE and the RECALL statements depend on such as the TAPEPNT routine at 0xF7BC and the GETARYPT routine at 0xF7D9. However, after I discovered the brilliant software of Egan Ford, I reinstalled the Applesoft RDBYTE routine, the Applesoft LOAD statement, the Applesoft RD2BIT routine, and the Applesoft CXREAD routine in order to support reading his Insta-Disk disk images. The RDBYTE routine, the Applesoft LOAD statement, and the RD2BIT routine have already been described. I placed the CXREAD routine, originally found at 0xC5D1 in the Apple //e CXROM, at 0xF39C and I reinstalled the cassette READ routine in the ROM Monitor at its traditional location of 0xFEFD. I continue to have no further use for the cassette WRITE routine in the ROM Monitor at its former location of 0xFECD, and that location is currently unused and it contains an RTS instruction. The CXREAD routine reads an audio waveform using the RD2BIT routine and its subroutine RDBIT as well as the RDBYTE routine. An audio waveform is comprised of a HEADER, a SYNC, and its DATA as 8-bit bytes. The CXREAD routine contains the timing information for the various waveforms that are utilized in order to differentiate the HEADER, the SYNC, and the DATA fields. The CXREAD

routine may be utilized to LOAD a single Applesoft file into memory or to READ a complete disk image onto an initialized diskette. I have only slightly modified the original CXREAD routine by incorporating a 16 millisecond delay and CHKSUM initialization before the traditional Wozniak routine begins. The usual procedure is to begin playing the AIFF Insta-Disk recording and then issuing the Applesoft LOAD statement on the Apple Command Line. The binary DATA is saved to memory using the address in A1 until A1 reaches the address in A2. Other routines from the collection of Insta-Disk software drivers perform nearly the same function as CXREAD using specific timing information for Insta-Disk data waveforms that can read random data up to 8 KHz or even 9.6 KHz. The CXREAD routine is designed to read a data waveform that contains random data having an equal number of zero bits and one bits at 1333 Hz. It is truly amazing what the Apple][computer is able to accomplish when it is placed into capable hands.

The Applesoft HGR2 statement at 0xF3D8 and the Applesoft HGR statement at 0xF3E2 both follow the CXREAD routine. Even though it is totally unnecessary for me to modify the software of these two statements, I found that I am able to process these two statements faster while adding a more elegant transition from the TEXT display to the respective HIRES graphics display after the screen is cleared. In other words, my graphic routines clear the respective HIRES graphics display before I address any soft switches. The viewer is not shown the HIRES graphics display as its memory pages are being cleared as the display is shown in the unmodified Applesoft. Rather, the viewer is shown the HIRES graphics display *after* its memory pages are fully cleared. To me, this makes a distinct impression when viewing the transition from TEXT display to HIRES graphics display. The Applesoft CLRHIRE routine at 0xF3EC follows HGR processing. Once CLRHIRE has initialized the target memory pages with a value of zero, it enables the HIRES graphics display and it disables the TEXT display. The specific graphic initialization routine enables PAGE2 and disables MIXED graphics for HGR2 processing and it enables PAGE1 and enables MIXED graphics for HGR processing. I designed CLRHIRE in such a way that I have also provided an additional entry point at 0xF3EE called SETHIRE. In order for a user to utilize the Applesoft SETHIRE routine, the A-register must contain the target HIRES graphics display value which is either 0x20 for PAGE1 or 0x40 for PAGE2 and the X-register must contain the value that will be used to initialize the target memory pages. The SETHIRE routine utilizes the COLSHIFT routine in order to invert the memory initialization value for all odd memory locations so that color is displayed homogeneously. It is left to the user to enable the appropriate soft switch for PAGE1 or for PAGE2 after SETHIRE returns to the user.

The Applesoft HPOSN routine at 0xF411 follows SETHIRE processing. HPOSN is used by HRPLOT and by DRAWCMD in order to establish the HIRES cursor position on the graphics screen. This HIRES cursor position requires a 16-bit integer value for the horizontal coordinate and an 8-bit integer value for the vertical coordinate. The established Applesoft protocol requires the horizontal coordinate to use the X-register for the horizontal LSB coordinate value, the Y-register for the horizontal MSB coordinate value, and the A-register for the vertical coordinate value. A series of highly complex, seemingly bizarre page-zero mathematical calculations are employed in order to set GBAS, a 16-bit page-zero address, with the address of the vertical scan line and the Y-register with the horizontal byte number for the HIRES cursor. The value in the Y-register is saved to HRHORZ, and when shifted, determines if COLSHIFT is required to invert COLBITS which contains the value from HRCOLOR. After the value in the Y-register is calculated, that is, until the C-flag is clear, the value that remains in the A-register ranges from 0xF9 to 0xFF. That negative value is used as an index into BITABLE, an array of seven values. The selected array value is saved to COLOR and that value is used as a mask to operate specifically on the target color pixel that is within the selected horizontal byte which is pointed to by the Y-register on the selected vertical scan line whose address resides in GBAS.

F411		10 ;	C--A-reg-- -GBASL-- -GBASH--
F411 86 E0	11	HPOSN stx HRXC00R	
F413 84 E1	12	sty HRXC00R+1	
F415 85 E2	13	sta HRYC00R	; --ABCDEFGH -----
F417	14 ;		
F417 48	15	pha	; --ABCDEFGH -----
F418	16 ;		
F418 29 C0	17	and #\$C0	; --AB000000 -----
F41A 85 26	18	sta GBASL	; --AB000000 AB000000 -----
F41C	19 ;		
F41C 4A	20	lsr	; 0-0AB00000 AB000000 -----
F41D 4A	21	lsr	; 0-00AB0000 AB000000 -----
F41E	22 ;		
F41E 05 26	23	ora GBASL	; 0-ABAB0000 AB000000 -----
F420 85 26	24	sta GBASL	; 0-ABAB0000 ABAB0000 -----
F422	25 ;		
F422 68	26	pla	; 0-ABCDEFGH ABAB0000 -----
F423 85 27	27	sta GBASH	; 0-ABCDEFGH ABAB0000 ABCDEFGH
F425	28 ;		
F425 0A	29	asl	; A-BCDEFGH0 ABAB0000 ABCDEFGH
F426 0A	30	asl	; B-CDEFGH00 ABAB0000 ABCDEFGH
F427	31 ;		
F427 0A	32	asl	; C-DEFGH000 ABAB0000 ABCDEFGH
F428 26 27	33	rol GBASH	; A-DEFGH000 ABAB0000 BCDEFGHC
F42A	34 ;		
F42A 0A	35	asl	; D-EFGH0000 ABAB0000 BCDEFGHC
F42B 26 27	36	rol GBASH	; B-EFGH0000 ABAB0000 CDEFGHCD
F42D	37 ;		
F42D 0A	38	asl	; E-FGH00000 ABAB0000 CDEFGHCD
F42E 66 26	39	ror GBASL	; 0-FGH00000 EABAB000 CDEFGHCD
F430	40 ;		
F430 A5 27	41	lda GBASH	; 0-CDEFGHCD EABAB000 CDEFGHCD
F432 29 1F	42	and #\$1F	; 0-000FGHCD EABAB000 CDEFGHCD
F434	43 ;		
F434 05 E6	44	ora HRPAG	; 0-PPPFHCD EABAB000 CDEFGHCD
F436 85 27	45	sta GBASH	; 0-PPPFHCD EABAB000 PPPFGHCD

Figure 2. Vertical Coordinate Conversion to GBAS

Figure 2 displays the processing in HPOSN and how the address in GBAS is calculated from the vertical coordinate. I have always wondered, if Wozniak had utilized a couple more logic chips, *could* he have reduced the complexity of mapping the HIRES display location to memory location? If he were to achieve that capability, would that have actually altered the HIRES animation routines that I incorporated in my software development for Sierra On-Line? My animation routines calculated GBAS by utilizing lookup tables that mapped vertical scan line directly to memory address. One cannot achieve a faster calculation than using a lookup table. Therefore, however Wozniak mapped the HIRES display location to memory address using hardware does not really matter when drawing or animating objects on the HIRES graphics

display. One is always bound to use the fastest method possible when critical timing loops are totally dependent on how fast one can map a specific display pixel to a specific bit in a specific byte that is within the memory range of a HIRES graphics display in the Apple][computer. I continue to be amazed at Wozniak's innovations.

The Applesoft HRPLOT routine at 0xF457 follows HPOSN processing and this routine must be utilized with all microprocessor registers configured in order to call the Applesoft HPOSN routine. Having the Y-register and GBAS configured accordingly allows this routine to extract the target HIRES byte, mask out all of its bits that conform to the color byte in COLBITS, and then mask the specific target pixel or HIRES bit using the value in COLOR in order to turn that target pixel ON or OFF. That final pixel state is saved back to the screen within the target HIRES byte. This is the general procedure that is also used by the Applesoft DRAWIT routine. The Applesoft XDRAWIT routine modifies this general procedure slightly in order to achieve the ability to *reverse engineer* the previous HIRES drawing. The Applesoft HRMOVLf routine at 0xF465 follows HRPLOT processing. This routine is one of four routines that is used to modify the Y-register and/or the address in GBAS in order to change the HIRES cursor position in one of four directions. The HRMOVLf routine essentially moves the HIRES cursor position to the left by decrementing the Y-register and updating HRHORZ with its new value if necessary, or it updates the value in COLOR by moving the bit mask appropriately to the right, and it updates the color byte value in COLBITS simply by falling into the Applesoft COLSHIFT routine that was previously used by SETHIRES and HPOSN. Perhaps this is the best time to explain how the Apple][hardware draws pixels from the data it finds in memory. The Apple][hardware reads a byte of data from HIRES memory whose address is based on the hardware address mapper logic and that data is clocked into a shift register. If the MSB of that data byte is set, the output of that shift register is delayed by one period of the 14 MHz clock. This delay introduces a shift to the phase angle relative to color burst which changes the perceived color. The shift register is always shifted to the right such that the LSB is the first data bit to be drawn as a pixel. If that data bit is ON, that pixel is displayed ON and the data byte is shifted to the right six more times. The last data bit to be displayed in that data byte is bit six. The address mapper increments and the next data byte is displayed. As soon as the Y-register becomes negative, the register is initialized with the value of 39 and COLOR is initialized with the value of 0xC0 in order to mask bit six, the left-most pixel. The Applesoft COLSHIFT routine at 0xF47E follows HRMOVLf processing. COLSHIFT simply inverts the value in COLBITS if that value is greater than 0x1F and less than 0xE0, otherwise COLSHIFT does not modify the value in COLBITS. The Applesoft HRMOVRT routine at 0xF484 follows COLSHIFT processing. HRMOVRT moves the HIRES cursor position to the right by incrementing the Y-register and updating HRHORZ with its new value if necessary, or it updates the value in COLOR by moving the bit mask appropriately to the left, and it updates the color byte value in COLBITS. As soon as the Y-register becomes equal to 40, the register is initialized with the value of 0 and COLOR is initialized with the value of 0x81 in order to mask bit zero, the right-most pixel. In summary, HRMOVRT shifts the value in COLOR to the left in order to move the pixel cursor position to the right and HRMOVLf shifts the value in COLOR to the right in order to move the pixel cursor position to the left.

It amazes me how little testing Randy Wigginton and Cliff Huston must have done when they designed their XDRAWIT routine and limited this routine to drawing a SHAPE definition that is only white in color no matter what setting is used for HCOLOR= and without regard to the background color. How impressive is that? Without knowing any more of the history of the development of the Applesoft interpreter when the early Apple][computer was released for purchase, I can only surmise that time was of the essence in order to produce a product quickly and without much regard to whether the best choices were made in the design of many of the HIRES routines. Clearly, the DRAW and the XDRAW functions are not thoroughly well designed. When I began my development of *SHAPE Manager*, I realized that the Applesoft XDRAW function provided all of the HIRES drawing capabilities that I needed and were required by *SHAPE Manager* only after I made substantial modifications to Applesoft. Initially, I was very confused as to what capabilities the DRAW

function provides and what capabilities the XDRAW function provides. The DRAW and XDRAW functions have no relationship or interdependencies, and these two Applesoft functions are **not** designed to be used in conjunction with the other. The DRAW function is designed to manipulate the pixels on the HIRES screen in order to place a SHAPE definition which is drawn from a SHAPE table *over* or *on top of* whatever HIRES pixels are currently being displayed. There is **no** mechanism to programmatically remove this SHAPE definition except by drawing another SHAPE definition over the same HIRES pixels that are currently being displayed. The DRAW function does not incorporate any of the old HIRES pixel information with any of the pixel information in the new SHAPE definition. The DRAW function draws colors to the HIRES screen such that the data that is drawn replaces whatever data may previously exist on the HIRES screen. The XDRAW function incorporates the old HIRES pixel information with the new SHAPE definition pixel data such that the new SHAPE definition can be easily removed and the old HIRES pixel information can be restored as it was previously simply by performing another XDRAW with the same SHAPE definition at the same screen location. As with all graphic routines that make use of the exclusive-OR microprocessor instruction, color complements must be taken into consideration when using the XDRAW function. The XDRAW function draws colors to the HIRES screen such that the data that is drawn becomes the complement of whatever data may previously exist on the HIRES screen. The main purpose in using the XDRAW function is to provide a simple way to erase a shape and to easily redraw that same shape or another shape at the same HIRES screen location or at another HIRES screen location without erasing the background data. The DRAW function is far more straightforward to use in many respects. However, shapes that are drawn by the DRAW function cannot easily be programmatically removed from the HIRES screen as easily as those shapes that are drawn by the XDRAW function. All HIRES animation uses XDRAW inspired routines. Both DRAW and XDRAW functions may be used from the Apple Command Line or from within an Applesoft program.

I have heavily modified the draw shape routines such that the new Applesoft DRAWHDR routine at 0xF49C follows HRMOVRT processing and DRAWHDR prefaces the modified Applesoft XDRAWIT and DRAWIT routines. DRAWHDR processes common code from the beginning of the original XDRAW and DRAW routines at 0xF49C and 0xF4B3, respectively, in the unmodified Applesoft. In the modified Applesoft, DRAWHDR processes new instructions then enter either the XDRAWIT routine or the DRAWIT routine. The Applesoft XDRAWIT routine at 0xF4A6 follows DRAWHDR processing. XDRAWIT utilizes COLBITS in order to support color which the original XDRAW routine failed to do. If another object exists at this screen location, XDRAWIT branches to increment a common collision counter HRCOLCNT, otherwise XDRAWIT branches to the common XDRAW/DRAW routine. The Applesoft DRAWIT routine at 0xF4B8 follows XDRAWIT processing. DRAWIT is based on the original DRAW routine and if there exists another object at this screen location, DRAWIT falls into the collision counter HRCOLCNT before entering the common XDRAW/DRAW routine at 0xF4C2. Once the common XDRAW/DRAW routine displays the intended pixel, the common SHAPE processing begins where the C-flag is clear during horizontal processing or the C-flag is set during vertical processing. In other words, every SHAPE command is processed once for its horizontal information and once for its vertical information. If the resulting rotation to the SHAPE results in the C-flag being set, a branch is made to HRMOVLF as previously discussed. Otherwise, SHAPE rotation logic enters the Applesoft HRMOVUP routine at 0xF4D1. If the value in the A-register is negative, a branch is made to the Applesoft HRMOVDN routine at 0xF501. These are the last two routines of four that are used to modify the Y-register and/or the address in GBAS in order to change the HIRES cursor position in one of four directions. HRMOVUP modifies GBAS in order to move up one scan line or to move to the very last scan line whose address is HRPAG plus 0x1FD0. I removed the unnecessary CLC instruction from the top of this routine. The Y-register, the COLOR variable, and COLBITS variable are never modified. HRMOVDN also modifies GBAS in order to move down one scan line or to move to the very first scan line whose address is found in HRPAG. I removed the unnecessary CLC instruction from the top of this routine and I added a true termination at the end of this routine in order to accelerate processing. Once again, the Y-register, the COLOR variable, and the

COLBITS variable are never modified. The BITBYT table values that are utilized by HRMOVUP and HRMOVDN at 0xF52D follow HRMOVDN processing. And, the BITABLE table at 0xF530 follows the BITBYT values. Again, the BITABLE is utilized by the Applesoft HPOSN routine in order to initialize the COLOR variable.

The Applesoft HLIN routine at 0xF53A follows the BITABLE table values. HLIN is only used by HPLOT and it would have been far more practical to include HLIN inline within HPLOT and four bytes would have been saved. However, HLIN can be utilized by an external user to Applesoft in order to draw one HIRES line. As documented in the *DOS 4.5 Volume and File Disk Management System Second Edition*, HLIN is hopelessly flawed. I have always disliked the unsymmetrical look of a HIRES diagonal line when it is drawn either in the horizontal or in the vertical direction ever since I acquired my Apple][+. And this same HLIN routine persists in the Applesoft of the Apple //e unchanged, which is shameful in my opinion. After I analyzed HLIN, I found that the routine does not correctly calculate the delta difference of the horizontal and of the vertical start to end points before drawing the requested line. It is easy to demonstrate this error before and after installing the modified Applesoft or by using an assembly language routine that contains the HLIN routine with and without the necessary modifications. There are two memory locations that require a small code adjustment. The first code adjustment is made at 0xF57A and the second code adjustment is made at 0xF5A5. You will simply be amazed at how *lovely* and *symmetrical* diagonal lines are drawn either from left to right, from right to left, from top to bottom, or from bottom to top. I am literally appalled that the original Applesoft passed any sort of testing and/or code review vis-à-vis how trivial these two modification are and how elegant the results appear to be. The established Applesoft protocol for HLIN requires the horizontal coordinate to use the A-register for the horizontal LSB end coordinate value, the X-register for the horizontal MSB end coordinate value, and the Y-register for the vertical end coordinate value. This protocol is different from the protocol that is used for HPOSN which establishes the start coordinates. I used the rts instruction at 0xF52C for the branch instruction at 0xF59C and I replaced the bvc instruction at 0xF5B0 with a jmp instruction because I have the Applesoft space.

HLIN always draws a HIRES line from the start coordinates that are established by HPOSN or from the end coordinates of a previous call to HLIN to the end coordinates of the current call to HLIN. HLIN utilizes the four routines that are used to modify the Y-register and/or the address in GBAS in order to change the HIRES cursor position in one of four directions: HRMOVLF to move left, HRMOVRT to move right, HRMOVUP to move up, and HRMOVDN to move down. The preprocessing that HLIN performs initially is to establish the flag value for 0xD3 where bit six determines whether HRMOVLF or HRMOVRT is utilized and bit seven determines whether HRMOVUP or HRMOVDN is utilized. The four possible values that are found at 0xD3 are 0x00, 0x7F, 0x80, and 0xFF. HLIN preprocessing also calculates the horizontal and the vertical deltas between the start and the end coordinates and it sums those deltas in order to create the total number of iterations that are required to draw a particular HIRES line. For example, it requires 472 iterations to draw a HIRES line from coordinate 0,0 to coordinate 279,191, that is, $280 + 192 = 472$. By means of the continual subtraction of the vertical delta from the horizontal delta, HLIN transitions from horizontal processing to vertical processing. HLIN utilizes the identical set of HIRES drawing instructions that are found in DRAWIT. Thus, a HIRES line that is drawn by HLIN does not incorporate any of the current HIRES pixel information with any of the pixel information that is part of the new HIRES line. HLIN draws a colored line to the HIRES screen such that the data that is drawn replaces whatever data may previously exist on the HIRES screen. Other than the two modifications that I made to HLIN in order to correct its flawed logic, the HLIN routine is a well-conceived routine that performs its task as efficiently as possible.

The Applesoft ROTATBL table at 0xF5B3 follows HLIN processing and the values that comprise this table are used to rotate a SHAPE in steps of 5.625 degrees in any single quadrant. ROTATBL provides seventeen entries where the first sixteen cosine entries are used to initialize the horizontal ROTHVAL variable and the last sixteen sine entries are used to initialize the vertical ROTVVAL variable. In other words, fifteen of the

ROTATBL values are shared by the ROTHVAL and ROTVVAL variables. The values that are contained in ROTATBL are calculated as cosine products using the expression $\text{COS}(90 * X/16) * 0x100$. The unmodified Applesoft uses a multiplication factor of $0xFF$ rather than $0x100$ as in the modified Applesoft. This multiplication factor is quite critical because the horizontal and the vertical summation registers are based on setting the C-flag when their sum exceeds $0x100$ and NOT $0xFF$. The 6502 and the 65C02 microprocessor instruction set does not provide any branch instructions that are based on exceeding the value of $0xFF$. Yet, the unmodified version of Applesoft persists in using this logic to its detriment when it calculates the values for ROTATBL. The value of $0x00$ should be utilized for *no rotation* rather than $0xFF$. This seemingly small difference of opinion will be highlighted very soon in this discussion.

The Applesoft DRAWCMD routine at $0xF5C7$ follows the ROTATBL table and this routine is utilized specifically by the Applesoft DRAW statement and by the Applesoft XDRAW statement. The DRAWCMD routine in the modified Applesoft is based on a unique design that combines the Applesoft DRWPNT routine and the common components of the XDRAW1 and the DRAW1 routines which become the DRAWSHP routine that is only found in the unmodified Applesoft. The XDRAW1 and the DRAW1 routines are nearly identical except for their unique pixel processing instructions which I have extracted into the XDRAWIT and the DRAWIT units of DRAWHDR that are only found in the modified Applesoft. DRAWHDR utilizes a unique flag in OPRND that is used to select either XDRAWIT or DRAWIT, and this flag is initialized to one value by the Applesoft XDRAW statement or to another value by the Applesoft DRAW statement. Not only have I extracted a sizeable amount of common code from the unmodified Applesoft, but I have also accelerated the drawing of a SHAPE definition. The first part of DRAWCMD, that is, the DRWPNT routine that is in the unmodified Applesoft, evaluates the expression of either the XDRAW or the DRAW statements for the requested SHAPE definition that is contained in the given SHAPE table. The user must have already initialized the HRSHTPTBL variable with the 16-bit address of the SHAPE table that must already reside in memory. The DOS 4.5.08H SHLOAD command performs the initialization of the HRSHTPTBL variable automatically as well as initializing FRETOP and HIMEM in order to protect the SHAPE table usually from the Character String Pool. If the user selects a valid SHAPE definition, DRAWCMD locates the data for that selected SHAPE definition and initializes the SHAPE variable with the address that points to the data of that selected SHAPE definition. DRAWCMD further evaluates the given expression for the Applesoft AT statement if it should exist. If the AT statement does exist in the expression, DRAWCMD uses the Applesoft GETFNS routine in order to extract and range check the horizontal and the vertical coordinates in where to draw the first pixel of the requested SHAPE definition on the HIRES screen. With all three microprocessor registers initialized with the values of the horizontal and the vertical coordinates, DRAWCMD calls HPOSN in order to calculate the 16-bit scan line address for GBAS and the horizontal byte number for the Y-register. If the AT statement is not found in the given expression, the requested SHAPE definition is drawn starting at the last pixel drawn by the most recently executed HPLLOT, XDRAW, or DRAW statement. Applesoft is certainly not designed to verify the validity of the current values that reside in the HRXCOORD or the HRYCOORD variables, so if these variables contain erroneous values, the requested SHAPE definition might be drawn outside of the selected HIRES screen which could potentially destroy the contents of memory throughout the Apple][computer. At this point in the processing of the equivalent DRWPNT routine, processing would have completed and would have returned to its caller. However, in the modified Applesoft, DRAWCMD now begins the common processing that is found in the XDRAW1 and in the DRAW1 routines, that is, the DRAWSHP routine that is found in the unmodified Applesoft.

Applesoft is capable of rotating a SHAPE definition in all four quadrants, so the rotation value that is found in HRRROT has a range of $00:63$. Each quadrant contains sixteen possible rotations with some constraints imposed by the scale value that is found in HRSCALE. At $0xF600$, the value in HRRROT is divided by sixteen and the target quadrant number is saved to ROTQVAL. The masked value of HRRROT is used to select the requested quadrant rotational value from ROTATBL for the horizontal component that is saved to ROTHVAL and for the vertical component that is saved to ROTVVAL. The Y-register is restored from HRHORZ and

the collision counter HRCOLCNT is initialized to 0. At this point in DRAWSHP processing, the modified Applesoft DRAWCMD makes a dramatic diversion in order to implement the design of a far superior SHAPE drawing algorithm. In the unmodified Applesoft, the fractional vectors ROTHSUM and ROTVSUM are both initialized with 0x80 whenever a SHAPE vector is obtained from a new value that is read from the SHAPE table or from a value that is shifted from the current SHAPE table value. These two fractional vectors determine when it is time to draw a horizontal or cosine pixel and when to draw a vertical or sine pixel. Whenever their value overflows 0x100 by successively adding the ROTHVAL value to ROTHSUM and adding the ROTVVAL value to ROTVSUM, the C-flag becomes set and a new pixel is drawn. This algorithm yields the example SHAPE definition that is shown in Figure 3 using all sixty-four values in HRRROT having a scale value of eleven in HRSCALE. Figure 3 easily shows the visual distortions, the angle irregularities, and the unequal length of all lines other than at the precise horizontal and vertical axes. The DRAWSHP routine that is used in the unmodified Applesoft is simply wrong, unacceptable, and rather useless.

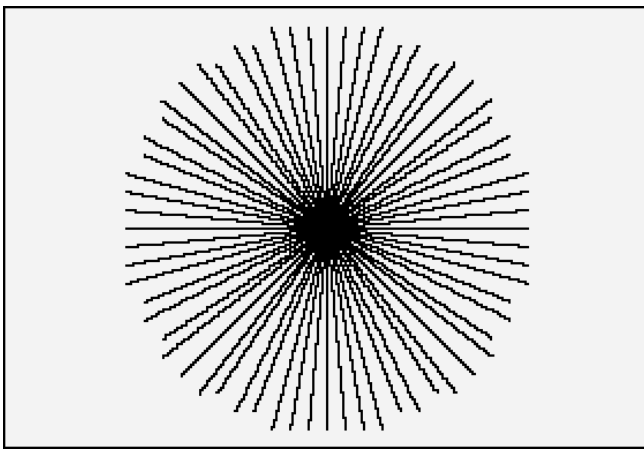


Figure 3. Unmodified Applesoft DRAWCMD

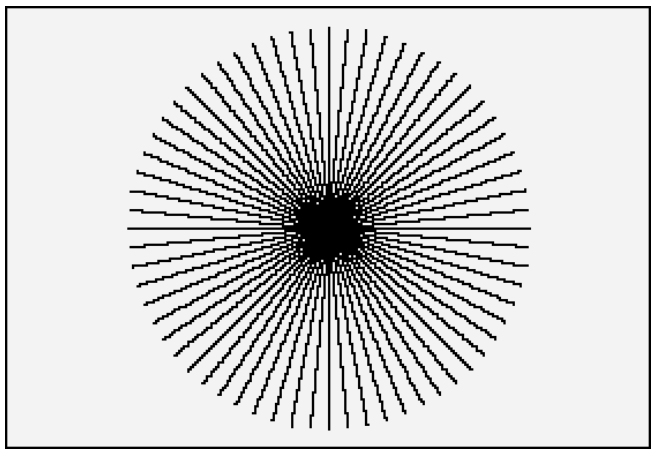


Figure 4. Modified Applesoft DRAWCMD

The SHAPE drawing algorithm in the modified Applesoft that is found in DRAWCMD begins by initializing a new page-zero variable called SHPOLD with the value of 0xFF before the collision counter HRCOLCNT is initialized to 0. As long as the next SHAPE vector, either from a new value that is read from the SHAPE table or a value that is shifted from a current SHAPE table value, *does not change*, ROTHSUM and ROTVSUM are not reinitialized. When SHPOLD is not equal to the current SHAPE vector as in the initial state of 0xFF, the current SHAPE vector is saved to SHPOLD and ROTHSUM and ROTVSUM are initialized to 0x00 and not to 0x80. The X-register is initialized with the value in HRSCALE and the remaining scale-loop processing in DRAWCMD is the essentially the same as in DRAWSHP where DRAWHDR is called with the C-flag clear whenever a horizontal pixel is drawn and DRAWHDR is called with the C-flag set whenever a vertical pixel is drawn. The state of the C-flag prior to calling DRAWHDR is used in the summation of SHPVAL and ROTQVAL in order to determine the next pixel cursor move direction that will either modify GBAS and/or modify the Y-register. I find it truly amazing that with all of the processing that occurs in DRAWHDR that the state of the C-flag is maintained until much later when the summation of SHPVAL and ROTQVAL occurs. The results of this far superior SHAPE drawing algorithm is shown in Figure 4. Figure 4 shows that there are no longer any visual distortions, all angles are regular and equal, and the lengths of all sixty-four lines or spokes are equal to the precise horizontal and vertical axes. Can this SHAPE display get any better? NO! This SHAPE display is the best possible display that can be obtained from any SHAPE drawing algorithm.

What is most remarkable about this unique algorithm is that it only costs eight extra bytes of code and a single page-zero variable. Furthermore, this algorithm confirms that initializing ROTHSUM and ROTVSUM with 0x80 is not correct and that the calculation and utilization of the values in ROTATBL is not correct in the unmodified Applesoft. DRAWCMD in the modified Applesoft is the correct algorithm.

A tremendous amount of Applesoft space is now available in the modified Applesoft after identifying the common components of the XDRAW1 and the DRAW1 routines and removing their duplicate components. After DRAWCMD processing, there is enough Applesoft space for the continuation of SQR at 0xF666, for the continuation of COPYA2F at 0xF68E, for the continuation of COPYF2A at 0xF693, and for the new Applesoft COPYT32A routine at 0xF6A8. COPYT32A first copies T3GUARD to ARGGUARD and then this routine initializes the A-register and the Y-register with the address of the T3 floating-point register. COPYT32A uses the LOADARG routine in order to copy the floating-point number from the T3 floating-point register into the ARG floating-point register. There are five unused bytes at 0xF6B4.

The Applesoft GETFNS routine at 0xF6B9 follows COPYT32A processing and this routine is used by DRAWCMD and by HPLLOT. As previously detailed, GETFNS evaluates a statement expression and it extracts and range checks the horizontal and the vertical coordinates in where to draw the first pixel of the requested SHAPE definition or HLIN on the HIRES screen. With all three microprocessor registers initialized with the values of the horizontal and the vertical coordinates, DRAWCMD can call HPOSN directly or HPLLOT can call HPOSN indirectly by means of HRPLLOT. The call to HPOSN uses the values that are found in all three microprocessor registers in order to calculate the 16-bit scan line address for GBAS and the horizontal byte number for the Y-register. GETFNS range checks the horizontal coordinate value to be less than 280 and the vertical coordinate value to be less than 192. GETFNS also syntactically verifies that there exists a comma between the two coordinate values. The Applesoft HCOLOR statement at 0xF6E9 follows GETFNS processing. HCOLOR evaluates the statement expression for its value, and that value is range checked and utilized as an index into the HRCOLTBL table of color values. HCOLOR extracts the HRCOLTBL color value and it saves that color value to HRCOLOR. The Applesoft HRCOLTBL table is at 0xF6F6 and it follows HCOLOR processing. The HRCOLTBL color value table is comprised of eight color values from two color groups. Earlier in this discussion about the COLSHIFT routine, I pointed out that when the Apple][hardware reads a byte of data from HIRES memory whose address is based on the hardware address mapper logic, that data is clocked into a shift register. If the MSB of that data byte is set, the output of that shift register is delayed by one period of the 14 MHz clock. This delay introduces a shift to the phase angle relative to color burst which changes the perceived color. The two color groups that form the HRCOLTBL table of color values consists of four values whose MSB is OFF and another set of identical values whose MSB is ON. The first color group contains the color values for the colors BLACK1, GREEN, PURPLE, and WHITE1. The second color group contains the color values for the colors BLACK2, ORANGE, BLUE, and WHITE2. Of course, individual television or monitor circuits may present the color of these color values somewhat differently or with a different hue. The horizontal and the vertical timing circuits in the Apple][computer are close enough to the older NTSC standard or to the more recent ATSC standard: the signals do not need to be that precise.

The Applesoft HPLLOT statement at 0xF6FE follows the HRCOLTBL color value table. HPLLOT can be utilized in three construction formats: 1) horizontal and vertical coordinates are specified, 2) the Applesoft TO statement is followed by horizontal and vertical coordinates, 3) horizontal and vertical coordinates are specified, the Applesoft TO statement is specified, then horizontal and vertical coordinates are specified. If the first construction format is found, HPLLOT calls HRPLLOT to draw a single pixel on the appropriate HIRES screen. The *Applesoft TO statement followed by horizontal and vertical coordinates* construction format may be repeated any number of times until a TO statement is no longer found. Once HPLLOT evaluates its expression for the TO statement, the values for the coordinates are obtained by means of GETFNS. The coordinate values are rearranged in the microprocessor registers so that they are made compatible to the

input requirements of HLIN. HLIN draws the requested line on the appropriate HIRES screen and the HPLLOT expression is further evaluated for another T0 statement, otherwise HPLLOT processing exits. Merely by inspection, HLIN exits when its COLCOUNT variable becomes equal to zero at 0xF59C since HLIN has no other exit path. I find it interesting that the Applesoft developers did not utilize this fact about HLIN since they used a very expensive jmp jump instruction at 0xF71E rather than the valid beq branch instruction. It just seems out of character from the Applesoft language developers who appeared to leverage off of every possible nuance they programmed into their routines. There would be no point in modifying HPLLOT since another Applesoft statement follows HPLLOT and whose entry address I wish to maintain. The Applesoft ROT statement at 0xF721 follows HPLLOT processing. ROT evaluates its statement following an equal sign for a rotational value which it stores in HRRROT. ROT does not mask this value with 0x3F or range check this value to be less than 64. However, the HRRROT value is indirectly masked when its upper nibble is utilized in the summation of SHPVAL and ROTQVAL and clamped by a comparison. The Applesoft SCALE statement at 0xF727 follows ROT processing. SCALE evaluates its statement following an equal sign for a scale value which it stores in HRSCALE. The value in HRSCALE can range from 0:255 where a value of 0 is interpreted to be 256. A value of 1 for HRSCALE would provide a point for point reproduction of a SHAPE definition. Applesoft does not initialize the value in HRSCALE even in the Applesoft COLDSTRT routine. It is important to remember to initialize the value in HRSCALE before using the Applesoft XDRAW or DRAW statements. However, the modified Applesoft does initialize HRSCALE to 1 in COLDSTRT due to available space.

Another large amount of Applesoft space is now available in the modified Applesoft where the Applesoft DRWPNT routine was placed at 0xF72D. DRWPNT is incorporated into DRAWCMD in the modified Applesoft. Since I have combined the common components of XDRAW1 and DRAW1 from the unmodified Applesoft and incorporated those common components into DRAWCMD, I have removed the DRWPNT routine. The Applesoft space at 0xF72D in the modified Applesoft is now used for the continuation of the processing for RND. This section of RND processing utilizes the Peasant algorithm in order to multiply ARGMANT and MULMANT and save its 32-bit integer product into FACMANT and IRAND. However, this RND processing must straddle the next two Applesoft statements so that the remaining RND processing at 0xF775 can convert the 32-bit integer currently in FACMANT into a floating-point fraction. That floating-point fraction is either returned to the user without further modification or, if the user supplied a Range value, that floating-point fraction is multiplied by the Range value that was saved in TEMP1. An Applesoft floating-point multiply routine can now be used safely for the Range value multiplication. The product of the Range value multiplication is converted into an integer and that integer is returned to the user either for possible plotting or for graphing.

The third to last and second to last Applesoft statements are the Applesoft DRAW statement at 0xF769 and the Applesoft XDRAW statement of 0xF76F. These two statements reside at the very same memory location as found in the unmodified Applesoft, and they follow DRWPNT processing in that version of Applesoft. I developed a unique software design for these two statements in order to easily differentiate their utilization by their common DRAWHDR routine. In the modified Applesoft, DRAWHDR incorporates some common processing before choosing whether to enter the processing of XDRAWIT or to enter the processing of DRAWIT. That choice or decision is entirely based on the value that DRAW saves into the OPRND variable flag or the value that XDRAW saves into the OPRND variable flag. If the MSB of the OPRND flag is set, then DRAWHDR continues its processing using XDRAWIT. If the MSB of the OPRND flag is clear, then DRAWHDR continues its processing using DRAWIT. In other words, DRAW clears the MSB of the OPRND flag and XDRAW sets the MSB of the OPRND flag. Both statements utilize DRAWCMD in order to begin drawing the selected SHAPE definition on the selected HIRES display from the SHAPE table that is currently in memory.

The Applesoft interpreter in the modified Applesoft ends at 0xF791 after seven bytes of unused space. I have placed the ROM Monitor TITLE at 0xF791 which is a unique ASCII string, and when this ASCII string is displayed, the ROM Monitor TITLE Apple //e+ is shown centered at the top of the TEXT display.

How fun is that! A number of modifications were made to Applesoft that was installed in the Apple //e computer when that computer was first introduced in the early 1980's. These Applesoft modifications were made in order to support an 80 column TEXT display and to support lower case in Applesoft program entry and utilization. The first three modification routines are at 0xF79B, 0xF7A0, and 0xF7AE, and these modification are used in the Applesoft PARSE routine that begins at 0xD56C. Two more modification routines are at 0xF7BE and 0xF7C6, and these modifications are used in the Applesoft LIST statement that begins at 0xD6A5. The modification routine at 0xF7C6 and another modification routine at 0xF7D5 are both used in the Applesoft PRINT statement that begins at 0xDAD5. The final modification routine at 0xF7DC is loosely tied to the modification routine at 0xF7D5 depending upon the setting of the MSB in the RDVID80 switch at 0xC01F. This final modification is utilized by the very last Applesoft statement HTAB. The Applesoft HTAB statement at 0xF7E7 is the last and final Applesoft statement and this statement is located, remarkably, at its traditional Applesoft location. However, its processing is somewhat modified since HTAB now depends on the final modification routine at 0xF7DC. This final modification maintains the location of the horizontal TEXT cursor not only at CH, but also at OURCH for 80 column display utilization. It is rather interesting that some narcissistic individual placed their initials in the final three bytes of the unmodified Applesoft that is found in the Apple][+, that is, the Applesoft that does not support 80 column display and lower case entry. Those three bytes, however, are utilized in the Apple //e Applesoft that does support 80 column display and lower case entry. Thank goodness a clever software engineer was able to utilize just the right amount of available Applesoft space to transform the HTAB statement in order for HTAB to support the utilization of the 80 column display and to allow the entry of and the utilization of lower case ASCII characters in the Apple //e computer.

Derived Transcendental Arithmetic Operations

The *Basic Programming Reference Manual for Applesoft II* on pages 103-104 lists all of the derived transcendental arithmetic operations that can be calculated from the Applesoft intrinsic transcendental arithmetic operations. These intrinsic transcendental arithmetic operations include, of course, the sine, cosine, tangent, arctangent, logarithm, and the exponential operation. The arithmetic operations that support the intrinsic transcendental arithmetic operations include subtraction, addition, multiplication, division, and the square root function as well as the SGN function. These intrinsic Applesoft operations are utilized in order to calculate all of the following derived transcendental arithmetic operations. These Applesoft operations may also be implemented by using the Applesoft DEF FN statement pair.

The secant, denoted as sec, is a trigonometric function that is defined as the ratio of the length of the hypotenuse to the length of the side that is adjacent to a given angle in a right triangle. The secant is also the reciprocal of the cosine function as long as the cosine function is not zero. The sec is expressed mathematically as

$$\text{SEC}(X) = 1 / \text{COS}(X)$$

The cosecant, denoted as csc, is a trigonometric function that is defined as the ratio of the length of the hypotenuse to the length of the side that is opposite to a given angle in a right triangle. The cosecant is also the reciprocal of the sine function as long as the sine function is not zero. The csc is expressed mathematically as

$$\text{CSC}(X) = 1 / \text{SIN}(X)$$

The cotangent, denoted as \cot , is a trigonometric function that is defined as the ratio of the length of the adjacent side to the length of the opposite side in a right triangle. It can also be expressed as the cosine of an angle divided by the sine of that same angle as long as the sine function is not zero or as the reciprocal of the tangent function as long as the tangent function is not zero. The \cot is expressed mathematically as

$$\text{COT}(X) = \text{COS}(X) / \text{SIN}(X) = 1 / \text{TAN}(X)$$

The inverse sine or arcsin, denoted as \sin^{-1} , is the inverse function of the sine trigonometric function. It is used to find the angle whose sine value is a given number. The arcsin is expressed mathematically as

$$\text{ARCSIN}(X) = \text{ATN}(X / \text{SQR}(1 - X^2))$$

The inverse cosine or arccos, denoted as \cos^{-1} , is the inverse function of the cosine trigonometric function. It is used to find the angle whose cosine value is a given number. The arccos is expressed mathematically as

$$\text{ARCCOS}(X) = -\text{ATN}(X / \text{SQR}(1 - X^2)) + \text{PI}/2$$

The inverse secant or arcsec, denoted as \sec^{-1} , is the inverse function of the secant trigonometric function. It is used to find the angle whose secant value is a given number. The arcsec is expressed mathematically as

$$\text{ARCSEC}(X) = \text{ATN}(\text{SQR}(X^2 - 1)) + (\text{SGN}(X) - 1) * \text{PI}/2$$

The inverse cosecant or arccsc, denoted as \csc^{-1} , is the inverse function of the cosecant trigonometric function. It is used to find the angle whose cosecant value is a given number. The arccsc is expressed mathematically as

$$\text{ARCSEC}(X) = \text{ATN}(1 / \text{SQR}(X^2 - 1)) + (\text{SGN}(X) - 1) * \text{PI}/2$$

The inverse cotangent or arccot, denoted as \cot^{-1} , is the inverse function of the cotangent trigonometric function. It is used to find the angle whose cotangent value is a given number. The arccot is expressed mathematically as

$$\text{ARCCOT}(X) = -\text{ATN}(X) + \text{PI}/2$$

The hyperbolic sine or \sinh is the hyperbolic function of the sine trigonometric function and it is based on hyperbolic geometry. Hyperbolic geometry is based on the hyperbola rather than the circle. The \sinh is expressed mathematically as

$$\sinh(X) = (\exp(X) - \exp(-X)) / 2$$

The hyperbolic cosine or \cosh is the hyperbolic function of the cosine trigonometric function and it is based on hyperbolic geometry. Hyperbolic geometry is based on the hyperbola rather than the circle. The \cosh is expressed mathematically as

$$\cosh(X) = (\exp(X) + \exp(-X)) / 2$$

The hyperbolic tangent or \tanh is the hyperbolic function of the tangent trigonometric function and it is based on hyperbolic geometry. Hyperbolic geometry is based on the hyperbola rather than the circle. The \tanh is expressed mathematically as

$$\tanh(X) = - \exp(-X) / (\exp(X) + \exp(-X)) * 2 + 1$$

The hyperbolic secant or sech is the reciprocal of the hyperbolic cosine function and it is based on hyperbolic geometry. Hyperbolic geometry is based on the hyperbola rather than the circle. The sech is expressed mathematically as

$$\operatorname{sech}(X) = 2 / (\exp(X) + \exp(-X))$$

The hyperbolic cosecant or csch is the reciprocal of the hyperbolic sine function and it is based on hyperbolic geometry. Hyperbolic geometry is based on the hyperbola rather than the circle. The csch is expressed mathematically as

$$\operatorname{csch}(X) = 2 / (\exp(X) - \exp(-X))$$

The hyperbolic cotangent or coth is the hyperbolic function of the cotangent trigonometric function and it is based on hyperbolic geometry. Hyperbolic geometry is based on the hyperbola rather than the circle. The coth is expressed mathematically as

$$\operatorname{coth}(X) = \exp(-X) / (\exp(X) - \exp(-X)) * 2 + 1$$

The inverse hyperbolic sine or $\operatorname{arcsinh}$, denoted as \sinh^{-1} , is the inverse function of the hyperbolic sine trigonometric function. It is used to find the hyperbolic angle whose hyperbolic sine value is a given number. The $\operatorname{arcsinh}$ is expressed mathematically as

$$\operatorname{ARCSINH}(X) = \ln(X + \sqrt{ X^2 + 1 })$$

The inverse hyperbolic cosine or arccosh, denoted as \cosh^{-1} , is the inverse function of the hyperbolic cosine trigonometric function. It is used to find the hyperbolic angle whose hyperbolic cosine value is a given number. The arccosh is expressed mathematically as

$$\text{ARCCOSH}(X) = \text{LN}(X + \text{SQR}(X^2 - 1))$$

The inverse hyperbolic tangent or arctanh, denoted as \tanh^{-1} , is the inverse function of the hyperbolic tangent trigonometric function. It is used to find the hyperbolic angle whose hyperbolic tangent value is a given number. The arctanh is expressed mathematically as

$$\text{ARCTANH}(X) = \text{LN}((1 + X) / (1 - X)) / 2$$

The inverse hyperbolic secant or arcsech, denoted as sech^{-1} , is the inverse function of the hyperbolic secant trigonometric function. It is used to find the hyperbolic angle whose hyperbolic secant value is a given number. The arcsech is expressed mathematically as

$$\text{ARCSECH}(X) = \text{LN}(\text{SQR}(1 - X^2) + 1) / X$$

The inverse hyperbolic cosecant or arccsch, denoted as csch^{-1} , is the inverse function of the hyperbolic cosecant trigonometric function. It is used to find the hyperbolic angle whose hyperbolic cosecant value is a given number. The arccsch is expressed mathematically as

$$\text{ARCCSCH}(X) = \text{LN}(\text{SGN}(X) * \text{SQR}(1 + X^2) + 1) / X$$

The inverse hyperbolic cotangent or arccoth, denoted as \coth^{-1} , is the inverse function of the hyperbolic cotangent trigonometric function. It is used to find the hyperbolic angle whose hyperbolic cotangent value is a given number. The arccoth is expressed mathematically as

$$\text{ARCCOTH}(X) = \text{LN}((X + 1) / (X - 1)) / 2$$

The expression $A \bmod B$ refers to the modulo operation which calculates the remainder when A is divided by B . This function could be incorporated into Applesoft as the Applesoft MOD statement. Perhaps the unmodified Applesoft sine polynomials could be reinstalled into the modified Applesoft in order to provide the Applesoft space that would be required to calculate this function. The modulo is expressed mathematically as

$$\text{MOD}(A) = \text{INT}((A/B - \text{INT}(A/B)) * B + 0.5) * \text{SGN}(A/B)$$

Testing Applesoft Floating-Point Routines

The Call-A.P.P.L.E. magazine published the article *Floating Point Arithmetic in Applesoft BASIC* by James W. Thomas in July, 1985, and this article appeared on pages 15 to 18. This article introduces the Standard Apple Numerics Environment or SANE as a result of efforts from the Apple Numerics Group. SANE is utilized in Apple Works, MacPascal, MacBASIC, the Lisa Workshop, and in several other Macintosh languages and applications. However, the purpose of this article is to bring attention to the problems and the issues that are found in Applesoft arithmetic which can be quantitatively identified. This article did not mention nor did it infer whether or not any arithmetic that is found in Applesoft BASIC is utilized in the development of SANE. That information would have been rather interesting to know. Returning to the content of Mr. Thomas's article, if I have been in anyway successful in eliminating any of the identified problems in Applesoft arithmetic, the examples from this article should easily prove my success.

Integer numbers as large as 1,048,576 or 2^{20} can be precisely expressed by an Applesoft floating-point number. However, not all decimal numbers can be precisely expressed by this floating-point notation. Applesoft floating-point notation is limited by its 8-bit exponent and its 32-bit mantissa, and Applesoft notation cannot precisely express many decimal numbers. Even IEEE floating-point double precision numbers cannot precisely express many decimal numbers. Mr. Thomas provides a very simple test to show the weakness of all similar floating-point notations. I have magnified the range of this test in order to also show the strengths that are inherent in Applesoft floating-point numbers as well.

```
10 HOME
20 FOR I = 0 TO 10
30 READ A
40 IF A = 0 OR A > .95 THEN PRINT
   "A = "; A; ".0"; GOTO 60
50 PRINT "A = "; A; ".0"; GOTO 60
60 GOSUB 200: PRINT "B = "; B
70 GOSUB 300: PRINT "B = "; B
80 NEXT I
90 END
200 B = 0
210 FOR J = 1 TO 1000: B = B + A: NEXT J
220 RETURN
300 B = INT (B + 0.25)
310 FOR J = 1 TO 1000: B = B - A: NEXT J
320 RETURN
1000 DATA 0, .1, .2, .3, .4, .5, .6, .7,
      .8, .9, 1
1*
```

Figure 5. Test 1 Applesoft Program

The Applesoft program for Test 1 is shown in Figure 5. Eleven fractional values are tested using 1000 loops of successive addition and 1000 loops of successive subtraction using the DATA values that are shown in line 1000. The first 1000 loops add the same value to a running sum and then the main routine prints that final sum on line 60. The next 1000 loops begin with an integer of that final sum and it subtracts that same value and then the main routine prints the final value in line 70. Both versions of Applesoft in an Apple //e display nearly the same problems when adding the same small fractional value repeatedly as shown in Figure 6 for the unmodified Applesoft and in Figure 7 for the modified Applesoft. Applesoft appears to have no problems when repeatedly adding the values of 0.3, 0.5, or 0.6. Why is that? For

0.1, 0.2, 0.4, and 0.8 the mantissa is 0x4CCCCCD, for 0.3 and 0.6 the mantissa is 0x1999999A, for 0.5 the mantissa is 0x00000000, for 0.7 the mantissa is 0x33333333, and for 0.9 the mantissa is 0x66666666. Apparently, the addition and roundup of a mantissa value of 0x1999999A creates no summing issues in Applesoft. The other mantissa values do cause summing issues for successive addition and roundup in Applesoft. Only the successive subtraction of 0.5 and 1.0 is handled well in Applesoft. After every addition or subtraction, Applesoft must call COPYFAC in order to save the contents of the FAC floating-point register to memory so that its value can be displayed. This call to COPYFAC requires a call to RNDUP and that call is unavoidable. I have no doubt that if these addition and subtraction loops remained entirely within Applesoft, better results would be obtained where RNDUP is called only once. The residual error is quite small and it shows that values to five or six places are precise in Applesoft.

```

A = 0.0, B = 0
A = 0.1, B = 0
A = 0.1, B = 99.9999963
A = 0.2, B = 3.71650094E-06
A = 0.2, B = 199.999993
A = 0.2, B = 7.43300188E-06
A = 0.3, B = 300
A = 0.3, B = -2.18860805E-07
A = 0.4, B = 399.999985
A = 0.4, B = 1.48660038E-05
A = 0.5, B = 500
A = 0.5, B = 0
A = 0.6, B = 600.000001
A = 0.6, B = -4.3772161E-07
A = 0.7, B = 700.000027
A = 0.7, B = -2.71582976E-05
A = 0.8, B = 799.999971
A = 0.8, B = 2.97320075E-05
A = 0.9, B = 900.000032
A = 0.9, B = -3.15443613E-05
A = 1.0, B = 1000
A = 1.0, B = 0
1

```

Figure 6. Test 1 Unmodified Applesoft

```

A = 0.0, B = 0
A = 0.1, B = 0
A = 0.1, B = 99.9999963
A = 0.2, B = 3.71650094E-06
A = 0.2, B = 199.999993
A = 0.2, B = 7.43300188E-06
A = 0.3, B = 300
A = 0.3, B = -2.18860805E-07
A = 0.4, B = 399.999985
A = 0.4, B = 1.48660038E-05
A = 0.5, B = 500
A = 0.5, B = 0
A = 0.6, B = 600
A = 0.6, B = -4.3772161E-07
A = 0.7, B = 700.000027
A = 0.7, B = -2.71582976E-05
A = 0.8, B = 799.99997
A = 0.8, B = 2.97320075E-05
A = 0.9, B = 900.000032
A = 0.9, B = -3.15443613E-05
A = 1.0, B = 1000
A = 1.0, B = 0
1

```

Figure 7. Test 1 Modified Applesoft

Non-commutative addition where intermediate operations may present different results to subsequent operations can occur depending on the positions of those operations that are determined by the formula evaluation routine. In other words, the consequence of the formula evaluation routine may present different results when the variables of the formula are manipulated in a slightly different order. For example, the formula $A + BC$ should present the same result as $BC + A$. Mr. Thomas correctly points out that in the unmodified Applesoft, BC in the first formula stays in the FAC floating-point register with its guard byte and BC in the second formula is rounded and saved as a temporary variable. When A is added to BC , different results are presented to the user. I wrote Test 2 so that it utilizes the LIST statement to automatically list the Applesoft program when the DOS RUN command is issued on the Apple Command Line. Test 2 also prints the contents of each variable showing only what FPOUT is capable of printing even though variable A is equal to 7FC0000001 in memory in both Figure 8 and in Figure 9. That lowly 32nd mantissa bit interferes in the addition of the second formula because of the inadequate utilization of guard bytes in the unmodified Applesoft. Figure 9 shows off the redesigned LIST routine and the redesigned FPOUT routine in the modified Applesoft as well as its immunity to non-commutative addition. LIST increases the number of characters that are displayed on a TEXT line. FPOUT prefaces a fractional value with a 0 if scientific notation is not utilized. Because the modified Applesoft utilizes guard bytes for every set of internal arithmetic calculations, it makes no difference whether A is added to BC or whether BC is added to A . That lowly 32nd mantissa bit is managed with a 40-bit mantissa in either formula. The results from Figure 9 show that non-commutative addition errors have most likely been eliminated in the modified Applesoft.

```

10 HOME : LIST : PRINT
20 A = 2 ^ ( - 2 ) + 2 ^ ( - 3 ) +
30 2 ^ ( - 33 )
40 B = 2 ^ ( - 1 ) + 2 ^ ( - 32 )
50 C = 2 ^ ( - 1 ) + 2 ^ ( - 2 )
60 PRINT "A = "; A
70 PRINT "B = "; B
80 PRINT "C = "; C: PRINT
90 PRINT "A + B * C = "; A + B * C
100 PRINT "B * C + A = "; B * C +
110 A
END
A = -.375
B = .5
C = .75
A + B * C = 0
B * C + A = 1.16415322E-10
]

```

Figure 8. Test 2 Unmodified Applesoft

```

10 HOME : LIST : PRINT
20 A = 2 ^ ( - 2 ) + 2 ^ ( - 3 ) + 2 ^
30 ( - 33 )
40 B = 2 ^ ( - 1 ) + 2 ^ ( - 32 )
50 C = 2 ^ ( - 1 ) + 2 ^ ( - 2 )
60 PRINT "A = "; A
70 PRINT "B = "; B
80 PRINT "C = "; C: PRINT
90 PRINT "A + B * C = "; A + B * C
100 PRINT "B * C + A = "; B * C + A
110 END
A = -.375
B = .5
C = .75
A + B * C = 0
B * C + A = 0
]

```

Figure 9. Test 2 Modified Applesoft

```

10 HOME : LIST : PRINT : PRINT
20 A = 1 - 2 ^ ( - 31 )
30 B = 2 ^ ( - 33 )
40 C = 2 ^ ( - 1 )
50 PRINT "A = "; A
60 PRINT "B = "; B
70 PRINT "C = "; C: PRINT
80 PRINT "C + (A + B) * C - A = "
90 PRINT "C + (A + B) * C - A = "
100 END
A = 1
B = 1.16415322E-10
C = .5
C + (A + B) * C - A = 4.65661287E-10
C + C * (A + B) - A = 2.32830644E-10
]

```

Figure 10. Test 3 Unmodified Applesoft

```

10 HOME : LIST : PRINT : PRINT
20 A = 1 - 2 ^ ( - 31 )
30 B = 2 ^ ( - 33 )
40 C = 2 ^ ( - 1 )
50 PRINT "A = "; A
60 PRINT "B = "; B
70 PRINT "C = "; C: PRINT
80 PRINT "C + (A + B) * C - A = "; C +
90 PRINT "C + (A + B) * C - A = "; C +
100 END
A = 1
B = 1.16415322E-10
C = .5
C + (A + B) * C - A = 0
C + C * (A + B) - A = 0
]

```

Figure 11. Test 3 Modified Applesoft

Non-commutative multiplication where intermediate operations may present different results to subsequent operations can occur depending on the positions of those operations that are determined by the formula evaluation routine. In other words, the consequence of the formula evaluation routine may present different results when the variables of the formula are manipulated in a slightly different order. For example, the formula $C + (A + B) * C - A$ should present the same result as $C + C * (A + B) - A$. Mr. Thomas correctly points out that the remedy to evaluate these two formulas would involve nontrivial design decisions. He points out that the guard bytes should be pushed onto the STACK, also. I think he incorrectly believes that rounding the operands before their utilization would remedy the evaluation of these two formulas. And, of course, we both concur that the addition, subtraction, and multiplication routines in Applesoft require modifications in how operands that have different sized significands are utilized. I also wrote Test 3 so that it utilizes the LIST statement to automatically list the Applesoft program when the DOS RUN command is issued on the Apple Command Line. Test 3 prints the contents of each variable where variable B is equal to 6000000000 in memory in both Figure 10 and in Figure 11. Applesoft has difficulty when it is required to normalize variables for addition and for subtraction when the exponents of those variables differ by 0x20 and more. As shown in Figure 10, I find it a bit surprising that the first formula generates a remainder difference that is four times the value of the variable B. The second formula generates a remainder

difference that is twice the value of the variable B in the unmodified Applesoft. The modified Applesoft does utilize modifications to its addition, subtraction, and multiplication routines and these routines utilize guard bytes in all stages of their processing. Guard bytes are also pushed and popped from the STACK in the modified Applesoft as well. The results from Figure 11 show that non-commutative multiplication errors have most likely been eliminated in the modified Applesoft.

```

10 HOME : LIST : PRINT : PRINT
20 A = 2 ^ ( - 1 )
30 B = 2 ^ ( - 24 ) - 2 ^ ( - 33 )
40 PRINT "A = "; A
50 PRINT "B = "; B : PRINT
60 IF ( A + B ) = ( A + B ) THEN PRINT
70 IF ( A + B ) > ( A + B ) THEN PRINT
80 IF ( A + B ) < ( A + B ) THEN PRINT
90 END

A = 5
B = 5.94882295E-08
A+B > A+B
]␣

```

Figure 12. Test 4 Unmodified Applesoft

```

10 HOME : LIST : PRINT : PRINT
20 A = 2 ^ ( - 1 )
30 B = 2 ^ ( - 24 ) - 2 ^ ( - 33 )
40 PRINT "A = "; A
50 PRINT "B = "; B : PRINT
60 IF ( A + B ) = ( A + B ) THEN PRINT
70 IF ( A + B ) > ( A + B ) THEN PRINT
80 IF ( A + B ) < ( A + B ) THEN PRINT
90 END

A = 0.5
B = 5.94882295E-08
A+B = A+B
]␣

```

Figure 13. Test 4 Modified Applesoft

Non-reflexive equality processing where intermediate operations may present different results to subsequent operations can occur and they are determined by the formula evaluation and comparator routines. In other words, the consequence of the formula evaluation and comparator routines may present different results when variables are compared even without changing their order. For example, the formula $A \text{ op } B$ should compare precisely to $A \text{ op } B$. Mr. Thomas states that the formula evaluator routine rounds and pushes one of the $A \text{ op } B$ results onto the stack and leaves the other $A \text{ op } B$ result in the FAC floating-point register and unrounded before the results are compared. If Mr. Thomas is referring to the FRMSTAK3 routine in the unmodified Applesoft, the FAC floating-point register is first rounded by the RNDUP routine before the register is pushed onto the STACK. Rather than call the RNDUP routine in the modified Applesoft, I push FACGUARD onto the STACK before I push the FAC floating-point mantissa onto the STACK. Likewise, in the modified Applesoft, I pull ARGUARD off the STACK after I pull the ARG floating-point mantissa off the STACK in the NOTMATH4 routine. The FAC floating-point register is then compared to the ARG floating-point register using FPCOMP which I have also modified in the modified Applesoft. I have serious objections to entertaining the use of any rounded values in Applesoft as a means to *fix* the Applesoft compare algorithm as Mr. Thomas suggests. I wrote Test 4 so that it utilizes the LIST statement to automatically list the Applesoft program when the DOS RUN command is issued on the Apple Command Line. Test 4 prints the contents of each variable where variable B is equal to 687F800000 in memory in both Figure 12 and in Figure 13. Applesoft has difficulty when it is required to normalize variables for addition and for subtraction when the exponents of two variables differ by 0x20 and more. These two variables differ by 0x17 so Applesoft should have no problems in adding these two variables, putting their sum onto the STACK, calculating their sum again, pulling the first sum off the STACK, and then comparing the two sums. Without question, the mantissas of these two sums should precisely compare. Figure 12 shows that the unmodified Applesoft has reached the wrong conclusion and that the mantissa of one sum is greater than the mantissa of the other sum. Figure 13 shows that the modified Applesoft has reached the correct conclusion and that

the mantissa of one sum is equal to the mantissa of the other sum. The results from Figure 13 show that non-reflexive equality processing errors have most likely been eliminated in the modified Applesoft utilizing far different techniques and more powerful modifications than what Mr. Thomas has suggested.

```

10 HOME : LIST : PRINT : PRINT
20 A = - (2 ^ ( - 127))
30 B = (A / 2) / A
40 PRINT "A = "; A: PRINT
50 PRINT "B = "; B
60 END

A = -5.87747176E-39
B = -.5
]

```

Figure 14. Test 5 Unmodified Applesoft

```

10 HOME : LIST : PRINT : PRINT
20 A = - (2 ^ ( - 127))
30 B = (A / 2) / A
40 PRINT "A = "; A: PRINT
50 PRINT "B = "; B
60 END

A = -5.87747175E-39
B = 0.5
]

```

Figure 15. Test 5 Modified Applesoft

The PROCXP routine in Applesoft purposefully made all quotients positive when its exponent is found to be equal to -128. This is not a software bug as Mr. Thomas seems to believe. This was purposefully coded and I have no idea why it was permitted to stand. Instead of storing zero into FACSIGN as the unmodified Applesoft does, I always store XORSIGN into FACSIGN in the modified Applesoft regardless whether the addition of #EXPBIAS to FACEXP is zero or not. Test 5 is shown in Figure 14 for the unmodified Applesoft. And, indeed, as Mr. Thomas points out, the sign of a small quotient value is wrong. The modified PROCXP routine in the modified Applesoft properly handles the sign of the quotient in all cases as shown in Figure 15. The results from Figure 15 show that the sign of the small quotient error in Applesoft has most likely been eliminated in the modified Applesoft.

```

10 HOME : LIST : PRINT
20 A = 2 ^ ( - 1) + 2 ^ ( - 24) -
  2 ^ ( - 31)
30 B = 1 * A
40 PRINT "A = "; A: PRINT
50 PRINT "B = "; B: PRINT
60 C = 0.500000059
70 PRINT "C = "; C
80 END

A = .50000003
B = .500000015
C = .500000029
]mon
*87B.88F
0087B- 41 00 80 00 00
00880- 00 FE 42 00 80 00 00 00
00888- 7F 43 00 80 00 00 00 FD
**

```

Figure 16. Test 6 Unmodified Applesoft

```

10 HOME : LIST : PRINT
20 A = 2 ^ ( - 1) + 2 ^ ( - 24) - 2
  ^ ( - 31)
30 B = 1 * A
40 PRINT "A = "; A: PRINT
50 PRINT "B = "; B: PRINT
60 C = 0.500000059
70 PRINT "C = "; C
80 END

A = 0.500000059
B = 0.500000059
C = 0.500000059
]mon
*87B.88F
0087B- 41 00 80 00 00
00880- 00 FE 42 00 80 00 00 00
00888- FE 43 00 80 00 00 00 FD
**

```

Figure 17. Test 6 Modified Applesoft

The Applesoft multiplication routine uses its full processing horsepower to shift a multiplicand a full eight bits when the multiplier byte is zero. In certain numbers where the second and the third mantissa bytes are zero, the Applesoft multiplication routine calls the SHFTBYT1 routine on behalf of the third mantissa byte (which is zero) with the C-flag properly set. The SHFTBYT1 routine exits properly and the routine is designed to clear the C-flag. Now, when the Applesoft multiplication routine calls the SHFTBYT1 routine on behalf of the second mantissa byte (which is zero), the C-flag is not properly set: the C-flag is clear. The bcs instruction at 0xE8F0 is designed as the exit for the Applesoft multiplication routine, the branch is not taken, and the multiplication routine enters processing that is not intended for this multiplication routine. Hence, the multiplicand is unfortunately shifted one bit to the right. This is truly a mistake on behalf of the Applesoft language developers. This is a forgotten possibility that can happen when processing certain numbers. The solution is to preface the call to the SHFTBYT1 routine with a sec instruction or to preface the SHFTBYT1 routine itself with a sec instruction. I chose the second option and I inserted a sec instruction at 0xE8CE at the top of the SHFTBYT1 routine. Only the Applesoft multiplication routine calls the SHFTBYT1 routine. As shown in Figure 16 for Test 6, the unmodified Applesoft shifts the multiplicand one bit to the right when the formula $B = 1 * A$ is processed. The variable C is saved properly to memory but the values of variables A, B, and C are not printed with their correct values. As shown in Figure 17 for Test 6, the modified Applesoft does not shift the multiplicand erroneously when the formula $B = 1 * A$ is processed. The variable C is saved properly to memory and the value of variables A, B, and C are printed with their correct values. Figure 17 demonstrates that the binary to decimal conversion algorithm is working correctly whereas the binary to decimal conversion algorithm in Figure 16 is not working correctly. I can fiercely state one more time, the results from Figure 17 show that non-commutative multiplication errors have most likely been eliminated in the modified Applesoft.

Mr. Thomas does elaborate on the decimal to binary and the binary to decimal conversion routines that reside in Applesoft. Surely, the GETINT decimal to binary conversion routine at 0xEC4A does its processing remarkably well. There is no question that the unmodified Applesoft converts the value of C properly into memory. My only complaint about the GETINT routine is that the Applesoft language developers inserted the ADD2FAC routine unnecessarily into the middle of the GETINT routine. The FPOUT binary to decimal conversion routine at 0xED34 depends on the services of the key routines MULFAC10, DIVFAC10, and FPCOMP. I have modified and fine-tuned all of these routines in the modified Applesoft. The results of this concerted effort is on display in Figure 17. No longer are the values of variables printed incorrectly. Having implemented the use of guard bytes in all stages of floating-point arithmetic assists the binary to decimal conversion routine to produce a far more accurate representation of a binary number that resides in memory.

The POWER^{\wedge} statement in Applesoft depends on INT, FPCOMP, LN, MULT, and EXP processing. Both the LN and the EXP routines must each process a Taylor polynomial expansion. Both of these polynomial expansions are supplied with modified polynomials over which I have no control except for the sine polynomials. These polynomials have been precisely tuned in order to produce the nicely behaved output values that are shown for the variable A in Figure 18 for Test 7, or at least up to $1.0\text{E}-09$. Perhaps a DATA statement would have been a better design choice to use for this test rather than using the POWER statement. This test was designed by Mr. Thomas so I yield to his test design for the moment. POWER statement processing tends to produce undervalued variables in the unmodified Applesoft as shown in Figure 18 and the modified Applesoft tends to produce overvalued variables as shown in Figure 19. Both versions of Applesoft do not produce a ratio of 1.0 for angles that approach zero, or at least less than 0.001 radians as shown in both figures. The results that are shown in Figures 20 and 21 for Test 7.1 insulate the values of A from the POWER statement so that a better comparison of the sine ratios can be easily observed. These sine ratios in the two Applesoft versions are nearly identical which should be expected since the eleven theoretical sine polynomials that are used in the modified Applesoft yield virtually the same results as the six modified sine polynomials that are used in the unmodified Applesoft.

[illegible]

Figure 18. Test 7 Unmodified Applesoft

[illegible]

Figure 19. Test 7 Modified Applesoft

[illegible]

Figure 20. Test 7.1 Unmodified Applesoft

[illegible]

Figure 21. Test 7.1 Modified Applesoft

[illegible]

Figure 22. Test 8 Unmodified Applesoft

[illegible]

Figure 23. Test 8 Modified Applesoft

Mr. Thomas conducted another series of sine calculations specifically for very large arguments. These calculations are shown in Figures 22 and 23 for Test 8. I cannot be sure what the point is for Test 8 except to show that for very large arguments, the sine functions will produce a value of zero. In fact, the modified Applesoft produces a value of zero for an input argument that is not as large as the input argument in the unmodified Applesoft, that is, at $1.0E+08$ rather than at $1.0E+09$. Mr. Thomas theorizes that the argument reduction algorithm that is utilized in sine processing is the actual culprit for that function being unable to obtain a valid quotient remainder that is obtained when the input argument is successively divided by $2*\pi$. In its attempt to properly position the input argument in the correct quadrant may also contribute to sine processing being unable to process this very small quotient. Mr. Sander-Cederlof suggests that the sine argument reduction algorithm can be replaced with far simpler ways in order to determine an input angle as a fraction of a full circle and fold the range of that angle into a quarter circle. This suggestion infers that the utilization of a MOD function would be a far better choice in processing very large arguments for the sine function. Mr. Thomas correctly points out that the flaws in the sine argument reduction algorithm are exacerbated in the cosine and in the tangent functions because those two functions depend entirely on the sine function. In Applesoft cosine processing, an argument is processed as $\cos(x) = \sin(x + \pi/2)$. In Applesoft tangent processing, an argument is processed as $\tan(x) = \sin(x) / \cos(x)$. And, the Applesoft tangent processing is even further flawed since $\tan(x) = \sin(x) / \sin(x + \pi/2)$. In other words, Applesoft tangent processing requires two calls to the sine function. Both the cosine and the tangent functions inherit the processing flaws that are native to the sine function.

```

10 HOME : LIST : PRINT
20 FOR I = 1 TO 12: READ A:B =
   TAN (A) / A: PRINT "A = ";A
   ;: HTAB 22: PRINT "B = ";B: NEXT
30 END
100 DATA .1,.01,1E-3,1E-4,1E-5,
1E-6,1E-7,1E-8,1E-9,1E-10,1E
-11,1E-12

A = .1           B = 1.00334672
A = .01          B = 1.000033334
A = 1E-03        B = 1.000000333
A = 1E-04        B = 1
A = 1E-05        B = .999999994
A = 1E-06        B = .9999998797
A = 1E-07        B = .9999984542
A = 1E-08        B = .9999755932
A = 1E-09        B = .997184394
A = 1E-10        B = 0
A = 1E-11        B = 0
A = 1E-12        B = 0

```

Figure 24. Test 9 Unmodified Applesoft

```

10 HOME : LIST : PRINT
20 FOR I = 1 TO 12: READ A:B = TAN
   (A) / A: PRINT "A = ";A;: HTAB
   22: PRINT "B = ";B: NEXT
30 END
100 DATA .1,.01,1E-3,1E-4,1E-5,1E-
6,1E-7,1E-8,1E-9,1E-10,1E-11,1E
-12

A = .1           B = 1.00334672
A = .01          B = 1.000033333
A = 1E-03        B = 1.000000333
A = 1E-04        B = 1
A = 1E-05        B = .999999994
A = 1E-06        B = .9999998797
A = 1E-07        B = .999998451
A = 1E-08        B = .9999755929
A = 1E-09        B = .997184394
A = 1E-10        B = 0
A = 1E-11        B = 0
A = 1E-12        B = 0

```

Figure 25. Test 9 Modified Applesoft

Another good test that shows off the behavior of the Applesoft tangent function for a range of arguments from small arguments to very small arguments is shown in Figures 24 and 25 for Test 9. For a small input argument, the tangent function returns with the same value as the input argument. Thus, when that value is divided by the input argument, a quotient of 1.0 should be obtained. The quotients that are shown in Figures 24 and 25 for the unmodified Applesoft and for the modified Applesoft, respectively, are nearly identical, though they are not precisely equal to 1.0 in all cases.

A very good way to test the Applesoft sine and cosine functions and their behavior together is to utilize a trigonometric identity that generates a known and easy to verify output. One trigonometric identity is $\sin(x)^2 + \cos(x)^2 = 1$. I have already established that the Applesoft POWER and EXP functions depend

on modified polynomials and that these functions do not provide optimal results for small input values. It is not necessary to utilize the POWER function in Test 10. One can simply multiply $\sin(x)$ times $\sin(x)$ to obtain $\sin(x)^2$. This is precisely what I do in Test 10 rather than what Mr. Thomas does in his version of this same test. Anyone who has studied this document will come to the conclusion that it is better to avoid the Applesoft POWER and EXP functions if at all possible. Even the multiply routine in the unmodified Applesoft is far more trustworthy over the Applesoft POWER and EXP functions as shown in Figure 26. However, the arithmetic routines in the modified Applesoft are obviously far more precise in calculating this particular trigonometric identity as shown in Figure 27. The multiplication of the Applesoft sine and cosine functions and their addition is precisely 1.0 until extraordinarily large input values are utilized. I believe that most scientists and engineers would restrict their utilization of such input values to those that might transcribe a circle no more than twice.

```

10 HOME : LIST : PRINT
20 FOR I = 1 TO 12: READ A: PRINT
   "A = "; A; HTAB 18: PRINT "S
   UM = "; SIN (A) * SIN (A) +
   COS (A) * COS (A): NEXT
30 END
100 DATA 10,100,1000,1E+4,1E+5
   1E+6,1E+7,1E+8,1E+9,1E+10,1
   E+11,1E+12

A = 10          SUM = 1
A = 100         SUM = 1.00000002
A = 1000        SUM = 1
A = 10000       SUM = 1
A = 100000      SUM = 1.00000171
A = 1000000     SUM = 1
A = 10000000    SUM = 1.997658571
A = 100000000   SUM = 1
A = 1E+09       SUM = 1.35355339
A = 1E+10       SUM = 0
A = 1E+11       SUM = 0
A = 1E+12       SUM = 0
]

```

Figure 26. Test 10 Unmodified Applesoft

```

10 HOME : LIST : PRINT
20 FOR I = 1 TO 12: READ A: PRINT
   "A = "; A; HTAB 18: PRINT "SUM
   = "; SIN (A) * SIN (A) + COS
   (A) * COS (A): NEXT
30 END
100 DATA 10,100,1000,1E+4,1E+5,1E
   +6,1E+7,1E+8,1E+9,1E+10,1E+11,1
   E+12

A = 10          SUM = 1
A = 100         SUM = 1
A = 1000        SUM = 1
A = 10000       SUM = 1
A = 100000      SUM = 1
A = 1000000     SUM = 1
A = 10000000    SUM = 1
A = 100000000   SUM = 1
A = 1E+09       SUM = 1.971286599
A = 1E+10       SUM = 0
A = 1E+11       SUM = 0
A = 1E+12       SUM = 0
]

```

Figure 27. Test 10 Modified Applesoft

```

10 HOME : LIST : PRINT
20 FOR I = 1 TO 12: READ A: PRINT
   "A1 = "; SIN (2 * A); HTAB
   20: PRINT "A2 = "; 2 * SIN (
   A) * COS (A): NEXT
30 END
100 DATA 10,100,1000,1E+4,1E+5
   1E+6,1E+7,1E+8,1E+9,1E+10,1
   E+11,1E+12

A1 = .912945251  A2 = .912945251
A1 = -.873297282 A2 = -.873297294
A1 = .930039928  A2 = .930039927
A1 = .581984028  A2 = .581984028
A1 = -.071460893 A2 = -.0714608542
A1 = -.655492853 A2 = -.655492853
A1 = -.766120466 A2 = -.766120466
A1 = -.671558955 A2 = -.671558955
A1 = 1           A2 = 1.30656297
A1 = 0           A2 = 0
A1 = 0           A2 = 0
A1 = 0           A2 = 0
]

```

Figure 28. Test 11 Unmodified Applesoft

```

10 HOME : LIST : PRINT
20 FOR I = 1 TO 12: READ A: PRINT
   "A1 = "; SIN (2 * A); HTAB 20:
   PRINT "A2 = "; 2 * SIN (A) * COS
   (A): NEXT
30 END
100 DATA 10,100,1000,1E+4,1E+5,1E
   +6,1E+7,1E+8,1E+9,1E+10,1E+11,1
   E+12

A1 = 0.912945251  A2 = 0.91294525
A1 = -0.873297318 A2 = -0.873297321
A1 = 0.930039928  A2 = 0.930039928
A1 = 0.581982696  A2 = 0.581982696
A1 = -0.0714773291 A2 = -0.0714773291
A1 = -0.655485570 A2 = -0.655485570
A1 = -0.767693137 A2 = -0.767693137
A1 = -0.660402362 A2 = -0.660402362
A1 = 0            A2 = 0
A1 = 0            A2 = 0
A1 = 0            A2 = 0
A1 = 0            A2 = 0
]

```

Figure 29. Test 11 Modified Applesoft

There is another useful trigonometric identity that can be utilized in order to provide a means to expose problems in the Applesoft trigonometric functions and in the Applesoft arithmetic routines. This

trigonometric identity is $\sin(2x) = 2 * \sin(x) * \cos(x)$. Test 11 is designed to calculate and show the result from $\sin(2x)$ processing and calculate and show the result from $2 * \sin(x) * \cos(x)$ processing. Figure 28 shows some variation of the results for the unmodified Applesoft. On the other hand, Figure 29 shows far less, if any, variation of the results for the modified Applesoft throughout the entire numerical range that is tested. This numerical range, again, is artificial and certainly far beyond that which is practical in any research or hardware analysis of reasonable functionality. Perhaps Mr. Thomas might have subtracted these two results or divided these two results to more clearly visualize the degree of equalness for this trigonometric identity as computed by the respective Applesoft. It is worth repeating that the modified Applesoft will provide excellent results for all trigonometric functions as long as the input variable is restricted in its utilization such that its value might transcribe a circle no more than twice.

Mr. Thomas does a thorough review of all of the implications that the above tests have shown. The Applesoft programmer has been failed by less accurate floating-point variables, the lack of guard bytes, the less than stellar binary to decimal conversions, the technical flaws in multiplication, a poorly designed argument reduction algorithm for the trigonometric functions, all of the excessive rounding, the stack pushes and pops, and the incompetence of the logarithmic and exponential functions. The errors, the failures, and the bugs that are inherent in the unmodified Applesoft have now been exposed and they have all been corrected in the modified Applesoft within the space that is provided for Applesoft as well as the addition of new functionality and better functionality for the entire Applesoft statement repertoire.

My only regret is not knowing how the modified polynomials are calculated for the logarithm, the exponential, the sine, and the arctangent functions. That was genius.

Installing Applesoft

My journey through Applesoft has only been an intellectual exercise unless I can actually inject this Applesoft into a real Apple //e computer and use this Applesoft for something creative. To use the modified Applesoft in an Apple //e computer, the Applesoft must be programmed into either a 128 Kb EPROM or two 64 Kb EPROMs depending on the model of the Apple //e. The *Lisa* assembler creates four output files when *Lisa* assembles my source code for the CXROM, for Applesoft, and for the ROM Monitor. The current version of my source code is ROM2E.SW16GCR.14. The four output files that *Lisa* generates are CØROM, DØROM, EØROM, and FØROM. Each of these four files are 4096 or 0x1000 bytes in size. I wrote an EXEC file BLDROMS that I put on the Virtual][ROM 14 Build Volume and that EXEC file builds the single SW16GCR.CF.ROM binary file and it builds the SW16GCR.CD.ROM and the SW16GCF.EF.ROM pair of binary files. These files can be used directly by the PROmGRAMMER hardware in order to program the desired EPROMs for a particular Apple //e. This is certainly the easiest and the most direct path in having the modified Applesoft internal to a real Apple //e computer. The BLDROMS EXEC file is shown in Figure 30.

I enjoy the freedom and the ease to develop 6502 assembly language software for an Apple //e computer using an Apple //e emulator on my Apple MacBook Pro computer. The Apple //e emulator that I have always used and trusted is Virtual][which is written and copyright by Gerard Putter. I have had a license to use Virtual][for many years and I am currently using Version 12.1.1. Building a 256 Kb software ROM for Virtual][is a little difficult and it requires the assistance of a C language program that processes in a UNIX environment. The 256 Kb software ROM is actually constructed by another EXEC file BLDV2ROM that I also put on the Virtual][ROM 14 Build Volume. The BLDV2ROM EXEC file is shown in Figure 31.

```

BLOAD C0ROM,A$1000,D1
BLOAD D0ROM,A$2000
BSAVE SW16GCR.CD.ROM,A$1000,L$2000,D2,B
BLOAD E0ROM,A$3000,D1
BLOAD F0ROM,A$4002
BSAVE SW16GCR.EF.ROM,A$3000,L$2000,D2,B
BSAVE SW16GCR.CF.ROM,A$1000,L$4000,B

```

Figure 30. BLDROM EXEC File

```

BLOAD ZEROPAGE,A$1000,D2
BLOAD ZEROPAGE,A$1100
BLOAD ZEROPAGE,A$1200
BLOAD PAGE3,A$1300
BLOAD ZEROPAGE,A$1400
BLOAD ZEROPAGE,A$1500
BLOAD PAGE6,A$1600
BLOAD ZEROPAGE,A$1700
BLOAD ZEROPAGE,A$1800
BLOAD ZEROPAGE,A$1900
BLOAD ZEROPAGE,A$1A00
BLOAD ZEROPAGE,A$1B00
BLOAD ZEROPAGE,A$1C00
BLOAD ZEROPAGE,A$1D00
BLOAD ZEROPAGE,A$1E00
BLOAD ZEROPAGE,A$1F00
BLOAD D0ROM,A$2000,D1
BLOAD E0ROM,A$3000
BLOAD F0ROM,A$4002
BLOAD C0ROM,A$5000
BLOAD D0ROM,A$6000
BLOAD E0ROM,A$7000
BLOAD F0ROM,A$8002
BSAVE APPLE2E.SW16GCR.14.ROM,A$1000,L$8000,D2

```

Figure 31. BLDV2ROM EXEC File

PAGE3 is a binary file and this file contains 256 bytes of 6502 instructions that reside at 0xC300:C3FF in the Apple //e. PAGE6 is also a binary file and this file contains 256 bytes of 6502 instructions that reside at 0xC600:C6FF in the Apple //e. ZEROPAGE is a binary file that contains 256 bytes of zero. I do not believe that the HELP pages for Virtual][specifies the layout nor the precise content of a Virtual][software ROM file for the emulation of an Apple //e. I believe that I constructed a Virtual][software ROM file simply by inspecting the content of various software ROM files that were provided with earlier versions of the Virtual

]] application many years ago. The BLDV2ROM EXEC file constructs a Virtual]] software ROM file for the emulation of an Apple //e and it uses all of the components that are shown in Figure 31. The BLDV2ROM EXEC file generates the APPLE2E.SW16GCR.14.ROM file in the Virtual]] ROM 14 Build Volume. The APPLE2E.SW16GCR.14.ROM file must be copied from the Virtual]] ROM 14 Build Volume and into the file system of the Apple MacBook Pro. Once this software ROM file is in the Apple MacBook Pro file system, it can be copied to /Users/<user>/Library/"Application Support"/"Virtual]]/ROM where <user> is the Apple MacBook Pro user account name. Apple MacBook Pro users may find that their Library directory is hidden, so that volume must be unhidden in order to utilize the cleanup command file that is found in the SW16GCR.14.20250412 directory.

```
xterm -geometry 116x32+5+5 -fa Monaco -fs 10 &  
xterm -geometry 104x32+950+5 -fa Monaco -fs 10 &  
xterm -geometry 116x20+5+610 -fa Monaco -fs 10 &  
xterm -geometry 104x20+950+610 -fa Monaco -fs 10 &
```

Figure 32. SETUP Command File

I prefer to use the XQuartz environment when I develop C language programs on my Apple MacBook Pro computer. The first command that I issue in the XQuartz window is the `csch` command in order to begin the C shell or the `tcsh` environment in that window. I am sure individuals have their own preference of the shell that they prefer to use, and that is just fine with me. However, I have been using the C shell for nearly my entire programming career and it is the shell that I prefer and it is the shell that works best with my C language software products. I encourage everyone to become acquainted with the C shell only for the purpose of exploring my unique software products. You should certainly return to the shell of your choice. Perhaps, you could even transpose my software products into your favorite shell environment. The C shell environment allows me to process an `alias` file in order to have various aliases available to me while in this window environment. The next command that I issue is the `setup` command in order to create and have available to me several `xterm` windows. Figure 32 shows an example `setup` command file. In any of these `xterm` windows, `cd` to a convenient location where the contents of the `ModSoft.tar` file can be extracted. These contents include the `appleV2code` directory, the *My Applesoft Journey.pdf* file, and the `ROM2E.14` directory. In order to extract the contents of the `ModSoft.tar` file, the `tar xvf ModSoft.tar` command can be used. The `appleV2code` directory contains all of the tools that are necessary in order to read the `2eRoms/SW16GCR.14.20250412/ROM2E.SW16GCR.14.Build.dsk` file that resides in the `appleV2code` directory. After a successful assembly and build of the `ROM2E.SW16GCR.14.Image` disk image from the `ROM2E.SW16GCR.14.Source` disk image is made by *Lisa*, the EXEC files on the `ROM2E.SW16GCR.14.Build.dsk` disk image can be processed and that disk image file can be copied into the `appleV2code/2eRoms/SW16GCR.14.20250412` directory. The `cleanup` command file that is found in the `SW16GCR.14.20250412` directory deletes all of the unnecessary files that are copied from the Virtual]] ROM 14 Build Volume. It also copies the `APPLE2E.SW16GCR.14.ROM` file to the Virtual]] ROM directory so that the Virtual]] application can utilize that software ROM file in order to initialize the ROM environment of the emulated Apple //e.

The C language programming environment that is available on the Apple MacBook Pro is extensive. However, the Apple MacBook Pro programming environment is somewhat different than what is typically found on a standard UNIX platform. I have designed the menu command file that is found in the `appleV2code` directory to initialize the necessary environment variables so that menu will correctly execute within the programming environments with which I am acquainted. Darwin is one of those environments that menu knows about. In order to modify any of the source code files that are found in the source directory or to add new source code files to the source directory, it is important to first execute the command `run.config`. Now, the environment variable `HOME_PATH` is correctly initialized and the `makefile` in the source directory and the `makefile` in the binary directory will operate correctly. Of course, if any source code files are added to the source directory or removed from the source directory, the `makefile` in the source directory and the `makefile` in the binary directory will need to be modified in order to generate the object code files in the source directory and the executable files in the binary directory. If any source code file is modified in the source directory, the `makefile` in the source directory needs to execute and the `makefile` in the binary directory needs to execute. If you wish to begin with a fresh start and compile and link all of the source code files, simply enter the command `make clean` and then enter the command `make` first in the source directory and then `make` in the binary directory. Hopefully, no compile or link errors should occur.

The final step in utilizing a new software ROM in Virtual][on an Apple MacBook Pro computer is to have Virtual][correctly load that software ROM. Start the Virtual][application and press `Reset` in the upper right-hand corner. Select the `Machine/Configure` tab. On the left-hand side, open the `Components` selection and select `ROM memory`. On the right-hand side, press the `Select...` button. From the MacBook Pro file system display window, select the desired software ROM. Changing the software ROM always requires restarting the virtual machine.

Appendix A

The following table lists all of the page-zero variables that are used by Applesoft, by the ROM Monitor, and by DOS 4.5.08H.

Address	Applesoft	Other	Monitor	DOS 4.5.08	Description
00	GOWARM	R0L	LOC0		00:03, JMP RESTART
01	LOC1	R0H	LOC1		
02	LOC2				
03	GOSTROUT				03:05, JMP STROUT
04	:				
05	:				
06					Free
07					Free
08					Free
09					Free
0A	GOUSR				0A:0C, JMP <USER address>
0B	:				
0C	:				
0D	BYTVALUE	CHARAC			CHARAC
0E	ENDCHR				
0F	EOLPTR	NUMDIM			TOKNCNTR
10	DIMFLG				Dimension flag
11	VALTY8P				
12					Free
13	DATAFLG	GARFLG			
14	SUBFLG				Subscript flag
15	INPUTFLG				
16	CPRMASK	TOGLFLG			
17					Free
18		R12L			Free (SWEET16 STACK Pointer)
19		R12H			Free
1A	SHAPE				1A:1B
1B	:				
1C	COLBITS	R14L			
1D	COLCOUNT	R14H			
1E		R15L			Free
1F		R15H			Free
20			WNDLFT		Left window column
21			WNDWDTH		Window width
22			WNDTOP		Top window line
23			WNCBTM		Bottom window line
24			CH		Horizontal cursor position

25			CV		Vertical cursor position
26		TEMPZ	GBASL	BUFRADRZ	Graphic plot base address
27		TEMP2Z	GBASH	:	
28			BASL	BASEZ	Window base address
29			BASH	:	
2A		ASPTRSAV	BAS2L	CURTRKZ	Scrolling base address
2B			BAS2H	SLOT16Z	
2C		LMNEM	H2	DRVFLAG	ADRDATMK, ADRFIELD
2D		RMNEM	V2	SECFNDZ	
2E		FORMAT	MASK	TRKFNDZ	CHKSUM
2F		LASTIN	LENGTH	VOLFNDZ	SIGN
30	COLOR		HMASK		HIRES mask, LORES color
31			MODE		Command processing
32			INVFLG		Video format control
33			PROMPT		Prompt character
34			YSAV	PHASE	Command processing
35		SYNCNT	YSAV1	PAGECNT	SAVXYREG, CMDINDXZ
36			CSWL		Character output
37			CSWH		
38			KSWL		Character input
39			KSWH		
3A			PCL		Program counter
3B			PCH		
3C		MOTORTIM	A1L	ROMTEMPZ	Pointer #1
3D			A1H	ROMSECTR	
3E		ODDBITSZ	A2L	BUFADR2Z	Pointer #2
3F			A2H	SECTORZ	
40		ROMDATA	A3L	TRACKZ	Pointer #3; FILEBUFZ 40:41
41		ROMTRACK	A3H	VOLUMEZ	
42			A4L	BUFADRZ	Pointer #4; 42:43
43			A4H		
44	MACSTAT	A5L	OPRND	DIRINDX	General value
45	T2GUARD		AREG		A-register value
46			XREG		X-register value
47			YREG		Y-register value
48			PREG		P-register value
49			SPNT		STACK pointer value
4A				IOBADR	4A:4B
4B				:	
4C				DOSPTR	4C:4D
4D				:	
4E	RNDL				Random number
4F	RNDH				
50	LINNUM		ACL	LINNUM	50:51
51	:		ACH		
52	TEMPPT				Temporary string index
53	LASTPT				53:54, Last string pointer

54	:				
55	TEMPST				55:5D, String scratch name/len
56	:				
57	:				
58	:				
59	:				
5A	:			DOSTEMP1	
5B	:			DOSTEMP2	
5C	:			DOSBUFR	5C:5D
5D	:			:	
5E	INDEX				5E:5F, Move string stack
5F	:				
60	DEST				60:61
61	OFFSET				Trick assembler in SHFTBYT1, SHFTBYT2
62	MULMANT				62:65, Multiply/Divide result
63	:				
64	:				
65	:				
66	MULGUARD				
67	PRGTAB				67:68, Program start
68	:				
69	VARTAB				69:6A, Simple variables
6A	:				
6B	ARYTAB				6B:6C, Array variables
6C	:				
6D	STREND				6D:6E, Array end
6E	:				
6F	FRETOP				6F:70, String variables
70	:				
71	FRESPC				71:72, String variables
72	:				
73	MEMSIZE			HIMEM	73:74, Top of memory
74	:				
75	CURLIN				75:76, Line being interpreted
76	:			ASRUN	RUN flag
77	OLDLIN				77:78, Last interpreted line
78	:				
79	TEXTPTR				79:7A, Current TEXT pointer
7A	:				
7B	DATLIN				7B:7C, Line containing data
7C	:				
7D	DATPTR				7D:7E, Absolute data location
7E	:				
7F	SRCPTR				7F:80, Current input source
80	:				
81	VARNAM				81:82, Last variable's name
82	:				

83	VARPTR				83:84, Last variable's value
84	:				
85	FORPTR				85:86, General pointer
86	:				
87	TXPTRSAV	LASTOP			87:88
88	:				
89	CPRTYPE				
8A	FUNCNAM	TEMP3			8A:8B; 8A:8E, FP Register #3
8B	:	:			
8C	DSCPTR	:			8C:8D
8D	:	:			
8E		:			
8F		T3GUARD			T3 guard byte
90	JMPADRS				GETS, JMP <address>
91	RTNADR				
92	ARGGUARD				ARG guard byte
93		TEMP1			93:97, FP Register #1
94	ARYPNT	:			94:95, HIGHDS, LEN, Block trans
95	PROCESS	:			(GARBAG)
96	HIGHTR	:			96:97, Block transfer
97	:	:			
98		TEMP2			98:9C, FP Register #2
99	COUNTER	:			(FPOUT)
9A	EXPCOUNT	:			(FPOUT), (GETINT)
9B	LOWTR	:			9B:9C, DPFLAG (GETINT)
9C	EXPSIGN	:			(GETINT)
9D	DSCTMP				9D:9F, FACEXP, FAC exponent
9E	FACMANT				9E:A1, FAC mantissa
9F	:				
A0	:	VARPTR			A0:A1, Variable address pointer
A1	:	:			
A2	FACSIGN				FAC sign bit
A3	COEFNUM	MINUSLOC			
A4	EXTSIGN				SIGNEXT, Additional sign bit
A5	ARGEXP				ARG exponent
A6	ARGMANT				A6:A9, ARG mantissa
A7	:				
A8	:				
A9	:				
AA	ARGSIGN				ARG sign bit
AB	XORSIGN	STRING1			ARG^FAC sign bit, AB:AC
AC	FACGUARD	:			FAC guard byte
AD	COEFPTR	STRING2			AD:AE, SAVY (FPOUT)
AE		:			
AF	PRGEN				AF:B0, Program end
B0	:				
B1	CHRGET				B1:C8, Get next character

B2	:				
B3	:				
B4	:				
B5	:				
B6	:				
B7	CHRGOT				Get current character
B8	TXTPTR				B8:B9, Current character pointer
B9	:				
BA	:				
BB	:				
BC	:				
BD	:				
BE	:				
BF	:				
C0	:				
C1	:				
C2	:				
C3	:				
C4	:				
C5	:				
C6	:				
C7	:				
C8	:				
C9	IRAND				C9:CC, Random number seed
CA	:				
CB	:				
CC	:				
CD	SIGNFLG	SPCLFLAG			(TAN); (GARBAG)
CE					Free
CF					Free
D0	HRXDELTA	SHPVAL			D0:D1 HIRES pointer
D1	:	ROTQVAL			ROTQVAL
D2	HRYDELTA	ROTHVAL			ROTHVAL
D3	HRFLAG	ROTVVAL			D3:D4, ROTVVAL
D4	HRWORK	ROTHSUM			ROTHSUM
D5	HRYEND	ROTVSUM			ROTVSUM
D6	RUNFLAG			PROTECT	Used for RUN command
D7		SHPOLD		SUBCODEZ	SUBCODE page-zero value
D8	ERRFLG			ASONERR	ONERR flag
D9				RKEYWORD	
DA	ERRLIN				DA:DB, Line containing error
DB	:				
DC	ERRPOS				DC:DD, TEXTPTR save for HNDLERR
DD	:				
DE	ERRNUM				Error number or code
DF	ERRSTK				STACK pointer before error
E0	HRXCOORD				E0:E1, HIRES X-coordinate

E1	:				
E2	HRYCOORD				HIRES Y-coordinate
E3					Free
E4	HRCOLOR				HIRES color byte
E5	HRHORZ				HIRES horizontal byte index
E6	HRPAG				HIRES active page (0x20 or 0x40)
E7	HRSCALE				HIRES scale factor
E8	HRSHPTBL				E8:E9, HIRES Shape Table address
E9	:				
EA	HRCOLCNT				HIRES collision counter
EB					Free
EC					Free
ED					Free
EE					Free
EF					Free
F0	FIRST				LORES plot destination
F1	SPEEDBYT				Display speed control, 0x00:FF
F2	TRACEFLG				
F3	FLASHBYT				Output character control mask
F4	TXTPTRSV				F4:F5, ONERR TEXT pointer save
F5	:				
F6	CURLINSV				F6:F7, ONERR line pointer save
F7	:				
F8	REMSTK				
F9	HRROT				HIRES shape rotation factor
FA					Free
FB					Free
FC					Free
FD					Free
FE					Free
FF					Free

Table A.1. Page-Zero Definitions

Appendix B

The following table lists all of the Applesoft statements, their token number, and the location in Applesoft where that statement is processed.

Statement	Token	Address	Description
END	0x80	0xD870	See 0xD000, adr BEND-1; start of BASIC statements
FOR	0x81	0xD766	See 0xD002, adr BFOR-1
NEXT	0x82	0xDCf9	See 0xD004, adr BNEXT-1
DATA	0x83	0xD995	See 0xD006, adr BDATA-1
INPUT	0x84	0xDBB2	See 0xD008, adr BINPUT-1
DEL	0x85	0xF331	See 0xD00A, adr BDEL-1
DIM	0x86	0xDFD9	See 0xD00C, adr BDIM-1
READ	0x87	0xDBE2	See 0xD00E, adr BREAD-1
GR	0x88	0xF390	See 0xD010, adr BGR-1
TEXT	0x89	0xF399	See 0xD012, adr BTEXT-1
PR#	0x8A	0xF1E5	See 0xD014, adr BPR-1
IN#	0x8B	0xF1DE	See 0xD016, adr BIN-1
CALL	0x8C	0xF1D5	See 0xD018, adr BCALL-1
PLOT	0x8D	0xF225	See 0xD01A, adr BCALL-1
HLIN	0x8E	0xF232	See 0xD01C, adr BHLIN-1
VLIN	0x8F	0xF241	See 0xD01E, adr BVLIN-1
HGR2	0x90	0xF3D8	See 0xD020, adr BHGR2-1
HGR	0x91	0xF3E2	See 0xD022, adr BHGR-1
HCOLOR=	0x92	0xF6E9	See 0xD024, adr HCOLOR-1
HPlot	0x93	0xF6FE	See 0xD026, adr BHPlot-1
DRAW	0x94	0xF769	See 0xD028, adr BDRAW-1
XDRAW	0x95	0xF76F	See 0xD02A, adr BXDRAW-1
HTAB	0x96	0xF7E7	See 0xD02C, adr BHTAB-1
HOME	0x97	0xFC58	See 0xD02E, adr HOME-1
ROT=	0x98	0xF721	See 0xD030, adr BROT-1
SCALE=	0x99	0xF727	See 0xD032, adr BSCALE-1
SHLOAD	0x9A	0xFF58	See 0xD034, adr IORTS-1; command removed
TRACE	0x9B	0xF26D	See 0xD036, adr BTRACE-1
NOTRACE	0x9C	0xF26F	See 0xD038, adr BNOTRACE-1
NORMAL	0x9D	0xF273	See 0xD03A, adr BNORMAL-1
INVERSE	0x9E	0xF277	See 0xD03C, adr BINVERSE-1
FLASH	0x9F	0xF280	See 0xD03E, adr BFLASH-1
COLOR=	0xA0	0xF24F	See 0xD040, adr BCOLOR-1
POP	0xA1	0xD96B	See 0xD042, adr BPOP-1
VTAB	0xA2	0xF256	See 0xD044, adr BVTAB-1
HIMEM:	0xA3	0xF286	See 0xD046, adr BHIMEM-1
LOMEM:	0xA4	0xF2A6	See 0xD048, adr BLOMEM-1

ONERR	0xA5	0xF2CB	See 0xD04A, adr BONERR-1
RESUME	0xA6	0xF318	See 0xD04C, adr BRESUME-1
RECALL	0xA7	0xFF58	See 0xD04E, adr IORTS-1; command removed
STORE	0xA8	0xFF58	See 0xD050, adr IORTS-1; command removed
SPEED=	0xA9	0xF262	See 0xD052, adr BSPEED-1
LET	0xAA	0xDA46	See 0xD054, adr BLET-1
GOTO	0xAB	0xD93E	See 0xD056, adr BGOTO-1
RUN	0xAC	0xD912	See 0xD058, adr BRUN-1
IF	0xAD	0xD9C9	See 0xD05A, adr BIF-1
RESTORE	0xAE	0xD849	See 0xD05C, adr BRESTORE-1
&	0xAF	0x03F5	See 0xD05E, adr USRAHAND-1; connected to USRAHAND
GOSUB	0xB0	0xD921	See 0xD060, adr BGOSUB-1
RETURN	0xB1	0xD96B	See 0xD062, adr BRETURN-1
REM	0xB2	0xD9DC	See 0xD064, adr BREM-1
STOP	0xB3	0xD86E	See 0xD066, adr BSTOP-1
ON	0xB4	0xD9EC	See 0xD068, adr BON-1
WAIT	0xB5	0xE784	See 0xD06A, adr BWAIT-1
LOAD	0xB6	0xD8DC	See 0xD06C, adr BLOAD-1
SAVE	0xB7	0xFF58	See 0xD06E, adr IORTS-1; command removed
DEF	0xB8	0xE313	See 0xD070, adr BDEF-1
POKE	0xB9	0xE77B	See 0xD072, adr BPOKE-1
PRINT	0xBA	0xDAD5	See 0xD074, adr BPRINT-1
CONT	0xBB	0xD896	See 0xD076, adr BCONT-1
LIST	0xBC	0xD6A5	See 0xD078, adr BLIST-1
CLEAR	0xBD	0xD66A	See 0xD07A, adr BCLEAR-1
GET	0xBE	0xDBA0	See 0xD07C, adr BGET-1
NEW	0xBF	0xD649	See 0xD07E, adr BNEW-1
TAB(0xC0		TK.TAB; referenced directly
TO	0xC1		TK.TO; referenced directly
FN	0xC2		TK.FN; referenced directly
SPC(0xC3		TK.SPC; referenced directly
THEN	0xC4		TK.THEN; referenced directly
AT	0xC5		TK.AT; referenced directly
NOT	0xC6		TK.NOT; referenced directly
STEP	0xC7		TK.STEP; referenced directly
+	0xC8	0xE7C1	TK.PLUS, OPLUS; TAG = 0x79; referenced directly
-	0xC9	0xE7AA	TK.MINUS, OMINUS; TAG = 0x79; referenced directly
*	0xCA	0xE982	OMULT; TAG = 0x7B; used directly
/	0xCB	0xEA69	ODIVIDE; TAG = 0x7B; used directly
^	0xCC	0xEE97	OPOWER; TAG = 0x7D; used directly
AND	0xCD	0xDF55	OAND; TAG = 0x50; used directly
OR	0xCE	0xDF4F	OOR; TAG = 0x46; used directly
>	0xCF	0xEED0	TK.GRTR; TAG = 0x7F; referenced directly
=	0xD0	0xDE9B	TK.EQUAL; TAG = 0x7F; referenced directly
<	0xD1	0xDF65	REL; TAG = 0x64; used directly
SGN	0xD2	0xEB90	See 0xD080, adr FSGN; start of FUNCTION1 statements
INT	0xD3	0xEC23	See 0xD082, adr FINT

ABS	0xD4	0xEBAF	See 0xD084, adr FABS
USR	0xD5	0x000A	See 0xD086, adr GOUSR
FRE	0xD6	0xE2DE	See 0xD088, adr FFRE
SCRN(0xD7	0xDEF9	See 0xD08A, adr FSCREEN
PDL	0xD8	0xDFCD	See 0xD08C, adr FPD
POS	0xD9	0xE2FF	See 0xD08E, adr FPOS
SQR	0xDA	0xEE8D	See 0xD090, adr FSQR
RND	0xDB	0xEFAE	See 0xD092, adr FRND
LOG	0xDC	0xEF3E	See 0xD094, adr FLOG
EXP	0xDD	0xEF09	See 0xD096, adr FEXP
COS	0xDE	0xEFEA	See 0xD098, adr FCOS
SIN	0xDF	0xEFF1	See 0xD09A, adr FSIN
TAN	0xE0	0xF03A	See 0xD09C, adr FTAN
ATN	0xE1	0xF09E	See 0xD09E, adr FATAN
PEEK	0xE2	0xE764	See 0xD0A0, adr FPEEK
LEN	0xE3	0xE6D6	See 0xD0A2, adr FLEN
STR\$	0xE4	0xE3C5	See 0xD0A4, adr FSTR
VAL	0xE5	0xE707	See 0xD0A6, adr FVAL
ASC	0xE6	0xE6E5	See 0xD0A8, adr FASC
CHR\$	0xE7	0xE646	See 0xD0AA, adr FCHR
LEFT\$	0xE8	0xE65A	See 0xD0AC, adr FLEFT; start of FUNCTION2 statements
RIGHT\$	0xE9	0xE686	See 0xD0AE, adr FRIGHT
MID\$	0xEA	0xE691	See 0xD0B0, adr FMID
PI	0xEB	0xEF48	See 0xD0B2, adr FPI; a BASIC statement
LN	0xEC	0xE941	See 0xD0B4, adr FLN; a FUNCTION1 statement

Table B.1. Modified Applesoft Statements

Appendix C

The following table lists all of the internal Applesoft entry points for the unmodified version of Applesoft under **Old Addr** and whether these entry points are the same or different in the modified version of Applesoft under **New Addr**. If the entry point addresses are different, the **Cng Flg** column is checked.

Cng Flg	Old Addr	New Addr	Name	Description
	-	0xC600	PROCVAR	Cornelis Bongers Garbage Collection Routine
	-	0xC64E	PROCSPCL	Cornelis Bongers Garbage Collection Routine
	-	0xC670	SW16	Sweet 16 Metaprocessor, revised original
	-	0xC7FF	SW16	End of revised Sweet 16 Metaprocessor
	-	0xCA71	OPTBLC	65C02 MNEML/MNEMR remapping
	-	0xCA7D	OPTBLL	65C02 MNEML/MNEMR remapping
	0xD000	0xD000	BASADDR	BASIC statements addresses = #ADDR/2 + 0x80
	0xD080	0xD080	FN1ADDR	FUNCTION statements addr = #ADDR/2 + 0x92
	-	0xD0B2	FS3LN	LN routine address, statement number = 0xEB
	-	0xD0B4	FS3PI	PI routine address, statement number = 0xEC
✓	0xD0B2	0xD0B6	TAGADDR	OPERATOR statements addr
✓	0xD0D0	0xD0D4	BASNAME	Statement names in DCI format
✓	0xD260	0xD25B	MESGS	Error messages in mixed case in DCI format
✓	0xD365	0xD362	GTFORPNT	Used by FOR/NEXT, accelerated
	0xD393	0xD393	BLTU	Block transfer utility
	0xD3D6	0xD3D6	CKSTKSIZ	Check STACK size
	0xD3E3	0xD3E3	CKSTRSIZ	Check memory size between arrays and strings
	0xD410	0xD410	OM.ERR	Out of Memory error entry
	0xD412	0xD412	PRterr	Print selected error message
	0xD431	0xD431	PRLINUM	Print string at (A/Y) using INDEX; modified
	0xD43C	0xD43C	RESTART	Default DOS restart WARMADR entry ASROMWRM
	0xD4F2	0xD4F2	ASENTER	Default DOS reset RESETADR entry ASROMRST
	0xD52C	0xD52C	INLIN	Read INPUT line, clear all MSBs; accelerated
✓	0xD553	-	INCHR	Removed INCHR as unnecessary
	0xD559	0xD559	PARSINPT	Parse and tokenize the INPUT line
	0xD56C	0xD56C	PARSE	Get next input character
	0xD61A	0xD61A	FNDLIN	Search for line number in (LINNUM)
	0xD649	0xD649	BNEW	Implement the NEW statement
	0xD64B	0xD64B	SCRtCH	Initialize for a new program environment
	0xD665	0xD665	SETPTRS	Default DOS RUN/CHAIN entry ASROMCLR
	0xD66A	0xD66A	BCLEAR	Implement the CLEAR statement
	0xD66C	0xD66C	CLEARC	Clear string area
	0xD683	0xD683	STKINIT	Start STACK at 0xF8
	0xD697	0xD697	STXTPTR	Initialize TXTPTR to program beginning
	0xD6A5	0xD6A5	BLIST	Implement the LIST statement
✓	0xD72C	0xD758	GETCHR	Get next character using (DSCTMP)

	0xD766	0xD766	BFOR	Implement the FOR/NEXT/STEP statements
	0xD7AF	0xD7AF	STEP	STEP phrase in FOR statement
	0xD7D2	0xD7D2	NEWSTT	Default DOS RUN/CHAIN entry ASROMNEW
	0xD805	0xD805	DOTRACE	Enable or disable program tracing
	0xD828	0xD828	DOSTAMT	Execute a statement; BASADDR or FN1ADDR
	0xD849	0xD849	BRESTORE	Implement the RESTORE statement
	0xD853	0xD853	SETDAPTR	Set DATPTR to (A/Y)
	0xD858	0xD858	ISCNTLC	Handle control-C; accelerated
	0xD865	0xD865	ASROMERR	Default DOS error ERRORADR; accelerated
	0xD86E	0xD86E	BSTOP	Implement the STOP statement
	0xD870	0xD870	BEND	Implement the END statement
	0xD896	0xD896	BCONT	Implement the CONT statement
✓	0xD8B0	-	SAVE	Removed this statement
✓	-	0xD8B0	DOHANDLR	Jump to HANDLERR
✓	0xD9C5	0xD8B3	PULL3A	Issue 3 pla instructions
✓	-	0xD8BB	RDBYTE	Used by CXREAD to read audio waveform
	0xD8C9	0xD8C9	BLOAD	Implement the LOAD statement
✓	-	0xD8FF	RD2BIT	Read two audio waveform transitions
✓	-	0xD902	RDBIT	Read one audio waveform transition
	0xD912	0xD912	BRUN	Implement the RUN statement
	0xD921	0xD921	BGOSUB	Implement the GOSUB statement
	0xD93E	0xD93E	BGOTO	Implement the GOTO statement
	0xD955	0xD955	ASROMSET	Default DOS RUN/CHAIN LINNUM initialization
	0xD96B	0xD96B	BPOP	Implement the POP statement; mod
	0xD96B	0xD96B	BRETURN	Implement the RETURN statement; mod
	0xD97C	0xD97C	US.ERR	Undefined Statement error entry
	0xD995	0xD995	BDATA	Implement the DATA statement
	0xD9A3	0xD9A3	DATSCAN	Scan ahead to next “:” or End of Line (EOL)
	0xD9C9	0xD9C9	BIF	Implement the IF statement
	0xD9DC	0xD9DC	BREM	Implement the REM statement
	0xD9EC	0xD9EC	BON	Implement the ON statement
	0xDA0C	0xDA0C	LINGET	Convert line number; repaired
	0xDA46	0xDA46	BLET	Implement LET statement
	0xDA7B	0xDA7B	PUTSTR	Install string descriptor address
	0xDAB7	0xDAB7	COPYSTR	Copy string into Character String Pool
	0xDACF	0xDACF	PRSTRING	Print string and get last character
	0xDAD5	0xDAD5	BPRINT	Implement the PRINT statement; accelerated
	-	0xDB32	UNARY2	Complete UNARY processing
✓	-	0xDB38	LINEOUT	Print number
✓	0xDB3A	0xDB3B	STROUT	Print string at (A/Y)
✓	0xDB3D	0xDB3E	STRPRT	Print string at (INDEX); accelerated
✓	0xDAFB	0xDB50	PRTCR	Print return character; repaired
✓	0xDB57	0xDB53	OUTSPC	Print space character
✓	0xDB5A	0xDB56	OUTPROMT	Print prompt character ‘>’ and not ‘?’
✓	0xDB5C	0xDB58	OUTCHR	Print character
✓	0xDB71	0xDB6F	INPUTERR	Also, READERR, ERRLINN, INPERR, RESPERR; mod
✓	0xDB7B	0xDB79	READERR	Gets the data location, not the data

✓	0xDB7F	0xDB7D	ERRLINN	Save data location
✓	0xDEC9	0xDB81	SY.ERR3	Syntax error entry
	0xDB86	0xDB86	INPERR	Pull data from the STACK
	0xDB87	0xDB87	RESPERR	Checks ONERR flag and handles both states
	0xDBA0	0xDBA0	BGET	Implement the GET statement
	0xDBB2	0xDBB2	BINPUT	Implement the INPUT statement; accelerated
	0xDBDC	0xDBDC	HEXTIN	Print PROMPT and input line
	0xDBE2	0xDBE2	BREAD	Implement the READ statement
	0xDBEB	0xDBEB	INPTLIST	Process input list
	0xDBF1	0xDBF1	INPTITEM	Process input item
	0xDC2B	0xDC2B	INSTART	Input the string or numeric data
	0xDC99	0xDC99	INPTFLG	Select INPUT or READ
	0xDCA0	0xDCA0	FINDATA	Locate TEXT data, colon, or End of Line
✓	0xDCC6	0xDCC7	INPTDONE	No more data requested
	0xDCDF	0xDCDF	MESG21	Message '>Extra Ignored', 0x0D
	0xDCEF	0xDCEF	MESG22	Message '>Reenter', 0x0D
	0xDCF9	0xDCF9	BNEXT	Implement the NEXT statement
✓	-	0xDD4D	FPCOMPT3	Initialize X-reg with T3GUARD for FPCOMP
✓	0xDD67	0xDD64	FRMNUM	Evaluate expression
✓	0xDD6A	0xDD67	CHKNUM	Make sure FAC is a number
✓	0xDD6C	0xDD69	CHKSTR	Make sure FAC is a string
✓	0xDD6D	0xDD6A	CHKVAL	Verify FAC is correct number type
✓	0xDD76	0xDD73	TM.ERR	Type Mismatch error entry
✓	0xDD0B	0xDD76	NF.ERR	NEXT without FOR error entry
	0xDD7B	0xDD7B	FRMEVAL	Evaluate expression at (TXTPTR)
	0xDDD7	0xDDD7	SAVOP	Call FREMEVAL recursively
	0xDDFD	0xDDFD	FRMRECUR	Use STACK/recursion to evaluate expression
	0xDE10	0xDE10	FRMSTAK	Get FACSIGN and precedence value
	0xDE15	0xDE15	FRMSTAK2	Pull return address, correctly increment
✓	0xDE20	0xDE23	FRMSTAK3	No RNDUP; push FACGUARD and FAC onto STACK
✓	0xDE35	0xDE32	NOTMATH	Setup the EXIT routine
✓	0xDE43	0xDE40	NOTMATH4	Pull ARG, ARGGUARD, and ARGSIGN from STACK
	0xDE60	0xDE60	FRMELMNT	Get array element number in expression
	0xDE81	0xDE81	STRTXT	Get first element string
	0xDE90	0xDE90	NOTFUNC	Evaluate NOT token
	0xDE98	0xDE98	OEQUAL	Implement the EQUAL operator; accelerated
✓	0xDEA4	0xDEA7	FNFUNC	Evaluate FN token; moved
✓	0xDEAB	0xDEAE	SGNFUNC	Evaluate SGN token; accelerated
	0xDEB2	0xDEB2	PARENCHK	Check open parenthesis, evaluate expression
	0xDEB8	0xDEB8	CHKCLSP	Check for closed parenthesis
	0xDEBB	0xDEBB	CHKOPNP	Check for open parenthesis
	0xDEBE	0xDEBE	CHKCOM	Check for comma
	0xDEC0	0xDEC0	SYNTAXCHK	Syntax routine
	0xDEC9	0xDEC9	SY.ERR	Syntax error entry
	0xDECE	0xDECE	MINUFUNC	Minus function entry
	0xDED0	0xDED0	EQUFUNC	Equal function entry
	0xDED5	0xDED5	GETIVAL	Input the string or numeric data

	0xDEF9	0xDEF9	FSCREEN	Implement the SCRN(function
✓	0xDF0C	0xDF09	UNARY	Modified to add LN and PI statements
	0xDF4F	0xDF4F	OOR	Handle OR operator
	0xDF55	0xDF55	OAND	Handle AND operator
	0xDF5D	0xDF5D	FALSE	Return FAC = 0
	0xDF60	0xDF60	TRUE	Return FAC = 1
	0xDF65	0xDF65	OLT	Perform relational operations
	0xDF7D	0xDF7D	STRCMP	String comparison function
	0xDFB0	0xDFB0	NUMCMP	Number comparison result
	0xDFCD	0xDFCD	FPDL	Implement the PDL statement; modified
	0xDFD9	0xDFD9	BDIM	Implement the DIM statement
	0xDFE3	0xDFE3	PTRGET	General variable scan, DIMFLG & SUBFLG; mod
	0xE000	0xE000	BASIC	COLDSTRT entry point
	0xE003	0xE003	BASIC2	RESTART entry point
✓	0xE07D	0xE0DA	CHKASCII	Set C-flag only for A-Z or clear; modified
✓	0xE0ED	0xE0E3	PNTARVAL	Compute address for the first array value
✓	-	0xE0FF	STRSETUP	Three byte patch for LEFT\$/RIGHT\$/MID\$
✓	0xE09A	0xE105	IVALZERO	16-bit integer value for zero
✓	0xE0FE	0xE107	FP8000	16-bit integer value for 32768; corrected
✓	0xE102	0xE10C	MAKINT	Evaluate expression, convert to integer
✓	0xE108	0xE112	AYPOSINT	Convert positive number to integer
✓	0xE10C	0xE116	AYINT	Convert to signed integer
✓	0XE11E	0xE128	ARRAY	Locate array element, create array; modified
✓	0xE196	0xE198	BS.ERR	Bad Subscript error entry
✓	0xE199	0xE19B	IQ.ERR	Illegal Quantity error entry
	0xE19E	0xE19E	RA.ERR	ReDIM'd Array error entry, modified
✓	0xE1BC	0xE1A1	OD.ERR	Out of Data error entry
	0xE24B	0xE24B	FINDELE	Find specified array element
	0xE2AD	0xE2AD	MULSUBS	Multiply array subscripts
	0xE2DE	0xE2DE	FFRE	Implement the FRE statement, calls GARBAG
	0xE2F2	0xE2F2	GIVAYFP	Float the signed integer in (A/Y)
	0xE2FF	0xE2FF	FPOS	Returns the current line position in (CH)
	0xE301	0xE301	SNGFLT	Float Y-register into FAC
	0xE306	0xE306	CHKIFDIR	Check MODE for direct or running
	0xE30E	0xE30E	UF.ERR	Undefined Function error entry
	0xE313	0xE313	BDEF	Implement the DEF statement
	0xE341	0xE341	GETFNC	Common routine for DEF FN and FN statements
	0xE354	0xE354	CALLFNC	Process the FN statement
	0xE3AF	0xE3AF	FNCDATA	Retrieve five bytes from STACK by (FUNCNAM)
	0xE3C5	0xE3C5	FSTR	Implement the STR\$ statement; modified
✓	0xE3D5	0xE3D0	STRINI	Get space for string descriptor in (FAC)
✓	0xE3DD	0xE3D8	STRSPA	Get space for string descriptor in (X/Y)
✓	0xE3E7	0xE3E2	STRLIT	Build a string descriptor in (A/Y)
✓	0xE42A	0xE426	PUTNEW	Store descriptor as a temporary descriptor
✓	0xE430	0xE449	FC.ERR	Formula too Complex error entry
✓	0xE474	0xE44C	OM.ERR3	Out of Memory error entry
✓	0xE452	0xE454	GETSSPC	Make string space in Character String Pool

	0xE484	0xE484	GARBAG	Implementation of C. Bongers GARBAG routine
	0xE597	0xE597	CAT2STR	Concatenate two strings
✓	0xE5B2	0xE5CF	SL.ERR	String too Long error entry
	0xE5D4	0xE5D4	MOVINS	Get string descriptor using (STRING1)
	0xE5E2	0xE5E2	MOVSTR	Move string at (X/Y) having length (A)
	0xE5FD	0xE5FD	FRESTR	Release descriptor
	0xE600	0xE600	FREFAC	Release temporary string
	0xE604	0xE604	FRETMP	Release (A/Y) string
	0xE635	0xE635	FRETMS	Free temporary descriptor
	0xE646	0xE646	FCHR	Implement the CHR\$ statement
	0xE65A	0xE65A	FLEFT	Implement the LEFT\$ statement
	0xE686	0xE686	FRIGHT	Implement the RIGHT\$ statement
	0xE691	0xE691	FMID	Implement the MID\$ statement, check for 0
✓	0xE6B9	0xE6BC	STRSET2	Continuation of STRSETUP patch
	0xE6D6	0xE6D6	FLEN	Implement the LEN statement
	0xE6DC	0xE6DC	GETSTRLN	Free string if temporary, return length
	0xE6E5	0xE6E5	FASC	Implement the ASC statement
	0xE6F5	0xE6F5	GETBYTC	Scan to next character and convert to byte
	0xE6F8	0xE6F8	GETBYT	Evaluate expression at (TXTPTR), return byte
	0xE6FB	0xE6FB	CONVINT	Convert (FAC) to single byte in X-register
	0xE707	0xE707	FVAL	Implement the VAL statement
	0xE73D	0xE73D	STRCOPY	Copy string from (STRING2) to (TXTPTR)
	0xE746	0xE746	GETASNUM	Evaluate expression for 16-bit value
	0xE74C	0xE74C	COMBYTE	Evaluate expression for 8-bit value
	0xE752	0xE752	GETADDR	Convert (FAC) to a 16-bit value
	0xE764	0xE764	FPEEK	Implement the PEEK statement
	0xE77B	0xE77B	BPOKE	Implement the POKE statement
	0xE784	0xE784	BWAIT	Implement the WAIT statement
✓	0xE7A0	-	FADDH	Removed add 0.5 to FAC function
✓	-	0xE7A1	FPPI	FP value for PI with guard byte value
	0xE7A7	0xE7A7	FSUB	Internal entry for subtraction, load guard
	0xE7AA	0xE7AA	OMINUS	Implement the - statement; modification
	0xE7BE	0xE7BE	FADD	Internal entry for addition, load guard byte
	0xE7C1	0xE7C1	OPLUS	Implement the + statement; modification
✓	0xE829	0xE81D	COMPFAC1	Normalize value in FAC
✓	0xE82E	0xE822	NORMFAC1	Shift bytes left in FAC, FACGUARD = 0
✓	0xE84E	0xE842	ZEROFAC	Clear FACEXP and FACSIGN to zero
✓	0xE874	0xE868	NORMFAC2	Shift FAC left and increment A-reg
✓	0xE89E	0xE892	COMPFAC2	Compliment FAC and then its mantissa
✓	0xE8C6	0xE8BA	INCMANT	Increment the FAC mantissa
✓	0xE8D5	0xE8C9	OF.ERR	Overflow error entry
✓	0xE8F0	0xE8CE	SHFTBYT1	Shift bytes to the right into FACGUARD
✓	0xE907	0xE8FC	SHFTBITS	Shift bits to the right into FACGUARD
✓	-	0xE908	PDL2	Ensure argument is 0:3 before call to PREAD
✓	-	0xE913	FPLOGE	FP value for LOG(e) to convert LN -> LOG
✓	0xE92D	0xE918	FPSQR0.5	FP value for the square root of 0.5
✓	0xE932	0xE91D	FPSQR2.0	FP value for the square root of 2.0

✓	0xE937	0xE922	FPN0.5	FP value for - 0.5
✓	0xE93C	0xE827	FPLN2	FP value for LN(2)
✓	0xE918	0xE92C	POLY.LOG	Coefficients to calculate natural log LN
	0xE941	0xE941	FLN	Implement the LN statement
	0xE97F	0xE97F	FMULT	Internal entry for multiplication, use guard
	0xE982	0xE982	OMULT	Implement the * statement; modification
	0xE9E3	0xE9E3	LOADARG	Load ARG register from (A/Y), ARGGUARD = 0
✓	0xEA0E	0xEA10	PROCEXP	Process exponents; modified
✓	0xEA2B	0xEA2E	ZEROFERR	Check for zero or overflow error
✓	0xEA36	0xEA32	OF.ERR2	Overflow error entry
✓	0xEAE1	0xEA35	DZ.ERR	Division by Zero error entry
✓	0xEA55	0xEA3A	MULFAC10	Multiply FAC by 10, no RNDUP, copy guards
✓	0xEA50	0xEA50	FP10.0	FP value for 10.0
✓	0xEA5E	0xEA55	DIVFAC10	Divide FAC by 10, no RNDUP; modified
	0xEA66	0xEA66	FDIV	Internal entry for division, use guard bytes
	0xEA69	0xEA69	ODIVIDE	Implement the / statement; modification
✓	0xF070	0xEAD7	FP.25	FP value for 0.25
✓	0xED17	0xEADC	FP1.0E9	FP value for 1.0E+09
	0xEAE6	0xEAE6	COPYM2F	Copy MULMANT to FACMANT; call NORMFAC1
	0xEAF9	0xEAF9	LOADFAC	Load FAC register from (A/Y), FACGUARD = 0
	0xEB1E	0xEB1E	COPYF2T2	Copy the FAC register to the TEMP2 register
	0xEB21	0xEB21	COPYF2T1	Copy the FAC register to the TEMP1 register
	0xEB27	0xEB27	COPYF2FR	Copy the FAC register to FORPNT
	0xEB2B	0xEB2B	COPYFAC	Copy the FAC register to (X/Y), use RNDUP
	0xEB53	0xEB53	COPYA2F	Copy ARG to the FAC register, copy guards
	0xEB63	0xEB63	COPYF2A	Copy FAC to the ARG register, copy guards
✓	0xEB72	0xEB70	RNDUP	Increment FAC if FACGUARD MSB set; modified
	0xEB82	0xEB82	SIGNCHK	Test FAC for negative, zero, positive
	0xEB90	0xEB90	FSGN	Implement the SGN statement
	0xFB93	0xEB93	FLOAT	Convert (A) to signed value of -128 to 127
	0xEBAF	0xEBAF	FABS	Implement the ABS statement
✓	0xEBB2	0xEBB5	FPCOMP	Compare FAC with (A/Y), use T3GUARD; mod
	0xEBF2	0xEBF2	FP2INT	Convert FAC to a 16-bit integer
	0xEC23	0xEC23	FINT	Convert FAC to integer, then refloat to FP
	0xEC40	0xEC40	CLRMANT	Clear FAC mantissa
	0xEC4A	0xEC4A	GETINT	Convert numerical string to FP in FAC; mod
✓	0xECD5	0xECD5	ADD2FAC	Add value in (A) to FAC; modified
✓	0xED19	0xED0A	PRTMSG19	Print ' in ', line number, CR; modified
✓	0xED24	0xED18	LINEPRT	Print (X/A) as a 16-bit integer
✓	0xD358	0xED25	MESG19	asc ' in '
✓	0xED0A	0xED2A	FP9.9E7	FP value for 9.9E07
✓	0xED0F	0xED2F	FP9.9E8	FP value for 9.9E08
	0xED34	0xED34	FPOUT	Print FAC as numerical string in STACK; mod
✓	0xEE6C	0xEE5C	FPDECTBL	Hex to decimal conversion table
	0xEE8D	0xEE8D	FSQR	Implement the SQR statement; modified
	0xEE97	0xEE97	OPOWER	Implement the ^ statement; modified
	0xEED0	0xEED0	OGT	Implement the > statement; also is NEGFAC

	0xEEDB	0xEEDB	FPINVLN2	FP value for 1/LN(2)
	0xEEEO	0xEEEO	POLY.EXP	Coefficients to calculate exponential
✓	0xE916	0xEF04	FP1.0	FP value for 1.0
	0xEF09	0xEF09	FEXP	Implement the EXP statement; modified
✓	0xEF72	0xEF31	POLYNOML	Save (A/Y) to (COEFPTR); moved
✓	-	0xEF3E	FLOG	Convert natural log LN to base-10 log LOG
✓	-	0xEF48	FPI	Load FAC and FACGUARD with the value of PI
✓	-	0xEF57	POLYSIN	Load (A/Y) with address of POLY.SIN coefs.
✓	0xEF5C	0xEF5B	POLYPROC	Odd polynomial processing, no RNDUP
✓	0xEF76	0xEF71	POLYNOM	Normal polynomial processing; modified
	0xEFA6	0xEFA6	RANDVAL1	Integer value A = 0x12B9B0A5; modified
	0xEFAA	0xEFAA	RANDVAL2	Integer value C = 0x361962EA; modified
	0xEFAE	0xEFAE	FRND	Implement the RND statement; modified
	0xEFEA	0xEFEA	FCOS	Implement the COS statement
	0xEFF1	0xEFF1	FSIN	Implement the SIN statement; modified
	0xF03A	0xF03A	FTAN	Implement the TAN statement; modified
✓	0xF066	0xF05C	FPIDIV2	FP value for PI/2
✓	0xEE67	0xF061	FP0.5	FP value for 0.5
✓	0xF075	0xF066	POLY.SIN	Coefficients to calculate SINE; modified
✓	0xF094	-	UNREFBYT	Unreferenced bytes for MICROSOFT! Backwards
✓	0xF06B	0xF099	FPIMUL2	FP value for 2*PI
	0xF09E	0xF09E	FATAN	Implement the ATN statement; accelerated
✓	0xF0CE	0xF0CC	POLY.ATN	Coefficients to calculate ARCTAN
	0xF10B	0xF109	PGZCODE	Routines that are copied to page-zero
	0xB1	0xB1	CHRGET	Increment TXTPTR and read ASCII character
	0xB7	0xB7	CHRGOT	Recall previous ASCII value read
	0xC9	0xC9	FPRAND	Random number generator seed
✓	0xF128	0xF125	COLDSTRT	BASIC enters to setup pointers and vectors
✓	-	0xF1B1	FRMSTAK4	Continuation of FRMSTAK3
✓	-	0xF1BA	COPYF2T3	Copy the FAC register to the TEMP3 register
✓	-	0xF1C5	INCCOEF	Increment the coefficient pointer
✓	-	0xF1CC	CLEARMUL	Clear MULMANT to 0x00 for FMULT
	0xF1D5	0xF1D5	BCALL	Implement the CALL statement
	0xF1DE	0xF1DE	BIN	Implement the IN# statement
	0xF1E5	0xF1E5	BPR	Implement the PR# statement
	0xF1EC	0xF1EC	PLOTFNS	Get LORES coordinates for H2 and V2 < 48
	0xF209	0xF209	LINC00R	Get A,B at C values for HLIN and VLIN
	0xF225	0xF225	BPLOT	Implement the PLOT statement
	0xF232	0xF232	BHLIN	Implement the HLIN statement
	0xF241	0xF241	BVLIN	Implement the VLIN statement
	0xF24F	0xF24F	BCOLOR	Implement the COLOR= statement
	0xF256	0xF256	BVTAB	Implement the VTAB statement
	0xF262	0xF262	BSPEED	Implement the SPEED= statement; modified
	0xF26D	0xF26D	BTRACE	Implement the TRACE statement
	0xF26F	0xF26F	BNOTRACE	Implement the NOTRACE statement
	0xF273	0xF273	BNORMAL	Implement the NORMAL statement
	0xF277	0xF277	BINVERSE	Implement the INVERSE statement

	0xF280	0xF280	BFLASH	Implement the FLASH statement
	0xF286	0xF286	BHIMEM	Implement the HIMEM: statement
	0xF2A6	0xF2A6	BLOMEM	Implement the LOMEM: statement
	0xF2CB	0xF2CB	BONERR	Implement the ONERR statement
	0xF2E9	0xF2E9	HANDLERR	Handles an active ONERR GOTO for errors
	0xF318	0xF318	BRESUME	Implement the RESUME statement
	0xF331	0xF331	BDEL	Implement the DEL statement
	0xF390	0xF390	BGR	Implement the GR statement; accelerated
	0xF399	0xF399	BTEXT	Implement the TEXT statement; accelerated
✓	0xF39F	-	STORE	Removed this statement
✓	-	0xF39C	CXREAD	Originally at 0xC5D1 to read audio waveforms
✓	0xF3BC	-	RECALL	Removed this statement
	0xF3D8	0xF3D8	BHGR2	Implement the HGR2 statement; modified
	0xF3E2	0xF3E2	BHGR	Implement the HGR statement; modified
✓	0xF3F2	0xF3EC	CLRHIRE	Clear and set selected HIRES screen; mod
✓	-	0xF3EE	SETHIRE	Set selected HIRES screen to a value; mod
	0xF411	0xF411	HPOSN	Set the HIRES cursor position
	0xF457	0xF457	HRPLOT	Plot a HIRES dot on the screen
	0xF465	0xF465	HRMOVL	Move HIRES cursor left
	0xF47E	0xF47E	COLSHIFT	If odd screen byte, inverse COLBITS
	0xF48A	0xF48A	HRMOVRT	Move HIRES cursor right
✓	-	0xF49C	DRAWHDR	Separates an XDRAW or a DRAW operation; new
✓	0xF457	0xF4A6	XDRAWIT	XDRAW one pixel
✓	0xF4B3	0xF4B8	DRAWIT	DRAW one pixel
✓	0xF4D5	0xF4D1	HRMOVUP	Move HIRES cursor up
✓	0xF505	0xF501	HRMOVDN	Move HIRES cursor down
✓	0xF530	-	HLINRL	Removed this routine; routine never called
✓	0xF5B8	0xF430	BITABLE	Used to initialize the COLOR variable
	0xF53A	0xF53A	HLIN	Draw line from last plotted point; mod
✓	0xF5BA	0xF5B3	ROTATBL	Rotational cosine table every 5.625°; mod
✓	0xF72D	0xF5C7	DRAWCMD	The DRAWCMD routine (DRWPNT) for DRAWSHP
✓	0xF5CB	-	HFIN	Removed this routine; routine never called
✓	0xF601	-	DRAW0	Removed this routine; routine never called
✓	0xF605	0xF600	-	The DRAWSHP routine for DRAW, now DRAWCMD
✓	0xF661	0xF600	-	The DRAWSHP routine for XDRAW, now DRAWCMD
✓	0xF65D	-	XDRAW0	Removed this routine; routine never called
✓	-	0xF666	SQR2	Continuation of FSQR
✓	-	0xF58E	COPYA2F3	Continuation of COPYA2F
✓	-	0xF693	COPYF2A2	Continuation of COPYF2A
✓	-	0xF6A8	COPYT32A	This is the COPYT32A routine; uses LOADARG
	0xF6B9	0xF6B9	GETFNS	Get HIRES plotting coordinates (X/Y) and (A)
	0xF6E9	0xF6E9	BHCOLOR	Implement the HCOLOR= statement
	0xF6F6	0xF6F6	HRCOLTBL	Used to initialize the HRCOLOR variable
	0xF6FE	0xF6FE	BHPLT	Implement the HPLT statement
	0xF721	0xF721	BROT	Implement the ROT= statement
	0xF727	0xF727	BSCALE	Implement the SCALE= statement
✓	-	0xF72D	RND2	Continuation of the FRND routine

	0xF769	0xF769	BDRAW	Implement the DRAW statement
	0xF76F	0xF76F	BXDRAW	Implement the XDRAW statement
✓	-	0xF775	RND3	Continuation of the RND2 routine
✓	0xF775	-	SHLOAD	Removed this statement
✓	-	0xF791	TITLE	“Apple //e+” in upper ASCII
✓	-	0xF79B	PARSIEX	Modifications for 80 columns and lower case
✓	0xF7BC	-	TAPEPNT	Removed this routine
✓	-	0xF7BE	LISTEX	Modifications for LIST in 80 columns
✓	-	0xF7C6	PRTCRES	Modifications for PRINT in 80 columns
✓	0xF7D9	-	GETARYPT	Removed this routine
	0xF7E7	0xF7E7	BHTAB	Implement the HTAB statement; 40/80 columns

Table C.1. Applesoft and Modified Applesoft Entry Points

Appendix D

A great deal of time and effort was put into an on-going project in order to modify DOS 3.3 so that a Binary file could be loaded directly into memory when I first began working at Sierra On-Line in late 1983. This effort ultimately produced a modified DOS BLOAD command that utilized additional keywords that would provide the necessary parameters in order to achieve its accelerated processing rate. That accelerated processing rate could be achieved by reading the data in each file sector and saving that sector data directly to memory one page at a time. Unfortunately, I have no further information on the list of additional keywords that were utilized and the extent of the modifications that went into DOS 3.3, the DOS BLOAD command, and the changes that were made to the Valid Keyword table. Binary files could be loaded into memory in a surprisingly accelerated rate by this uniquely modified DOS 3.3. After I redesigned the DOS HELP command for DOS 4.5.06H as I thoroughly explain in *DOS 4.5 Volume and File Disk Management System Second Edition*, I was able to include a number of additional features into that DOS. The DOS SLOAD and SSAVE commands were two DOS commands that I had available space to include. The DOS SLOAD command is very competitive to that modified Sierra On-Line BLOAD command and it is able to read into memory a *Special* Binary file in a surprisingly accelerated rate. The *Special* Binary file does **not** utilize the first four bytes in its first data sector for its memory load address and for its length in bytes, and those four bytes are simply not included. The memory load address and the length in bytes for a *Special* Binary file must already be known in order to write this file onto a disk volume or to read this file from a disk volume. Since I began developing the SHAPE management software and exploring the Applesoft routines that specifically manage SHAPE Tables, I have reconsidered the usefulness of the DOS SLOAD and SSAVE commands. Furthermore, removing all of the routines that depend on the cassette input and output data ports except for the LOAD and the READ Applesoft statements leaves the demand for developing a DOS SHLOAD command in order to replace the excised Applesoft SHLOAD statement as well as developing a companion DOS SHSAVE command. In view of these changes to the DOS 4.5.06H command repertoire and in the creation of the new DOS 4.5.08H repertoire that replace the DOS SLOAD and SSAVE commands, the following describes the new SHLOAD and SHSAVE Binary file commands that fully support the management of Applesoft SHAPE Tables.

Command	Command Syntax
BLOAD	f [,Ss][,Dd][,Vv][,Aa][,R]
BRUN	f [,Ss][,Dd][,Vv][,Aa]
BSAVE	f [,Ss][,Dd][,Vv][,Aa][,B][,L1][,R[1]]
LLOAD	f [,Ss][,Dd][,Vv][,Aa][,R]
LSAVE	f [,Ss][,Dd][,Vv][,Aa][,B][,L1][,R[1]]
SLOAD	f [,Ss][,Dd][,Vv][,Aa][,B][,R]
SSAVE	f [,Ss][,Dd][,Vv][,Aa][,B][,L1][,R[1]]

Table D.1. Binary File Commands in DOS 4.5.08H

The Binary File commands in the DOS 4.5.08H command repertoire consist of those commands that manage the general operation of Binary or assembly language files. The DOS BLOAD command loads a

Binary file into memory from a disk volume. The DOS BRUN command loads a Binary file into memory from a disk volume before it begins processing the instructions that now reside in memory. The DOS BSAVE command saves the Binary program that currently resides in memory into a file in a disk volume. The DOS LLOAD command loads a *Lisa* Binary file into memory from a disk volume. The DOS LSAVE command saves the *Lisa* Binary program that currently resides in memory into a file in a disk volume. The DOS SHLOAD command loads a SHAPE Table Binary file into memory from a disk volume. The DOS SHSAVE command saves a SHAPE Table Binary structure that currently resides in memory into a file in a disk volume.

The syntax of the Binary File commands for DOS 4.5.08H is shown in Table D.1. All of the Binary File commands are permitted to be used from within an Applesoft program or an assembly language routine as well as on the Apple Command Line.

SHLOAD Command

SHLOAD f [,Ss][,Dd][,Vv][,Aa][,B][,R]

Example: SHLOAD DRAW SHAPE.S,A\$B000
 SHLOAD DRAW SHAPE.S,A\$B000,B
 SHLOAD DRAW SHAPE.S,R

This command is not available in DOS 3.3 for Binary File commands and this command was initially developed for DOS 4.5.08H. The DOS SHLOAD command reads into memory the SHAPE Table Binary file *f* in the specified volume at memory address *a* if the *A* keyword is included. If the *A* keyword is not included with the DOS SHLOAD command, the SHAPE Table Binary file *f* is read into memory at the address the file was originally saved or last saved. SHAPE Table Binary files are *Special* Binary file Type 0x08.

The DOS SHLOAD command copies the 16-bit memory load address that resides in ADRVAL into the page-zero variable HRSHTBL at 0xE8:E9 in the same way that it is copied by the Applesoft handler for the Applesoft SHLOAD statement. The Applesoft handler for the Applesoft SHLOAD statement also copies the memory load address to the page-zero variables FRETOP at 0x6F:70 and to HIMEM at 0x73:74. Only if the *B* keyword is included with the DOS SHLOAD command will DOS copy the memory load address that is in ADRVAL to FRETOP and to HIMEM.

If the *R* keyword is included with the DOS SHLOAD command, the memory load address and the number of bytes that are read into memory are displayed. A SHAPE Table Binary file utilizes the first four bytes in its first data sector for its memory load address and for its length in bytes where both pair of bytes are in Lo/Hi byte order. Therefore, when the *A* keyword is not included with the DOS SHLOAD command, the memory load address information is obtained from the first pair of bytes in its first data sector. The DOS SHLOAD handler always obtains the number of bytes to read into memory from the second pair of bytes in its first data sector. The Applesoft SHLOAD statement is removed from the modified Applesoft.

SHSAVE Command

SHSAVE f [,Ss][,Dd][,Vv][,Aa][,B][,Ll][,R[1]]

Example: SHSAVE DRAW SHAPE.S
 SHSAVE DRAW SHAPE.S,B
 SHSAVE DRAW SHAPE.S,R
 SHSAVE DRAW SHAPE.S,AB000,L\$34,R1

This command is not available in DOS 3.3 for Binary File commands and this command was initially developed for DOS 4.5.08H. The DOS SHSAVE command saves the SHAPE Table Binary structure to file *f* on the specified volume using the memory address *a* and the length *l* in bytes if the A and the L keywords are included, respectively. These two keywords are **optional** in DOS 4.5, but if they are included, **both** keyword values are required. If the A and the L keywords are not included, the address *a* and the length *l* values of the previous SHLOAD or SHSAVE command are utilized. SHAPE Table Binary files are *Special* Binary file Type 0x08.

The B keyword can be used with the DOS SHSAVE command in order to implement the *File Delete/File Save* strategy. That is, the SHAPE Table Binary file *f* is deleted from the volume and then saved to the same volume in order to ensure that the TSL sector(s) of file *f* contain only those Track/Sector entry pairs that are required and utilized by the file.

If the R keyword is included with the DOS SHSAVE command, the memory save address and the number of bytes that are written to the specified volume are displayed. If a non-zero R keyword is included with the DOS SHSAVE command, the number of verified sectors is also displayed with the memory address and the file size information. If CONFIG Bit 1 is **set**, the SHAPE Table Binary file *f* is **not** verified after it is saved to the specified volume. The VALSCNFG variable can be cleared by using the R keyword with the DOS CONFIG command followed by a comma. The features of the newly designed DOS SHSAVE command were never included in the original Applesoft.

