

SHAPE Table Management in an Apple][Computer

Historical Introduction to Applesoft Basic

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

The Microsoft 6502 BASIC interpreter that Apple licensed was Version 1.1. When Apple received the interpreter, it required the work of several talented individuals in order to correct errors in the original source code and to incorporate unique LORES and HIRES graphic commands. Randy Wigginton and Cliff Huston were both instrumental in this effort. The first manual for Applesoft I was published in November, 1977, and the second manual for Applesoft II, the final version, was published nearly a year later in August, 1978. Applesoft II contained further code changes that had already been incorporated into Version 2 of the Microsoft 6502 BASIC interpreter. This final version of the Applesoft interpreter has been virtually unchanged in the Apple][computer even as this computer evolved from the Apple][Plus, the Apple //e, and the Apple //c. Beginning with the Apple //c, however, slight modifications were made to the interpreter in order for it to accept the input of statements in lowercase. Some of these modifications also included the addition of 80-column support and all of these modifications were folded into the Applesoft interpreter that was included in the Enhanced Apple //e Applesoft ROM. The eight-year license that Apple purchased for the Microsoft 6502 BASIC interpreter expired in 1985. In order to obtain a second eight-year license for this interpreter, Apple simply gave their MacBASIC software code to Microsoft in exchanged. This second and final license for the Microsoft 6502 BASIC interpreter expired in 1993.

Introduction to Applesoft Basic Source Code

Apple Computer has yet to publish the source code for any version of Applesoft, either Applesoft I or Applesoft II. 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 in order to provide, perhaps, the most complete source code and documentation of 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 Applesoft interpreter in my Enhanced Apple //e ROM 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 the source code comments by Mr. Sander-Cederlof in his documentation under advisement and I have modified my version of the Applesoft interpreter source code for the Enhanced Apple //e accordingly. Of course, the source code modifications that were suggested by Mr. Sander-Cederlof were only a few among the total number of improvements that I have incorporated into my personal version of the Applesoft interpreter. Some of my Applesoft interpreter source code modifications that pertain to floating-point and complex numbers are thoroughly documented in the paper I published entitled *Using Complex Numbers in an Apple][Computer*.

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	Garbage; Sweet 16	Several garbage collection routines and Sweet 16 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, LOG, LN, MULT, DIV, SGN routines
0xEC-0xEF	Routines	ABS, INT, FPOUT, SQR, POWER, EXP, 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, HPLLOT 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 1. General Layout of Applesoft ROM Routines

The general layout of the Applesoft interpreter is shown in Table 1. The COLDSTRT routine shown in boldface in Table 1 appears to complete Version 1.1 that Apple purchased from Microsoft as Applesoft I, the 6502 BASIC interpreter. The Applesoft statements that follow the COLDSTRT routine begin in the 0xF2 page of the ROM 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 page 0xF7 of the ROM for patches that support 40 and 80-column displays and to

provide a correctly functioning Applesoft HTAB statement. I have not changed the entry locations for any Applesoft statement. What I have changed are the routines that are used by their Applesoft handlers.

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 thirty-two-bit significand is incremented. When addition, subtraction, or multiplication is initiated, only one operand uses a full forty-bit significand and the other operand uses a thirty-two-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 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 zero 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.

Applesoft Random Number Generator

The random number generator that is utilized in the Applesoft ROM 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 which is linked to the Apple USR function. 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 are utilized by the Applesoft RND routine are located at 0xEFA6 and 0xEFAA, and yet they are used as floating-point variables. In the Applesoft interpreter, floating-point variables must be five bytes in size, 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 2^{32} 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 2^{32} 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 0x12B960A5 and C should equal 0x361962EA. These two values are quite different from the values that are found in the original Applesoft RND routine. 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 routines. Applesoft multiplies the seed at 0xC9 with its version of A and adds to that product its version of C. Applesoft then implements a modulo 2^{32} by changing the resulting exponent to 0x80 before it normalizes the final result. Simply stated, floating-point numerical routines are designed to **preserve** the most significant bits and **discard** the least significant bits during the implementation of those numerical

routines. 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 it uses a minimum of variable and program space. I do agree that the multiply routine that he utilizes 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. Figure 1 shows the Peasant integer multiply routine that I utilize in my Applesoft RND routine.

:	:	:
F740 A0 20	850	ldy #32
F742	851 ;	
F742 46 62	852 ^8	lsr MULMANT
F744 66 63	853	ror MULMANT+1
F746 66 64	854	ror MULMANT+2
F748 66 65	855	ror MULMANT+3
F74A	856 ;	
F74A 90 0F	857	bcc >1
F74C	858 ;	
F74C 18	859	clc
F74D	860 ;	
F74D A2 03	861	ldx #3
F74F	862 ;	
F74F B5 9E	863 ^9	lda FACMANT,X
F751 75 A6	864	adc ARGMANT,X
F753 95 9E	865	sta FACMANT,X
F755 95 C9	866	sta IRAND,X
F757	867 ;	
F757 CA	868	dex
F758 10 F5	869	bpl <9
F75A	870 ;	
F75A 06 A9	871 ^1	asl ARGMANT+3
F7FC 26 A8	872	rol ARGMANT+2
F75E 26 A7	873	rol ARGMANT+1
F760 26 A6	874	rol ARGMANT
F762	875 ;	
F762 88	876	dex
F763 D0 DD	877	bne <8
F765	878 ;	
F765 84 A2	879	sty FACSIGN
F767 84 AC	880	sty FACGUARD
:	:	:

Figure 1. Peasant Integer Multiply Routine

Every culture throughout history teaches their children the method or the algorithm that that culture uses in order to multiply two large 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 multiply method. 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, 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 works that is shown in Figure 1. The multiplier resides in the MULMANT register at 0x62 and it contains the four-byte variable A at 0xEFA4 (I rewrote the previous routine POLYNOM making it faster and two bytes smaller). The multiplicand resides in the ARGMENT register at 0xA6 and it contains the four-byte seed at 0xC9. The four-byte variable C at 0xEFA8 is copied into the FACMANT register at 0x9E which serves as the product register. After the multiplier in the MULMANT register is shifted right and if the C-flag is set noting an odd number, the multiplicand in the ARGMENT register is added to the product in the FACMANT register. Whether an addition occurs or not, the multiplicand is shifted left thus doubling its value. Any bit that is shifted into the C-flag by 0xA6 is discarded. Using four bytes in each of these registers ensure that modulo 2^{32} remains in force throughout the required thirty-two iterations of this algorithm.

My Applesoft 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 my Applesoft RND routine as in RND(-1234), for example, my RND routine saves that value to IRAND at 0xC9 as a positive 32-bit integer which will be the seed for the next random number iteration. If a zero argument is provided to my Applesoft RND routine as in RND(0), my RND routine 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 one is provided to my Applesoft RND routine as in RND(1), my RND routine returns a fractional value that has a range from zero to less than one 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 one is provided to my Applesoft RND routine as in RND(192) or RND(280), for example, my RND routine 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 the RND routine, and if that value is greater than zero, that value is saved to TEMP1 at 0x93 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 at 0x9D and that 32-bit product integer is normalized as a floating-point number using the NORMFAC1 routine at 0xE82E. If a Range of one is supplied to my RND routine, 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 TEMP1 using FMULT at 0xE97F, its product is converted to an integer value by FINT at 0xEC23, and the result is returned to the user as an integer value.

The goal in testing a random number generator is to ensure that the random number generator produces a random stream of data values. Testing a random number generator can include chi-square tests, Kilmogorov-Smirnov tests, serial-correlation tests, two-level tests, k-dimensional uniformity or k-distributivity tests, serial tests, spectral tests, or any combination of these tests. The chi-square test is by far the most commonly used test because it can be used for any distribution of data, a histogram can easily be prepared from the observed values, and the observed frequencies can easily be compared with known theoretical frequencies. Simply stated, a chi-square test shows the relationship between two entities and whether or not the observed patterns are likely to be purely random. The Kilmogorov-Smirnov test was designed for continuous distributions of data values. A serial-correlation test can show the degree of nonzero covariance and its measure of dependence on the random number generator. A two-level test first uses the chi-square test on a known number of samples that have a known given size, and then it performs a chi-square test on the same number of chi-square statistics that are obtained. Chi-square is known for its 1-distributivity property, so k-dimensional uniformity for 2-distributivity would generalize on this property of a random number generator in two or more dimensions. A serial test would build on the chi-square test in order to isolate the deviation of counts to expected counts, the degrees of freedom, and the use of k-tuples in order to extract non-overlapping values or values that are not independent of the chi-square test being

used. Finally, a spectral test can determine how densely the k-tuples can fill a k-dimensional hyperspace where the k-tuples can fall on a finite number of parallel hyper-planes.

Visual tests to display the randomness of a random number generator can easily process a massive bitmap image in order to visualize repetitions and patterns that are produced by a random number generator as side-effects. The US National Institute of Standards and Technology utilizes fifteen tests alone that include frequency tests, discrete Fourier transform tests, aperiodic tests, and linear complexity tests. Certainly, there does exist a huge array of tests and procedures that can be used to quantify the effectiveness of a random number generator. In the Apple computer, the length of the period before the random number generator begins to repeat is its most important feature. And, indeed, that period can determine if the random number generator is **sufficient**.

The random number generator in the unmodified Applesoft ROM can easily be tested using Applesoft Program 1 that is shown in Figure 2. As an aside, any Applesoft program can be processed and its instructions can be displayed as in Figure 2 by an assembly language program that I created and called *Applesoft List*. The Applesoft program RND Test 1 enables HIRES using the color WHITE, or 0x7F, it selects full screen mode, and it uses the Applesoft RND statement in order to set the X and the Y coordinates to turn a pixel ON using the Applesoft HPLLOT statement. As long as a key is not pressed, the program will continue to select X and Y coordinates in order to turn another pixel ON. If the RND routine can transform the entire screen area from black to white, the RND function has a period of substantial size such that the RND routine is **sufficient** for use on the Apple computer. Figure 3 shows another Applesoft program called RND Test 2 that is nearly identical to RND Test 1. RND Test 2 utilizes the Range feature in my Applesoft RND routine which I installed in ROM in place of the original RND routine. The Range feature simply performs the coordinate multiplication which was explicitly done in RND Test 1. Figure 4 shows the results when RND Test 1 executes on a Virtual][simulated Apple //e computer that contains an unmodified ROM and the computer is set to maximum speed. No new pixels are created after 4.5 seconds of processing and the HIRES screen is left only partially converted from black to white. When RND Test 2 executes on another Virtual][simulated Apple //e computer that contains a modified ROM with my RND routine installed and the computer is set to the same processing speed, the HIRES screen is *fully* converted from black to white in 1:47.5 minutes. Most of the HIRES screen is converted in the first twenty seconds. It simply requires a bit more time in order to reach **ALL** of the necessary random number values that will eventually convert the entire HIRES screen as shown in Figure 5. Figure 5 demonstrates visually that my Applesoft RND routine is **sufficient** and, therefore, far more useful in the Apple][computer.

```
10 HGR
   HCOLOR = 3
   POKE 49234, 0
20 X = RND( 1 ) * 280
   Y = RND( 1 ) * 192
   HPLLOT X, Y
   IF PEEK( 49152 ) < 128 THEN GOTO 20
30 POKE 49168, 0
   TEXT
   END
```

Figure 2. RND Test 1

```
10 HGR
   HCOLOR = 3
   POKE 49234, 0
20 X = RND( 280 )
   Y = RND( 192 )
   HPLLOT X, Y
   IF PEEK( 49152 ) < 128 THEN GOTO 20
30 POKE 49168, 0
   TEXT
   END
```

Figure 3. RND Test 2

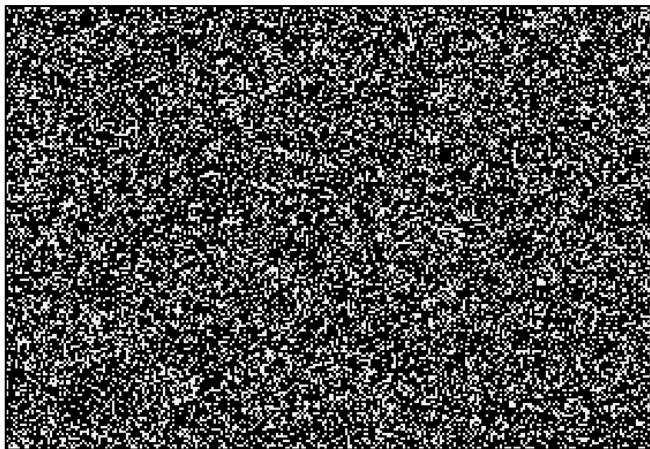


Figure 4. RND Test 1 Display



Figure 5. RND Test 2 Display

Memory	Name	Description
0xDE23	FRMSTAK3	Call RNDUP to roundup FAC, push FAC onto the stack, and jump to (INDEX)
0xDE4A	NOTMATH	Pull ARGEXP first from stack, set ARGSIGN and XORSIGN, and load FACEXP into A-reg
0xE3AF	FUNCDATA	Pulls five data bytes from stack into (FUNCNAM) indexed by Y-reg from 0-4
0xEAE6	COPYM2F	Copy four bytes of MULMANT into FACMANT
0xEAF9	LOADFAC	Copy five bytes from (INDEX) into FAC setting FACSIGN; set FACGUARD to zero
0xEB1E	COPYF2T1	Copy address of TEMP1 into INDEX; copy FAC into (INDEX); set FACGUARD to zero
0xEB21	COPYF2T2	Copy address of TEMP2 into INDEX; copy FAC into (INDEX); set FACGUARD to zero
0xEB27	COPYF2FR	Copy address of FORPNT into INDEX; copy FAC into (INDEX); set FACGUARD to zero
0xEB2B	COPYFAC	Copy FAC into (INDEX) indexed by Y-reg 0-4; set FACGUARD to zero
0xEB53	COPYA2F	Copy ARGSIGN to FACSIGN; copy ARG to FAC (unwound); set FACGUARD to zero
0xEB63	COPYF2A	Call RNDUP to roundup FAC; copy FACSIGN to ARGSIGN; copy FAC to ARG (unwound); set FACGUARD to zero

Table 2. Routines That Copy Floating-Point Registers

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 floating-point register in Applesoft is FAC at 0x9D and the secondary floating-point register is ARG at 0xA5. The multiply function also utilizes MULMANT at 0x62, a four-byte register that holds only the floating-point mantissa. Polynomial processing utilizes TEMP1 and TEMP2 at 0x93 and 0x98, respectively, which are full, five-byte floating-point registers. TAN processing utilizes TEMP1, TEMP2, and TEMP3, for example, and the TEMP3 five-byte register resides at 0x8A. EXP also uses TEMP3 and SQR uses TEMP1. There are eight routines that copy these registers from one area of memory to another area of memory or from one register to another register. Most of these routines copy the data to or from a register byte by byte in order to affect the fastest transfer speed of data at the expense of code space. There are two routines, however, that favor code space at the expense of data transfer speed. I have unwound both of these indexed register loops ensuring that the processor register that is used to index the loop is set to its terminating value. Unfortunately, there is a

location at 0xDE23 in Applesoft where the entire content of the FAC register is pushed onto the stack after RNDUP is called. Its complement routine at 0xDE4A pulls from the stack the entire content of the ARG register. FUNCDATA at 0xE3AF is a routine that pulls from the stack the entire content of a floating-point number in order to copy it to the address that resides in FUNCNAM, or 0x8A:8B. Table 2 presents all of these floating-point copy routines, their location in ROM, their names, and a description of their function.

A Better Square Root Routine for Applesoft

The power operator ^ or exponentiation routine POWER in the Applesoft interpreter requires a substantial amount of processing. Given the general format equation, $Z = X^Y$, the Applesoft interpreter copies the argument that resides in X into the ARG register and copies the argument that resides in Y into the FAC register. Once the processing is complete, the Applesoft interpreter copies the result that is in the FAC register to the Z variable. The Applesoft processing for POWER is:

$$FAC = EXP[LN(ARG) * FAC]$$

Processing the ARG register, that is, the value of X, for the value of its natural logarithm adds the square root of 0.5, divides by the square root of 2.0, subtracts 1.0, executes a polynomial expansion using four floating-point variables, adds -0.5, and finally multiplies the ARG register by the natural logarithm of 2.0. The ARG register is then multiplied by the FAC register which contains the argument that resides in Y. That product is used to raise e to that power by the EXP routine. The EXP routine multiplies the FAC register by the inverse of the natural logarithm of 2.0 or $1/\ln(2)$, manipulates the exponent of FAC, swaps the FAC and the ARG registers, subtracts those registers, executes a polynomial expansion using eight floating-point variables, and finally manipulates the exponent of FAC one last time. This is by far a substantial amount of processing simply to raise a number to the power of 0.5. That is, to calculate the square root of a number. Of course, I agree that the necessary routines are already available and in service for their own Applesoft statement processing: POWER for power and EXP for exponentiation. Is using these available routines necessarily the better choice rather than specifically using the Newton-Raphson iteration method which is far more accurate and its processing is faster? The equation for the Newton-Raphson iteration method is:

$$R = [(N / X) + X] / 2 \text{ until } R \approx X \text{ else } X = R$$

Selecting the most 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 its exponent bias is removed. For example, Table 3 lists a few floating-point numbers, their value in Applesoft floating-point numerical representation, and their square root value, also in Applesoft floating-point numerical representation. It is rather easy to understand that the exponent of the square root value is around half of its given value once the exponent bias of 0x80 is removed. And, that is precisely why the Microsoft engineers utilized the natural logarithm properties and the Applesoft exponentiation routine. If the natural logarithm of a number is multiplied by 2.0, for example, when e is raised to that power, the original number is squared. However, the Applesoft exponentiation routine requires a substantial amount of processing in order to produce a correct value having up to ten numerical places.

Number	Applesoft Floating-Point		Square Root Value	
	With Bias	Without Bias	With Bias	Without Bias
0.1234	0x7D7CB923A3	0x037CB923A3	0x7F33DB69B1	0x0133DB69B1
1.234	0x811DF3B646	0x011DF3B646	0x810E308398	0x010E308398
123.4	0x8776CCCCCD	0x0776CCCCCD	0x8431BCA47E	0x0431BCA47E
1,234,000.0	0x9516A28000	0x1516A28000	0x8B0ADB6082	0x0B0ADB6082

Table 3. Floating-Point Numbers and Their Square Roots

:	:	:
EE81 20 1E EB	614 FSQR	jsr COPYF2T1
EE84 F0 51	615	beq RTN.EE.D
EE86	616 ;	
EE86 84 47	617	sty YREG
EE88	618 ;	
EE88 17	619	clc
EE89	620 ;	
EE89 49 80	621	eor #EXPBIAS
EE8B 10 01	622	bpl >1
EE8D	623 ;	
EE8D 38	624	sec
EE8E	625 ;	
EE8E 6A	626 ^1	ror
EE8F	627 ;	
EE8F 18	628	clc
EE90	629 ;	
EE90 69 80	630	adc #EXPBIAS
EE92 85 9D	631	sta FACEXP
EE94	632 ;	
EE94 4C 64 F6	633	jmp FSQR2
:	:	:
F66B 20 21 EB	661 FSQR2	jsr COPYF2T2
F66E	662 ;	
F66E A9 93	663	lda #TEMP1
F670 A0 00	664	ldy /TEMP1
F672 20 66 EA	665	jsr FDIV
F675	666 ;	
F675 A9 98	667	lda #TEMP2
F677 A0 00	668	ldy /TEMP2
F679 20 BE E7	669	jsr FADD
F67C	670 ;	
F67C C6 9D	671	dec FACEXP
F67E E6 47	672	inc YREG ; iteration counter
F680	673 ;	
F680 A9 98	674	lda #TEMP2
F682 A0 00	675	ldy /TEMP2
F684	676 ;	
F684 20 B2 EB	677	jsr FPCOMP
F687 D0 E2	678	bne FSQR2
F689	679 ;	
F689 60	680	rts
:	:	:

Figure 6. The Applesoft FSQR Handler Routine

The supplied floating-point argument is saved to the TEMP1 register at 0x93 in line #614 in Figure 6. Using floating-point example numbers like those shown in Table 3 help to ascertain those few instructions that are shown from lines #619 to #631 in order to quickly and easily create an appropriate initial value for the routine. The routine continues in the 0xF0 ROM at 0xF66B. The current approximation of the square root is saved to the TEMP2 register at 0x98. The supplied argument that resides in the TEMP1 register is divided by the value that resides in the FAC register, and that quotient is added to the current approximation that resides in the TEMP2 register. In order to divide the value that now resides in the FAC register by two, the FAC exponent is simply decremented. The FAC register now contains a new approximation of the square root. That new square root approximation is compared to the previous approximation that still resides in the TEMP2 register. If the values compare to ten numerical places, the routine returns with the value of the square root in the FAC register, or it returns to line #661. Usually, only about four iterations are required.

Management of Coordinate Displacements in HLIN

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 by the Applesoft HPLLOT statement. The HPLLOT handler utilizes the Applesoft HLIN routine in order to draw a line and this HLIN routine persists unchanged even in the Applesoft of an Enhanced Apple IIe, which is shameful in my opinion. I have analyzed the HLIN routine and found that the routine does not correctly calculate the delta difference of the horizontal and the vertical coordinate displacements before drawing the requested line. There are two memory locations that simply require a small code adjustment. The first code adjustment is made at 0xF57A and the second code adjustment is made at 0xF5A5. It is simply amazing how *lovely* and *symmetrical* diagonal lines can be drawn either from left to right, from right to left, from top to bottom, or from bottom to top. How did the original Applesoft code pass any sort of testing and/or code review is beyond my comprehension. The issue primarily rests on how one interprets a delta displacement. The Apple engineers Randy Wigginton and Cliff Huston, based on my assessment of their HLIN routine, believe that data displacement equals the difference between two pixel locations both in the horizontal and in the vertical direction. I believe that data displacement is equal to the difference of two pixel locations minus one in both the horizontal and in the vertical direction. For example, given two pixel points where A is at (14, 9) and B is at (23, 5), the horizontal displacement *delta* would be equal to $|14 - 23| - 1 = 8$ and the vertical displacement *delta* would be equal to $|9 - 5| - 1 = 3$. The unmodified HLIN routine calculates the displacement *difference* for these two pixel points, A and B, as 9 for the horizontal displacement *difference* and 4 for the vertical displacement *difference*. Figure 7 shows pixel points A and B drawn on graph paper. Only when displacement deltas are used will the modified HLIN routine *elegantly* draw a diagonal line from one pixel to the other pixel.

The Applesoft HPLLOT handler along with the HLIN line drawing routine turns pixels ON or OFF only at discrete pixel locations that are within the HIRES drawing area. One simply needs to count how many pixels, either in the horizontal direction or in the vertical direction, that exist between points A and B in order to understand how many pixels this handler and this line drawing routine are required to manipulate. There are only 8 pixels horizontally between A and B and there are only 3 pixels vertically between A and B that can be manipulated. Aside from pixels A and B themselves, only those pixels between A and B need to be manipulated and either turned ON or turned OFF in order to create the most visually pleasing diagonal line between pixel points A and B. The displacement *difference* between A and B is meaningless within the context of the HLIN line drawing routine. Using a displacement *difference* to calculate which pixels to turn ON or which pixels to turn OFF will create a visually irregular and unnatural looking diagonal line each and every time. When calculating which pixels to manipulate, using a displacement *delta* to manipulate the correct line pixels will create the most visually pleasing diagonal line between two HIRES pixels.

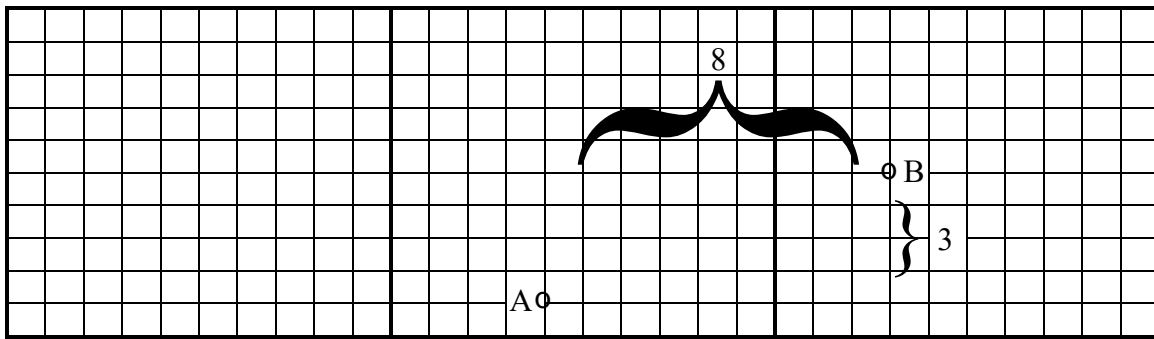


Figure 7. HLIN Delta Displacement Calculation

Applesoft Commands Used with SHAPE Table Drawing Routines

The COLDSTRT routine at 0xF128 and the Applesoft statements CALL at 0xF1D5, IN at 0xF1DE, and PR at 0xF1E5 appear to complete Version 1.1 of the Microsoft 6502 BASIC interpreter that Apple Computer originally leased according to the general layout of Applesoft ROM routines that is shown in Table 1 above. The remaining ROM code space beginning at 0xF1EC is entirely devoted to those Applesoft statements that Randy Wigginton and Cliff Huston added to the Applesoft source code in order to support the LORES and the HIRES screens using specific Applesoft graphic routines. The LORES and the HIRES screens and graphic routines set the Apple][apart from all previous home computers. These are also the Applesoft graphic routines that sold the Apple][to the world of amazed engineers, universities, schools, and programmers, from young adolescents to the retired, having little or vast amounts of programming experience. Never before was there such a magnificent machine that could easily create visual magic using only Applesoft.

The Applesoft DRAW, XDRAW, ROT, SCALE, and SHLOAD statements, and a number of other Applesoft routines that support the HIRES drawing handlers of the first four of these five statements, reside between 0xF1EC and 0xF7FF. Of course, the ROM Monitor is sacrosanct and 0xF800 through 0xFFFF is not utilized in graphic routines. Upon casual inspection, there are a few routines in the lower 0xF0 ROM that are never utilized by Applesoft. I believe that the cassette tape recorder data input and output ports are useless except to support Insta-Disk and c2t processing, routines that were both developed by Egan Ford. Thus, the LOAD and the READ Applesoft statements are required for this special processing and they are retained in my modified ROM. All other Applesoft statements that depend on the cassette data ports are unnecessary and these statements include RECALL, SAVE, STORE, and SHLOAD. It so happens that DOS 4.5.08H conveniently includes a versatile SHLOAD command as well as a companion SHSAVE command that, together, provide far more functionality and usefulness than the Applesoft RECALL, SAVE, STORE, and SHLOAD statements combined. All decisions that concern Applesoft memory utilization require sufficient justifications whether that memory is better suited for one function rather than for another function. It is my attempt to provide those justifications in order to design a far more intelligent set of HIRES drawing routines for Applesoft. It is also mandatory that all Applesoft statements that reside in this lower area of the 0xF0 ROM begin at their traditional entry points. Any non-critical subroutine or function that is moved to another memory location will be identified. Typically, programmers only utilize the memory addresses of major statement handlers and they rarely need to utilize the memory addresses of minor subroutines or functions that are within or near their statement handlers. Therefore, I will present the 0xF1EC to 0xF7FF memory space and identify all of the major Applesoft statements or routines, their addresses, and their names as well as all of the minor subroutines and functions, their addresses, and their names. Table 4 lists all of the major and the minor handlers, routines, and functions that reside in the Applesoft 0xF0 ROM between 0xF1EC and 0xF7FF.

Memory	Name	Description
0xF1EC	PLOTFNS	Sets LORES comma separated coordinates for H2 and V2
0xF206	GOIQERR	Jumps to print Illegal Quantity Error
0xF209	ODRCOOR	Gets A, B and C values for HLIN and VLIN
0xF225	PLOT	Processes the Applesoft PLOT statement
0xF232	HLIN	Processes the Applesoft HLIN statement
0xF241	VLIN	Processes the Applesoft VLIN statement
0xF24F	COLOR	Processes the Applesoft COLOR statement
0xF256	VTAB	Processes the Applesoft VTAB statement
0xF262	SPEED	Processes the Applesoft SPEED statement
0xF26D	TRACE	Processes the Applesoft TRACE statement
0xF26F	NOTRACE	Processes the Applesoft NOTRACE statement
0xF273	NORMAL	Processes the Applesoft NORMAL statement
0xF276	INVERSE	Processes the Applesoft INVERSE statement
0xF280	FLASH	Processes the Applesoft FLASH statement
0xF286	HIMEM	Processes the Applesoft HIMEM statement
0xF2A6	LOMEM	Processes the Applesoft LOMEM statement
0xF2CB	ONERR	Processes the Applesoft ONERR statement
0xF2E9	HANDLERR	Handles active ONERR GOTO Applesoft statements
0xF318	RESUME	Processes the Applesoft RESUME statement
0xF331	DEL	Processes the Applesoft DEL statement
0xF390	GR	Processes the Applesoft GR statement
0xF399	TEXT	Processes the Applesoft TEXT statement
0xF39F	STORE	This Applesoft statement is removed
0xF39F	CXREAD	Reads audio waveform for HEADER, SYNC, and binary DATA
0xF3BC	RECALL	This Applesoft statement is removed
0xF3D8	HGR2	Processes the Applesoft HGR2 statement
0xF2E2	HGR	Processes the Applesoft HGR statement
0xF3EE	CLRHIREs	Clears selected HIRES screen; replaces HCLR at 0xF3F2 and BCGND at 0xF3F6
0xF411	HPOSN	Sets the HIRES pixel cursor position; X,Y-reg horizontal 0-279; A-reg vertical 0-191
0xF457	HPL0T	Turn a HIRES pixel ON; X,Y-reg horizontal; A-reg vertical
0xF465	HRMOVLF	Minor routine to move the HIRES pixel cursor left
0xF47E	HRODDBYT	Minor routine to test if COLBITS (0x1C) > 0x1F or COLBITS < 0xE0
0xF48A	HRMOVRT	Minor routine to move the HIRES pixel cursor right
0xF49C	DRAWHDR	Minor routine to test SHPVAL (0xD0); uses OPRND (0x44) to fall into XDRAWIT or DRAWIT
0xF4A6	XDRAWIT	Minor routine to map out a pixel; original routine was flawed
0xF4B6	DRAWIT	Minor routine to map in a pixel
0xF4D3	HRMOVUP	Minor routine to move the HIRES pixel cursor up
0xF505	HRMOVDN	Minor routine to move the HIRES pixel cursor down
0xF532	BITABLE	Minor table of mask values that are used to set COLOR (0x30); original is at 0xF5B2
0xF530	HLINRL	This minor routine is removed (never called by Applesoft)
0xF532	BITBYTS	B1TBYT03, B1TBYT04, and B1TBYT1C as discrete B1T values
0xF53A	HLIN	Processes the Applesoft HLIN statement; A,X-reg horizontal 0-279; Y-reg vertical 0-191
0xF5B2	MSKTBL	Minor table moved to 0xF532 (BITABLE)
0xF5B3	ROTATBL	Rotation table in 360/64 degrees; COS(90 * X/16) * 0x100, X = 0:15; original at 0xF5BA
0xF5BA	COSTABLE	Minor table moved to 0xF5B3 (ROTATBL)
0xF5C4	DRAW	Processes the Applesoft DRAW statement; original is at 0xF769, jump to 0xF605

0xF5C6	XDRAW	Processes the Applesoft XDRAW statement; original is at 0xF76F, jump to 0xF661
0xF5CB	HFIND	This minor routine is removed (never called by Applesoft)
0xF601	DRAW0	This minor routine is removed (never called by Applesoft); initialize SHAPE (0x1A:1B)
0xF605	DRAW1	This minor routine is folded into DRAW at 0xF5C4
0xF65D	XDRAW0	This minor routine is removed (never called by Applesoft); initialize SHAPE (0x1A:1B)
0xF661	XDRAW1	This minor routine is folded into XDRAW at 0xF5C6
0xF66B	FSQR2	Continuation of Newton-Raphson iteration for square root, $R = ((N/X) + X) / 2$
0xF68A	COPYA2F3	Continuation of COPYA2F; copy ARG to FAC; see Table 2
0xF69B	COPYF2A3	Continuation of COPYF2A; copy FAC to ARG; see Table 2
0xF6B9	GETFNS	Get HPOSN coordinates; X/Y-reg for horizontal, A-reg for vertical
0xF6E6	DOIQERR	Jumps to print Illegal Quantity Error
0xF6E9	HCOLOR	Processes the Applesoft HCOLOR statement
0xF6F6	HRCOLTBL	HIRES color table
0xF6FE	HPLOT	Processes the Applesoft HPLOT statement
0xF721	ROT	Processes the Applesoft ROT statement
0xF727	SCALE	Processes the Applesoft SCALE statement
0xF72D	DRWPNT	This routine is removed; called by DRAW -> DRAW1 and XDRAW -> XDRAW1
0xF72D	FRND2	Continuation of Applesoft RND random number handler
0xF775	SHLOAD	This Applesoft statement is removed
0xF78F	TITLE	ROM Monitor cold start initialization writes TITLE to the top of the TEXT screen
0xF799	PARSIEX1	Start of patches that support the 80 column display in the Apple //e
0xF7BC	TAPEPNT	This minor routine is removed (used by STORE and RECALL)
0xF7D9	GETARYPT	This minor routine is removed (used by STORE and RECALL)
0xF7E7	HTAB	Processes the Applesoft HTAB handler correctly

Table 4. Applesoft ROM Routines from 0xF1EC to 0xF7FF

The routines in my modified Applesoft ROM for the Enhanced Apple //e reside at the same address as they do in the unmodified Applesoft ROM from 0xF1EC to 0xF39F as shown in Table 4. At 0xF39F I replaced the Applesoft STORE and the RECALL statements with the CXREAD routine that was originally at 0xC5D1 before I installed the Mini-Assembler into the CXROM space in my modified ROM. Even though it was unnecessary for me to modify the HGR2 and HGR handlers that support these Applesoft statements, I found that I was able to process these two handlers faster while adding in an elegant transition from the TEXT page to the cleared HIRES page. In other words, my routine clears the respective HIRES pages before addressing any soft switches. The viewer is not shown the HIRES pages as the pages are being cleared as they are in the unmodified ROM; rather, the viewer is shown the HIRES pages *after* they have been cleared. Perhaps HLINRL that was at 0xF530 is a residue of an abandoned effort that began with the initialization of the 0xE0:E2 HIRES coordinate registers. This routine certainly does not appear to have any usefulness in view of the other HIRES routines such as HLIN, HPLOT, DRAW, and XDRAW. The rotational table, color table, and the BIT value table are placed primarily at locations in order to maintain the original addresses of the major handlers that support their Applesoft statements. The unused and deleted HFIND routine at 0xF5CB appears to be the converse of the HPOSN routine that is retained at 0xF411 because it is used by HPLOT and DRAWCMD.

I found a pair of routines that consist of nearly identical instructions to my good fortune and certainly to my surprise in the unmodified 0xF0 ROM that resides in my Enhanced Apple //e. The pair of routines is DRAW1 at 0xF605 and XDRAW1 at 0xF661. Both of these routines are exactly 0x58 bytes in size and they each

include the DRAWSHP routine as it is called in the modified 0xF0 ROM. And, the routines are identical except that DRAW1 calls either LRUD1 or LRUD2 and XDRAW1 calls either LRUDX1 or LRUDX2. The only difference between LRUD1 and LRUD2 is that LRUD1 begins with a c1c instruction and it falls directly into the LRUD2 routine. The same holds true for LRUDX1 and LRUDX2: LRUDX1 begins with a c1c instruction and it falls directly into the LRUDX2 routine. Actually, DRAW1 could call the same routine, LRUD for example, simply by moving that c1c instruction into the DRAW1 routine just prior to the first call to LRUD. The same would hold true for XDRAW1. The LRUD1 and LRUD2 set of routines and the LRUDX1 and LRUDX2 set of routines both begin with the same three instructions, so these pair of routines can easily be combined if a single routine can determine if DRAW1 or if XDRAW1 is processing. That is precisely how DRAW and XDRAW begin in the modified 0xF0 ROM and the same software instructions follow their introduction. Instead of calling the DRAWCMD routine, that is DRWPNT at 0xF72D in DRAW1 and in XDRAW1 in the unmodified 0xF0 ROM, the DRAWCMD routine is put in-line following the short introduction of DRAW and XDRAW. Furthermore, the DRAWSHP routine is also put in-line following the DRAWCMD routine. It is the DRAWSHP routine that is the most interesting in terms of flawed logic, illogical decisions, and incorrect data table values. Once these issues are corrected, DRAW and XDRAW become quite amazing tools for the drawing of a shape definition whose data is contained within a SHAPE table. A SHAPE table can be efficiently loaded into memory using the DOS 4.5.08H SHLOAD command. This DOS command initializes the HRSHTBL address at 0xE8:E9 with the address of the SHAPE table that is read into memory at its specified address as well as optionally initializing the FRETOP address at 0x6F:70 and the HIMEM address at 0x73:74 with the HRSHTBL address.

Figure 8 shows how the DRAWHDR routine at 0xF49E in Figure 12 can determine if DRAW or if XDRAW is processing simply by looking at the MSB of the value that resides in OPRND at 0x44. Once the value in OPRND is established, the code for DRAW and for XDRAW is virtually identical. The code for the DRAW and the XDRAW handlers for their Applesoft statements when added together requires 311 bytes for their processing in the unmodified 0xF0 ROM. In the modified 0xF0 ROM, 216 bytes are required to process both the Applesoft DRAW and the Applesoft XDRAW statements. Saving 95 bytes of precious ROM memory and still enjoy the same ROM functionality is very good fortune and certainly a welcomed surprise.

The modified 0xF0 ROM version for DRAW and for XDRAW includes further surprises when the flawed logic, illogical decisions, and incorrect data table values that exist in the unmodified 0xF0 ROM version are addressed. To complete the unification of the unmodified ROM routines LRUD1, LRUD2, LRUDX1, and LRUDX2, Figure 12 shows the DRAWHDR, XDRAWIT, and DRAWIT routines and how they literally fall into the common XDRAW/DRAW routine and the HIRES pixel cursor move routines. Before the data of any SHAPE definition is processed, the variable HRCOLCNT at 0xEA is initialized to zero. That variable is incremented every time DRAW or XDRAW finds unexpected congruent data on the HIRES screen that is not part of the SHAPE definition data. However, Applesoft does not provide any further processing of the HRCOLCNT variable. It is left to the user to utilize this collision counter for any useful SHAPE definition post-processing.

The Applesoft ROT statement sets the SHAPE rotation to a value from zero to sixty-three according to SHAPE table documentation in the *Applesoft][Basic Programming Reference Manual*, publication #030-0013-E. Each quadrant of a circle can support up to sixteen rotational values depending on the Applesoft SCALE value. As shown in Figure 8, the HRROT value at 0xF9 is divided by sixteen and the resulting target quadrant value is saved in ROTQVAL at 0xD1. That target quadrant value is used in Figure 12 along with the direction of movement information that is contained in the current SHPVAL value in 0xD0. Randy Wigginton and Cliff Huston were quite clever in their SHAPE value and direction of movement design when that value is added to the target quadrant value that is shown in lines 208 and 209. The SHPVAL value direction for pixel cursor movement is 0x00 for UP, 0x01 for RIGHT, 0x02 for DOWN, and 0x03 for LEFT. So, it is no surprise that when adding in, for example, the next quadrant value or 0x01, the next direction of pixel cursor movement is automatically selected in a clockwise fashion. If the sum in the A-register sets the C-flag, the

C-flag is re-configured for the direction of pixel cursor movement for lines 626 and 638 in Figure 13, and the HIRES pixel cursor is moved to its next position after the common XDRAW/DRAW processing is complete.

:	:	:
F5C4 18	500	DRAW clc
F5C5	501	;
F5C5 B0 00	502	bcs *+2
F5C7	503	dfs !-1
F5C6	504	;
F5C6 38	505	XDRAW sec
F5C7	506	;
F5C7 66 44	507	ror OPRND
F5C9	508	;
F5C9	509	;
F5C9	510	; This is the DRAWCMD routine.
F5C9	511	;
F5C9 20 F8 E6	512	jsr GETBYT ; get requested SHAPE number
:	:	:
F5FC 20 C0 DE	557	jsr SYNTAXCHK
F5FF 20 B9 F6	558	jsr GETFNS ; get coordinates
F602 20 11 F4	559	jsr HPOSN ; get GBASL/GBASH, E0:E2, E5
F605	560	;
F605	561	;
F605	562	; This is the DRAWSHR routine.
F605	563	;
F605 A5 F9	564	^5 lda HRR0T
F607	565	;
F607 4A	566	lsr
F608 4A	567	lsr
F609 4A	568	lsr
F60A 4A	569	lsr
F60B	570	;
F60B 85 D1	571	sta ROTQVAL
F60D	572	;
F60D A5 F9	573	lda HRR0T
F60F 29 0F	574	and #SROTMASK
F611 AA	575	tax
F612	576	;
F612 BC B3 F5	577	ldy ROTATBL,X
F615 88	578	dey
F616 84 D2	579	sty ROTHVAL
F618	580	;
F618 49 0F	581	eor #SROTMASK
F61A AA	582	tax
F61B	583	;
F61B BC B4 F5	584	ldy ROTATBL+1,X
F61E 84 D3	585	sty ROTVVAL
:	:	:

Figure 8. The DRAW and XDRAW Statement Handlers

The masked value of HRR0T in Figure 8 at line #574 is within a single quadrant and that value is used to select the horizontal and the vertical linear summation values from the ROTATBL table in lines #577 and

#584 and utilized in Figure 13. The ROTATBL table is shown in Table 5 and these entries are based on the linear calculations of $\text{COS}(90 * X/16) * 256$ and of $\text{SIN}(90 * X/16) * 256$ where X ranges from zero to fifteen. What is observable is that as HRRROT increases, the angle from the X-axis increases, the value of the COS result decreases, and the value of the SIN result increases. This is all based on the rules of trigonometric functions in order to create the hypotenuse of a triangle whose base is the horizontal argument and whose height is the vertical argument. **The sum of all of the horizontal movements along with the sum of all of the vertical movements create the path of the hypotenuse that is taken by the HIRES pixel cursor.**

Index	ROTHVAL	ROTVVAL	Description
0	0x00	0x00	$\text{COS}(0) * 256 = 0, \text{SIN}(0) * 256 = 0$
1	0xFF	0x19	$\text{COS}(5.625) * 256 = 255, \text{SIN}(5.625) * 256 = 25$
2	0xFB	0x32	$\text{COS}(11.25) * 256 = 251, \text{SIN}(11.25) * 256 = 50$
3	0xF5	0x4A	$\text{COS}(16.875) * 256 = 245, \text{SIN}(16.875) * 256 = 74$
4	0xED	0x62	$\text{COS}(22.5) * 256 = 237, \text{SIN}(22.5) * 256 = 98$
5	0xE2	0x79	$\text{COS}(28.125) * 256 = 226, \text{SIN}(28.125) * 256 = 121$
6	0xD5	0x8E	$\text{COS}(33.75) * 256 = 213, \text{SIN}(33.75) * 256 = 142$
7	0xC6	0xA2	$\text{COS}(39.375) * 256 = 198, \text{SIN}(39.375) * 256 = 162$
8	0xB5	0xB5	$\text{COS}(45) * 256 = 181, \text{SIN}(45) * 256 = 181$
9	0xA2	0xC6	$\text{COS}(50.625) * 256 = 162, \text{SIN}(50.625) * 256 = 198$
10	0x8E	0xD5	$\text{COS}(56.25) * 256 = 142, \text{SIN}(56.25) * 256 = 213$
11	0x79	0xE2	$\text{COS}(61.875) * 256 = 121, \text{SIN}(61.875) * 256 = 226$
12	0x62	0xED	$\text{COS}(67.5) * 256 = 98, \text{SIN}(67.5) * 256 = 237$
13	0x4A	0xF5	$\text{COS}(73.125) * 256 = 74, \text{SIN}(73.125) * 256 = 245$
14	0x32	0xFB	$\text{COS}(78.75) * 256 = 50, \text{SIN}(78.75) * 256 = 251$
15	0x19	0xFF	$\text{COS}(84.375) * 256 = 25, \text{SIN}(84.375) * 256 = 255$

Table 5. Rotational Values Based on HRRROT

```

10 HOME
   D$ = CHR$( 4 )
20 PRINT D$; "SHLOAD SPOKE SHAPE,A$B000,B"
50 HGR
   POKE 49234, 0
60 HCOLOR = 3
70 SCALE = 11
80 HPLOT 0, 0 TO 279, 0 TO 279, 191 TO 0, 191 TO 0, 0
100 FOR R = 0 TO 63
110   ROT = R
120   DRAW 1 AT 140, 96
130 NEXT
140 IF PEEK( 49152 ) < 128 THEN GOTO 140
150 POKE 49168, 0
160 TEXT
170 END

```

Figure 9. Applesoft Spoke Program

In the unmodified 0xF0 ROM, the rotational values that are shown in Table 5 are calculated using a maximum value of 255 or 0xFF and that generates an illogical value for the very first table entry! Furthermore, the sum of the horizontal ROTHVAL values and the vertical ROTVVAL values are illogical when the C-flag is utilized as shown in Figure 13 indicating an overflow of 256 and **not** of 255 which is the maximum base for the calculation of **these** rotational values. This progressively becomes a more serious flaw particularly when SCALE values are implemented in the unmodified 0xF0 ROM.

The DRAWSHP routine returns to the top of the main loop as shown in Figure 13 at line #600 every time a new shape vector is extracted either from the existing SHPVAL value or from the next SHPVAL value in the SHAPE definition data as shown in Figure 14. In the unmodified 0xF0 ROM, the instructions after line #600 masks off the SHPVAL value and *always* initializes the horizontal and the vertical summation registers with the value of 0x80. Whatever for? Even when the next shape vector may contain the same identical direction of movement regardless if a pixel is drawn or not, the coordinate summation registers are re-initialized to this bizarre value. This is totally absurd and ridiculous. Does this initialization value suggest that the C-flag should be set earlier by the coordinate summation registers? If the same trigonometric values are in play while the shape vector direction of movement does not change, the coordinate summation registers must certainly **NOT** be re-initialized. I was completely flabbergasted when I analyzed the DRAWSHP routine in the unmodified 0xF0 ROM and found this algorithm to be grossly illogical. Thinking and saying this algorithm is grossly illogical is not quite as powerful as showing why and to the extent this algorithm is grossly illogical. A simple SHAPE definition is utilized for the Applesoft Spoke program that is shown in Figure 9. The Spoke SHAPE table contains four values of 0x2D, that is, eight shape vectors that have a value of %0101 for PLOT, move RIGHT. Figure 10 shows the display when this simple Applesoft program is run on an unmodified Enhanced Apple //e. Figure 11 shows the display when this same Applesoft program is run on an Enhanced Apple //e that contains my modified 0xF0 ROM. This modified 0xF0 ROM processes the Applesoft DRAW and XDRAW statements using the Applesoft routines that I have modified and are shown in Figures 8, 12, 13, and 14 as well as a ROTATBL table that contains the values that are shown in Table 5.

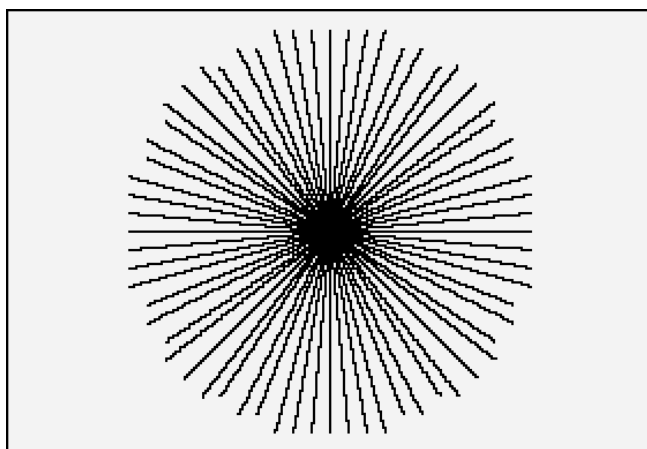


Figure 10. Unmodified ROM Spoke Processing

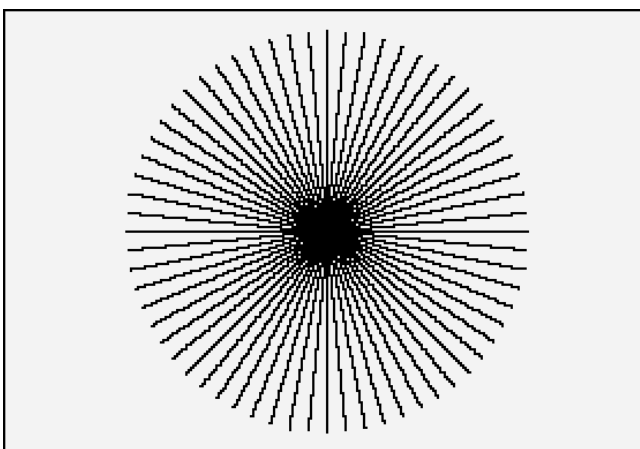


Figure 11. Modified ROM Spoke Processing

Even the 0xF0 ROM in the unmodified Enhanced Apple //e contains the HLIN routine modifications that I describe in the previous section *Management of Coordinate Displacements in HLIN*. I made these HLIN routine modifications so long ago that they reside in all of my 0xF0 ROM images. The visual distortions,

angle irregularities, and the length of all lines other than the precise horizontal and vertical axes are wrong in Figure 10. The DRAWSHP routine in the unmodified 0xF0 ROM is completely unacceptable and useless. For just eight extra bytes of code and a single page-zero variable SHPOLD at 0xD7, the DRAWSHP routine can be transformed into a very acceptable and useful tool in order to display the content of a SHAPE table as shown in Figure 11. Figure 11 demonstrates that the original initialization value of 0x80 for the coordinate summation registers is wrong and that these registers must **NOT** always be re-initialized for all following shape vectors. Figure 11 demonstrates that the values that are found in the unmodified ROTATBL table are wrong and the logic that was used to generate those table values is also wrong.

The section of the DRAWSHP routine that is shown in Figure 13 initializes the Y-register at line #587 from HRHORZ at 0xE5 and the X-register at line #612 from HRSCALE at 0xE7. For the remainder of this routine, only the A-register is used to process the coordinate summation registers, manipulate the HIRES screen, and process all HIRES pixel cursor movements except when the Y-register is appropriately incremented or decremented. That is an incredible undertaking. Take, for example, the processing of the coordinate summation registers. Assume that HRRROT is set to 0x08 so that both ROTHVAL and ROTVVAL registers will contain the value of 0xB5. The first summation of ROTHSUM and ROTVSUM will set these registers to 0xB5. The second to 0x6A with the C-flag set, the third to 0x1F with the C-flag set again, and the fourth to 0xD4. With the C-flag set twice, the overall sum of these registers is 0x2D4. The hypotenuse length for the first time the C-flag is set is $\sqrt{256^2 + 256^2}$ or 362. The second time the C-flag is set the hypotenuse length is again set to 362. For the remainder value of 0xD4, the hypotenuse length is $\sqrt{212^2 + 212^2}$ or 300. The total hypotenuse length is now $362 + 362 + 300 = 1024$. For a total coordinate length of 0x2D4, the hypotenuse length is $\sqrt{724^2 + 724^2} = 1024$. This example demonstrates that the coordinate summation registers ROTHSUM and ROTVSUM must be initialized to 0x00 and if the same trigonometric coordinate values are in play while the shape vector direction of movement does not change, the coordinate summation registers must **NOT** be re-initialized. Therefore, solely based on the setting of the C-flag will the HIRES screen be manipulated starting at DRAWHDR. After DRAWHDR processing, the setting of the C-flag at lines #626 and #638 is finally utilized at line #208 in order to precisely move the HIRES pixel cursor to its next calculated position on the HIRES screen. The DRAWSHP and the DRAWHDR routines are amazingly thorough in not only manipulating HIRES pixels ON but also HIRES pixels OFF throughout DRAWHDR processing.

The unmodified ROM does not contain the instructions that appears on lines #179 and #184 in Figure 12. Whatever is written in the Applesoft reference manual on page 98 for XDRAW is incorrect. XDRAW was designed originally to XDRAW a shape using only White1 or 0x7F and **no other color**. And that is precisely what the unmodified ROM code produces. Perhaps this is the time to explain what are the actual services of DRAW and of XDRAW. First and foremost, DRAW and XDRAW have no relationship or interdependencies and these two Applesoft statements are **not** designed to be used in conjunction with the other. DRAW 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 HIRES pixels that are currently being displayed. Another way to remove the present SHAPE definition is to paint the background color or new colors over the present SHAPE definition. DRAW does not incorporate any of the old HIRES pixel information with any of the pixel information data of the new SHAPE definition. On the other hand, XDRAW 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 coordinates. As with all graphic routines that make use of the eor microprocessor instruction, color complements must be taken into consideration when using the Applesoft XDRAW statement. Table 6 shows the shape color to expect with various HCOLOR values when drawing on various background colors.

:	:	:
F49E A5 D0	168	DRAWHDR lda SHPVAL
F4A0 29 04	169	and #PLOTMASK
F4A2 F0 24	170	beq >3
F4A4	171	;
F4A4 24 44	172	bit OPRND ; XDRAW/DRAW flag
F4A6 10 12	173	bpl DRAWIT
F4A8	174	;
F4A8	175	;
F4A8	176	; XDRAW code.
F4A8	177	;
F4A8 B1 26	178	XDRAWIT lda (GBASL),Y
F4AA 25 1C	179	and COLBITS ; added to support color
F4AC 25 30	180	and COLOR
F4AE 29 7F	181	and #MSBCLR
F4B0 D0 12	182	bne >2
F4B2	183	;
F4B2 A5 1C	184	lda COLBITS ; added to support color
F4B4 25 30	185	lda COLOR
F4B6 29 7F	186	and #MSBCLR
F4B8 10 08	187	bpl >1 ; always taken
F4BA	188	;
F4BA	189	;
F4BA	190	; DRAW code.
F4BA	191	;
F4BA B1 26	192	DRAWIT lda (GBASL),Y
F4BC 45 1C	193	eor COLBITS
F4BE 25 30	194	and COLOR
F4C0 D0 02	195	bne >2
F4C2	196	;
F4C2 E6 EA	197	^1 inc HRCOLCNT ; collision counter
F4C4	198	;
F4C4	199	;
F4C4	200	; Common XDRAW/DRAW code.
F4C4	201	;
F4C4 51 26	202	^2 eor (GBASL),Y
F4C6 91 26	203	sta (GBASL),Y
F4C8	204	;
F4C8	205	;
F4C8	206	; C-flag clear in horz summation and set in vert summation.
F4C8	207	;
F4C8 A5 D0	208	^3 lda SHPVAL
F4CA 65 D1	209	adc ROTQVAL
F4CC	210	;
F4CC 29 03	211	and #MOVEMASK
F4CE C9 02	212	cmp #2 ; for down or left
F4D0	213	;
F4D0	214	;
F4D0	215	; ROR sets MSB if move direction is down or left and C-flag
F4D0	216	; is set if move direction is right or left.
F4D0	217	;
F4D0 6A	218	ror
F4D1 B0 92	219	bcs HRMOVLF
F4D3	220	;
F4D3	221	;
F4D3 30 30	222	HRMOVUP bmi HRMOVDN ; branch if down
:	:	:

Figure 12. The DRAW and XDRAW Screen Routines

```

:           :           :
F620 A4 E5   587         ldy HRHORZ
F622         588         ;
F622 A2 FF   589         ldx #NEGONE
F624 86 D7   590         stx SHPOLD
F626         591         ;
F626 E8      592         inx
F627 86 EA   593         stx HRCOLCNT           ; initialize to zero
F629         594         ;
F629 A1 1A   595         lda (SHAPE,X)
F62B         596         ;
F62B         597         ;
F62B         598         ; If there is no change in direction, keep summing axes.
F62B         599         ;
F62B 85 D0   600 ^1      sta SHPVAL
F62D 29 07   601         and #SCMDMASK
F62F         602         ;
F62F C5 D7   603         cmp SHPOLD
F631 F0 08   604         beq >2
F633         605         ;
F633 85 D7   606         sta SHPOLD
F635         607         ;
F635 A9 00   608         lda #ZERO
F637 85 D4   609         sta ROTHSUM
F639 85 D5   610         sta ROTVSUM
F63B         611         ;
F63B A6 E7   612 ^2      ldx HRSCALE           ; get requested SCALE value
F63D         613         ;
F63D         614         ;
F63D         615         ; Horizontal summation.
F63D         616         ;
F63D 38      617 ^3      sec
F63E         618         ;
F63E A5 D4   619         lda ROTHSUM
F640 65 D2   620         adc ROTHVAL
F642 85 D4   621         sta ROTHSUM
F644 90 05   622         bcc >4
F646         623         ;
F646 18      624         clc
F647         625         ;
F647 20 9E F4 626         jsr XDRAWHDR
F64A         627         ;
F64A 18      628         clc
F64B         629         ;
F64B         630         ;
F64B         631         ; Vertical summation.
F64B         632         ;
F64B A5 D5   633 ^4      lda ROTVSUM
F64D 65 D3   634         adc ROTVVAL
F64F 85 D5   635         sta ROTVSUM
F651 90 03   636         bcc >5
F653         637         ;
F653 20 9E F4 638         jsr XDRAWHDR
F656         639         ;
F656 CA      640 ^5      dex
F657 D0 E4   641         bne <3
:           :           :

```

Figure 13. The SCALE Loop Routine in DRAWSHIP

:	:	:
F659 A5 D0	643	lda SHPVAL
F65B	644	;
F65B 4A	645	lsr
F65C 4A	646	lsr
F65D 4A	647	lsr
F65E	648	;
F64E D0 CB	649	bne <1
F660	650	;
F660	651	;
F660	652	; Point to next shape in table.
F660	653	;
F660 E6 1A	654	inc SHAPE
F662 D0 02	655	bne >6
F664	656	;
F664 E6 1B	657	inc SHAPE+1
F666	658	;
F666 A1 1A	659	lda (SHAPE,X)
F668 D0 C1	660	bne <1
F66A	661	;
F66A 60	662	rts
:	:	:

Figure 14. The SHPVAL Loop Routine in DRAWSHP

HCOLOR	Background Color							
	Black1 0x00/00	Green 0x2A/55	Purple 0x55/2A	White1 0x7F/7F	Black2 0x80/80	Orange 0xAA/D5	Blue 0xD5/AA	White2 0xFF/FF
0	no change	no change	no change	no change	no change	no change	no change	no change
1	Green	Black	White	Purple	Orange	Black	White	Blue
2	Purple	White	Black	Green	Blue	White	Black	Orange
3	White	striped mix	striped mix	~ Black	White	striped mix	striped mix	striped mix
4	no change	Black	no change	no change	no change	no change	no change	no change
5	Green	Black	White	Purple	Orange	Black	White	Blue
6	Purple	White	Black	Green	Blue	White	Black	Orange
7	White	striped mix	striped mix	~ Black	White	striped mix	striped mix	striped mix

Table 6. XDRAW Shape Colors with Background Colors

Colors in particular require two values in order to set the HIRES screen to a solid background color. One value is used for the even numbered bytes and the other value is used for the odd numbered bytes. Black and white background colors use only a single value for both the even and the odd numbered bytes. Lines #181 and #186 in Figure 12 ensure that the most significant bit in HCOLOR is not utilized nor can it be utilized for HCOLOR 4, 5, 6, and 7. The results for these colors are the same for HCOLOR 0, 1, 2, and 3 as shown in Table 6. A very unappealing mixture of colors that appear stripped results when a SHAPE definition is drawn using HCOLOR 3 or 7 over colored backgrounds. On the other hand, a very appealing white SHAPE definition is drawn using HCOLOR 1 or 2 (or 5 or 6) over both sets of its compliment background color. It amazes me how little testing Randy Wigginton and Cliff Huston must have done when they

designed their XDRAWIT routine and limited their routine to drawing a SHAPE definition that is only white in color no matter the setting of HCOLOR and without regard to the background color. How impressive is that? Table 6 shows that the color of any SHAPE definition that is drawn on a HIRES screen by the modified ROM is partially determined by HCOLOR and partially determined by the setting of the most significant bit of the background color. Other objects and SHAPE definitions that are drawn in the vicinity of a new SHAPE definition may be the prime driving forces in determining the final color of a new SHAPE definition as well as its bit placement within its bytes, whether those bytes are even or odd, and for the specific HIRES line.

Without knowing any more of the history of the development of the Applesoft interpreter when the early Apple II 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 routines that process the specific handlers of their Applesoft statements. Clearly, the DRAW and the XDRAW routines are not thoroughly well designed. The choice to use the exponentiation routine in order to calculate a square root rather than implementing a simple Newton-Raphson iteration routine is another example of choosing an inferior design. Using floating-point variables that are processed using floating-point routines in order to calculate a random number is the epitome of incompetence particularly in view of attempting to utilize a standard LCG equation that only accepts integers that are processed using only integer routines. Even though the two floating-point register copy routines at 0xEB53 and 0xEB63 are condensed processor register loops, more than forty extra processor cycles are consumed and wasted each time one or the other routine is utilized by other floating-point routines. For involved complex floating-point number utilization in manipulating matrices of any size, adding these extra processor cycles could very well have a clear impact on overall processing time and processing duration. Even small refinements in not switching the TEXT screen to the HIRES screen until the HIRES screen memory is fully initialized is just one example where a better choice will produce a more elegant result. I have no doubt that some testing was invariably utilized while the graphic routines were being developed for the early Applesoft interpreter. I do not believe that sufficient testing was performed or perhaps the test results were simply ignored in haste. It still amazes me that the Applesoft interpreter has never been updated even once during the production life of the Apple II computer. What amazes me even more is that the code space for all of the improvements that I have added to the modified Applesoft ROM have been fully covered by intelligently rewriting the DRAW and the XDRAW handlers for these Applesoft statements. And yet, there is still some code space remaining for further improvements. Having a software license may become an unfortunate detriment when software errors, incompetent routines, and gross negligence can never be remedied.

SHAPE Table Management

SHAPE Manager is an assembly language program that does not utilize any of the flawed HIRES drawing routines that are found in the unmodified Applesoft ROM. All of the HIRES drawing routines are contained within the *SHAPE Manager* program whether the user is utilizing a modified Applesoft ROM or not. Therefore, *SHAPE Manager* is not concerned with whether a SHAPE will or will not be correctly and precisely drawn when a machine is still utilizing an unmodified ROM. Perhaps the user may become so fond of how lovely the *SHAPE Manager* HIRES drawing routines draw lines and shapes that they may be encouraged to install a modified Applesoft ROM into their own machine. I have no doubt that I would be so inclined if I observed all of the capabilities that these modified Applesoft HIRES drawing routines provide. And, in combination with SHAPE table file handling in DOS 4.5 Build 08, *SHAPE Manager* provides all of the functions and the capabilities in order to generate any and all SHAPE tables that might be required by a number of versatile programs and utilities which could even include HIRES games.

SHAPE Manager utilizes several Draw ICON routines that accelerate the drawing of lines in order to create the SHAPE drawing windows. These same Draw ICON routines also reside in *ICON Maker*, another assembly language program I wrote that can be used to design and build HIRES icons. A thorough discussion of *ICON Maker* can be found in *DOS 4.5 Volume and File Disk Management System Second Edition*. Draw ICON utilizes lookup tables that are necessary in order to draw lines and shapes at extraordinary speeds. DRAW and XDRAW both depend on the same routines that calculate HIRES addresses for the HLIN routine that the Applesoft HPLLOT statement depends on. The lookup tables that are used for Draw ICON are large in size so they can be considered a very expensive option in view of ROM utilization and, therefore, not as practical as the address calculation routines like those that are found in the Applesoft ROM. On the other hand, routines that incorporate lookup tables are very practical when they are utilized in assembly language programs or in hybrid Applesoft program like *BFI*, a program that attaches relocatable assembly language routines to an Applesoft program. *BFI* itself uses Draw ICON in order to draw all of the HIRES icon images that *BFI* uses in order to select its various processing algorithms.

A SHAPE drawing window is a defined area on the HIRES graphic screen where a SHAPE definition from a SHAPE table is drawn. The location of the first pixel that is drawn from the data of a SHAPE definition is always relative to the upper left-hand corner of its drawing window. Also, that first SHAPE definition pixel is at some index bit that is within some index byte from the left side of screen and some index scan line from the top of the screen. In other words, the SHAPE definition start location is some number of pixels to the left from the left side of the HIRES screen and some number of pixels or scan lines down from the top of the HIRES screen. When the drawing window is defined, its location is specified by its X- and Y-coordinates in pixels. These coordinates are used additively to the SHAPE definition data when drawing each pixel that is defined by each DRAWHDR vector that are contained within the SHAPE definition data bytes.

In order to speed up and accelerate the drawing of lines in Draw ICON or in any other utility or game that utilizes HIRES graphic routines and animation, lookup tables are necessary in order to obtain maximum calculation speed. Given an X-location on a scan line that is some number of pixels from the left side of the screen, that location must be converted into a bit index that is within some byte index. There are forty bytes of memory that comprise each visible scan line and these forty bytes contain the data for the 280 pixels that can be displayed which amounts to seven pixels for each byte of data. Obviously, converting pixel number to byte index requires the division by seven with some bit index remainder in pixels. This is not an easy integer division to implement without using lookup tables. The first set of lookup tables is called XBASEL and XBASEH, and these tables accomplish this division easily at the expense of 0x118 bytes of data. XBASE simply determines the byte index from the base screen location that is found in GBAS, a page-zero pointer at 0x26:0x27. Horizontal pixel number is also used to index into a MASKNDX table in order to determine which index bit will be turned OFF or turned ON within its byte index, so this table provides the value for the remainder in the division by seven operation. The MASKNDX table is 0x100 bytes in size. In order to support color on the HIRES graphic screen, Draw ICON uses a COLORNDX table that is 0x100 bytes in size and this table is also indexed by horizontal pixel number. COLORNDX determines which color byte is selected from an 8-byte COLORBYT table. Specific bytes are copied from the COLORTBL table in order to form the COLORBYT table that masks in a selected color for the graphic line that is currently being drawn. The COLORTBL is 0x20 bytes in size. Already, 0x338 bytes of table data are required just to support the X-location in pixels in order to accelerate the calculations for drawing a graphic line by Draw ICON.

The Y-location in pixels or scan lines is used to initialize the base screen location or address for the page-zero pointer GBAS. Steven Wozniak constructed the Apple][HIRES graphic screen in three distinct sections where each section consists of eight TEXT lines and each TEXT line consists of eight scan lines. Therefore, there are a total of 192 scan lines that are available for the entire HIRES graphic screen that is visible. Within each of the three screen sections, the address of one TEXT line to the next TEXT line or group of eight scan

lines to the next group of eight scan lines is incremented by $0x80$, and each screen section to the next section is incremented by $0x28$. Within one TEXT line, its eight scan lines are each incremented by $0x400$. Wozniak's design of the Apple][hardware that displays the data that resides in the HIRES graphic screen memory requires the least number of hardware components when that memory data is addressed for display using these address specifications. In view of these address specifications, it is quite understandable that using a lookup table in order to initialize GBAS by using scan line as the index into a YBASE table will definitely accelerate the calculations for drawing lines by Draw ICON. YBASE actually consists of two tables called YBASEL and YBASEH that define the full 16-bit base memory address for the start of each scan line. Obviously, these two tables are each 192 bytes in size. If Applesoft HGR2 is supported, a second set of YBASE tables would be required.

The total number of bytes that are required for all of the lookup tables that are used to accelerate the calculations for drawing lines in Draw ICON is $0x4B8$ bytes for one HIRES graphic screen. Is using nearly five pages of memory just for lookup tables really worth it? You bet it is! When lookup tables are utilized, a line can be drawn nearly instantaneously. If the address calculation procedures that are found in the Applesoft HPLLOT routine at $0xF457$ and in the Applesoft HLIN routine at $0xF53A$ are used instead, a line would be drawn comparatively at a snail's pace. There is **no** faster method to accelerate the calculations for drawing lines on the Apple][HIRES graphic screen than using these lookup tables in order to initialize the base screen address, to establish the correct index byte, to select the index bit that is within that index byte, and to utilize the correct color mask. Another advantage for not using the Applesoft HPLLOT and the Applesoft HLIN handler routines is that the HLIN routine is fundamentally flawed as I pointed out in *Management of Coordinate Displacements in HLIN*. My analysis of the assembly language instructions for the HLIN routine shows that the routine does not correctly calculate the delta difference of the horizontal and of the vertical start to end points before the routine draws a line. This calculation error severely affects the appearance of all diagonally drawn lines in my opinion. The HLIN routine that is used in Draw ICON does not contain these flaws. All diagonal lines are drawn precisely and diagonal lines are segmented equally in all instances and the results will always be the same without regard to the direction in which the lines are drawn. The original Applesoft HLIN handler routine cannot make these same guarantees.

The Applesoft reference manual contains Chapter 9 on *High-Resolution Shapes*. This chapter consists of eight sections that include *How to Create a Shape* on page 92, *Saving a Shape Table* on page 97, *Using a Shape Table* on page 98, *DRAW* on page 98, *XDRAW* on page 98, *ROT* on page 99, *SCALE* on page 99, and *SHLOAD* on page 99. It is mandatory for anyone who wishes to make use of a SHAPE table within the context of an Applesoft program to read and to fully understand the contents of Chapter 9 in its entirety except for the section on *SHLOAD*. The modified Applesoft ROM no longer contains the routines that are necessary to read a SHAPE table into memory by means of a cassette tape recorder using the Applesoft SHLOAD statement since the Applesoft SHLOAD handler has been removed. DOS 4.5 Build 08 is fully capable of loading a SHAPE table into memory from an S Type $0x08$ file that is in a DOS volume as well as saving a SHAPE table from memory into an S Type $0x08$ file that is in a DOS volume.

The Applesoft reference manual states that a SHAPE table may contain up to 255 shape definitions. A shape definition consists of a sequence of plotting vectors that are stored in a series of bytes. Each byte within a shape definition contains three sections where each section may define a Plot vector or a Move vector. Plot vector A uses bits 0:2, Plot vector B uses bits 3:5, and Move vector C uses bits 6:7. The last data byte in a shape definition is always zero. Plot vectors A and B use their lower two bits to define a move direction and their upper bit to define whether to draw a pixel if the bit is set or not to draw a pixel if the bit is not set. Move vector C can only define a move direction other than UP. Move directions are defined as $\%00$ for UP, $\%01$ for RIGHT, $\%10$ for DOWN, and $\%11$ for LEFT. For example, the Plot vector to draw and to move RIGHT would be $\%101$. It is important to understand that the DRAWSHP routine first implements pixel

management to draw or not to draw when DRAWHDR looks at the plot bit and then the routine implements move management to move the HIRES pixel cursor. And, it is important to remember that when the DRAWSHP routine prepares to process the next Plot vector or Move vector, the routine will only process a value that is not zero. Furthermore, when the DRAWSHP routine reads the next byte from a SHAPE table and that value is zero, the routine does not process any further bytes for that shape definition.

These Plot vector, Move vector, move direction, and termination rules all determine what a shape definition byte may contain and what a shape definition byte may not contain. The *SHAPE Manager* program must employ these same rules and restrictions when a user constructs a shape definition and only a Plot or a No Plot bit followed by pixel cursor movement bits may be used. The most troublesome of all vectors is the No Plot UP Plot vector and the Move UP vector because both of these vectors have a value of zero. The Move vector can be used for any movement except for the UP direction. The No Plot UP Plot vector cannot be used alone; therefore, this vector must be used with either another Plot vector that is not zero or used with any Move vector that is not zero. Thus, a No Plot UP vector can only be used in vector A since vector B and vector C cannot equal zero according to Line #649 in Figure 14. A No Plot UP vector can never be used for vector B. In my experience, I have found that it is better to begin by drawing the desired shape on graph paper and simply draw a circle in the squares where a pixel is ON and an arrow showing the direction of movement to the next pixel. Once the complete shape is drawn, select a starting point that would tend to avoid the UP direction that contains a No Plot pixel, and definitely avoid the UP direction that contains successive No Plot pixels. There is absolutely no problem utilizing the UP direction as long as that direction also draws a pixel.

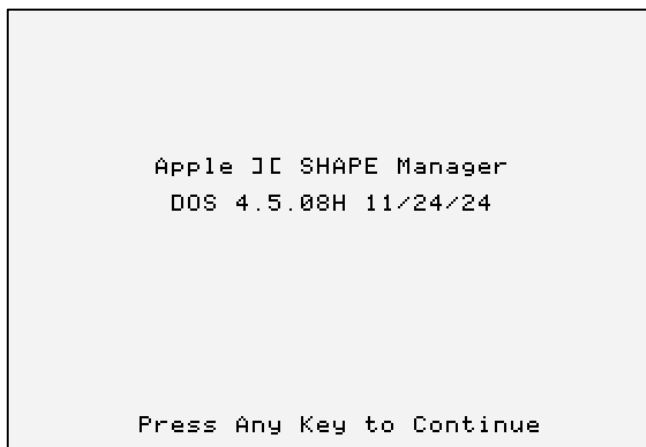


Figure 15. SHAPE Manager Introduction

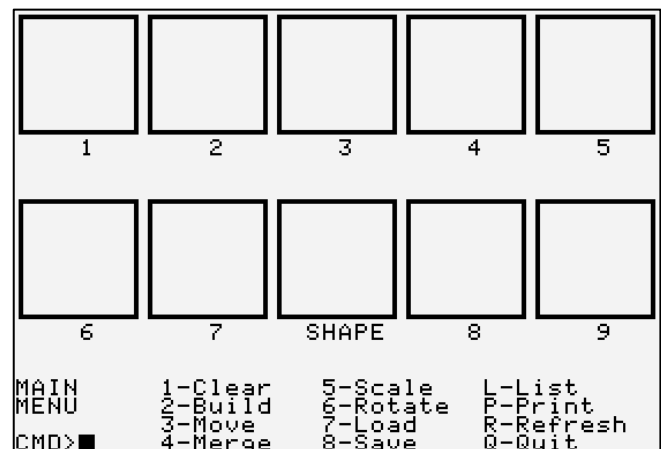


Figure 16. SHAPE Manager Main Menu

The Apple II *SHAPE Manager* program introduction is shown in Figure 15 when that program first begins its processing. *SHAPE Manager* is written having DOS 4.5.08H as its required disk operating system since the DOS SHLOAD and the DOS SHSAVE commands are utilized and these DOS commands are required by this program. Figure 16 shows the Main Menu for *SHAPE Manager*. The Main Menu allows easy access to all of the capabilities and the functions that are available in *SHAPE Manager*. *SHAPE Manager* is designed to provide nine Build windows and one Merge window where all of the various shape definitions can be combined to form one HIRES SHAPE table. *SHAPE Manager* provides many possibilities that can be utilized to easily design a HIRES SHAPE table in a short amount of time. The ten windows that are utilized in *SHAPE*

Manager are drawn by Draw ICON routines for their WIDTH of 48, for their HEIGHT of 48, and for their THICKNESS of 2. A WIDTH of 48 provides 17% of the HIRES screen width and a HEIGHT of 48 provides 25% of the HIRES screen height. There is certainly more than ample space to create any SHAPE table that can be used to define a HIRES shape or even a HIRES shape that can be animated. *SHAPE Manager* only uses the Applesoft XDRAW statement and the capabilities of the DRAWSPH routine for its program functions. Therefore, a user must be very well acquainted with Table 6 when selecting a color for a shape and using its correct color complement against the intended background color. The Applesoft DRAW statement is far more straightforward to use in many respects. However, shapes that are drawn by DRAW cannot easily be programmatically removed from the HIRES screen as easily as those shapes that are drawn by XDRAW.

Selecting Option 1 from the *SHAPE Manager* Main Menu allows the user to clear the selected Build window or the Merge window and that selection requires a frame number as shown in Figure 17. If a shape definition does not exist in the selected Build or Merge window, *SHAPE Manager* issues the warning message that is shown in Figure 18. If a shape definition does exist in the selected Build or Merge window as shown in Figure 19, *SHAPE Manager* issues the completion message that is shown in Figure 20.

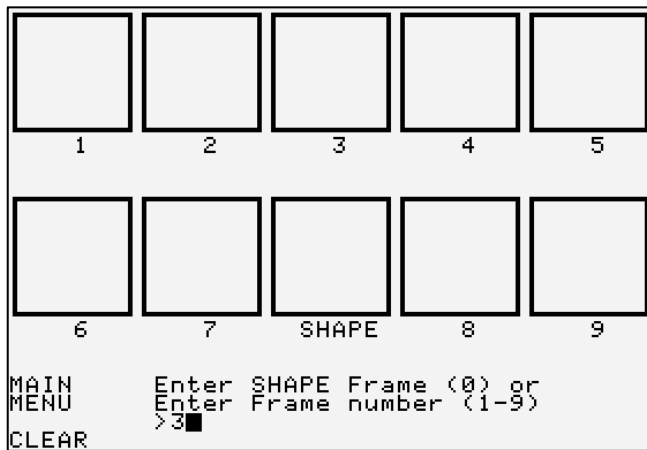


Figure 17. Enter Clear Frame Number

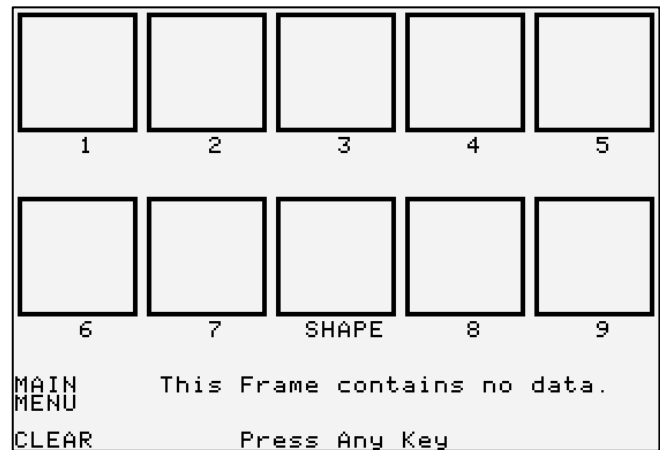


Figure 18. Clear Warning Message

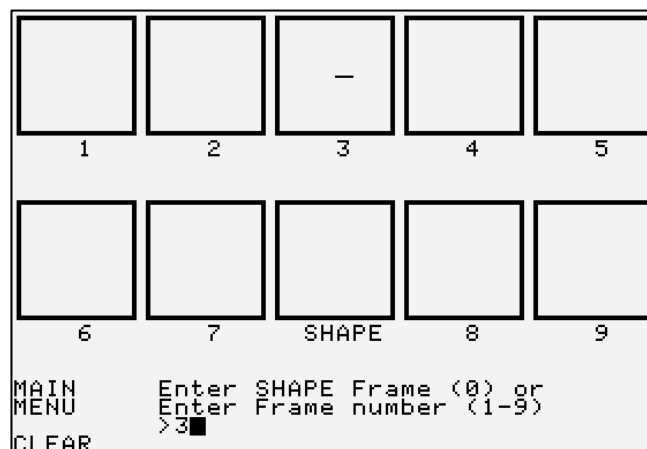


Figure 19. Enter Clear Frame Number

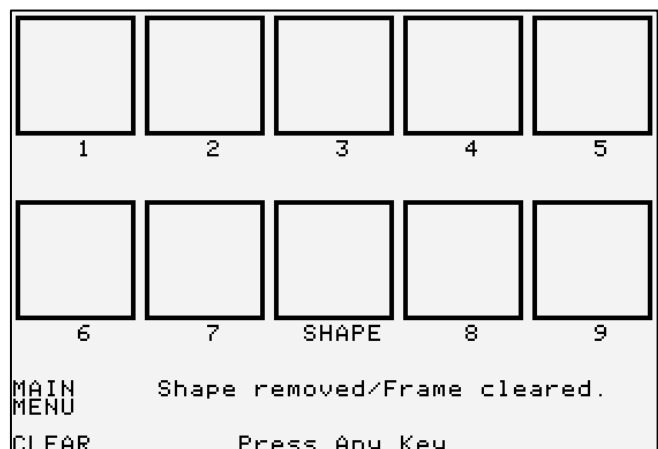


Figure 20. Clear Completion Message

Selecting Option 2 from the *SHAPE Manager* Main Menu allows the user to build a SHAPE definition within the selected Build window as shown in Figure 21. The color of the intended shape is selected next as shown in Figure 22. Figure 23 begins the cyclic loop of Build that first requests whether to draw a pixel or not and then Build requests the Move direction that is shown in Figure 24. Once the desired Arrow Key is pressed, the results of this first cycle of inputs is displayed in the lower right-hand corner as shown in Figure 25. The key for these results is N for Number of Plot and Move vectors, I for SHAPE table Index, F for next vector Field number, and D for current vector Data. The value for D that is shown in Figure 25 is for a Plot RIGHT vector. A No Plot UP vector is entered in Figure 26 and this causes *SHAPE Manager* to issue the warning message in Figure 27. Once any key is pressed, Figure 28 shows the current status that three vectors have been created, a No Plot UP vector has been entered, and *SHAPE Manager* is ready to receive the second vector for the current vector data byte. This vector must either draw a pixel or not move UP.

Once a Plot UP vector is entered in Figure 28, Figure 29 shows the complete summary of all data that has been entered as well as displaying the complete shape. *SHAPE Manager* issues the completion message that is shown in Figure 30 when Q is entered.

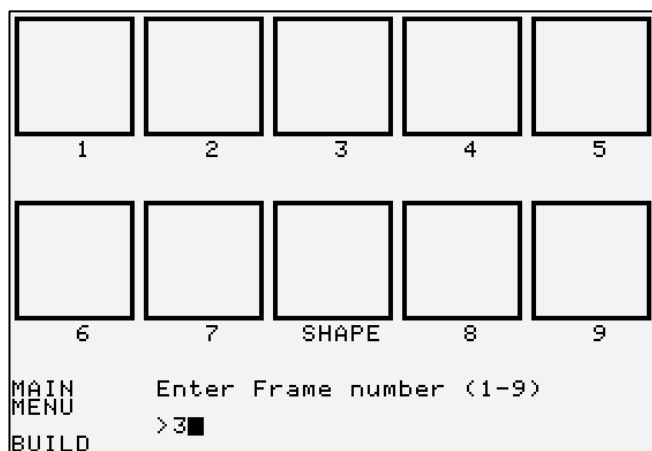


Figure 21. Enter Build Frame Number

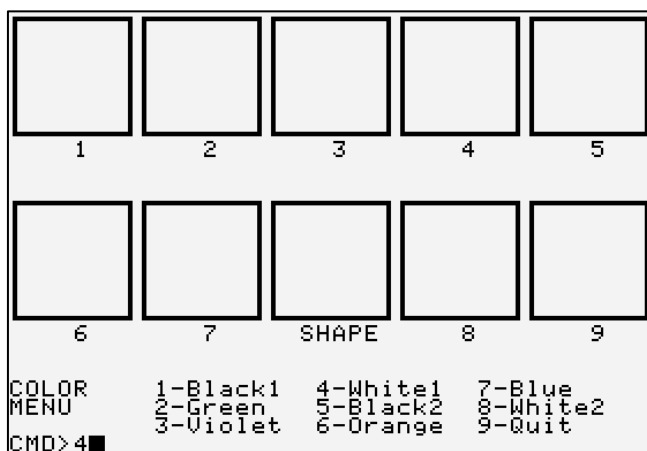


Figure 22. Enter Build Color Number

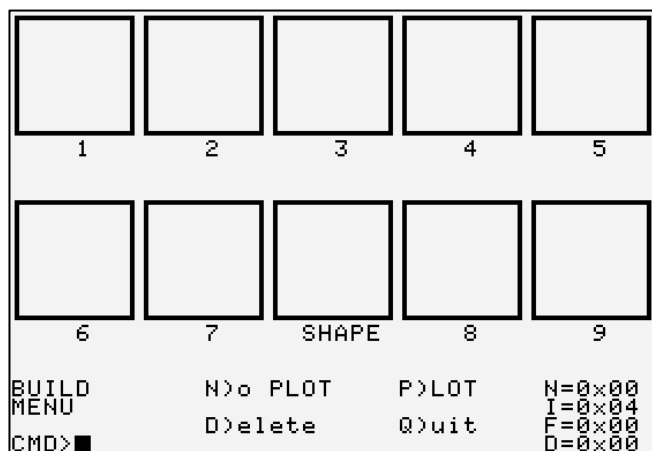


Figure 23. Enter Build Pixel ON/OFF

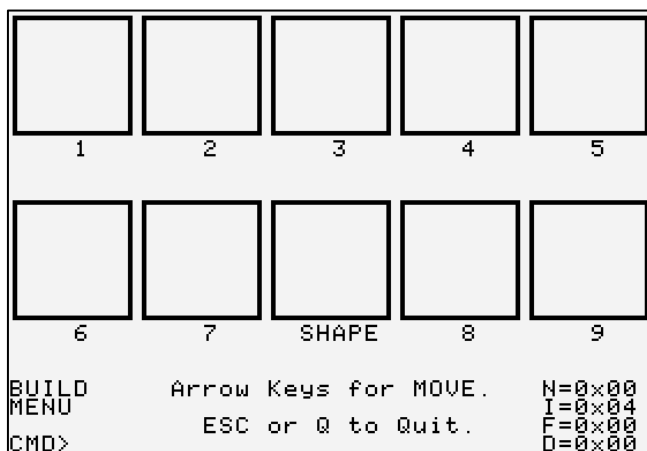


Figure 24. Enter Build Move Direction

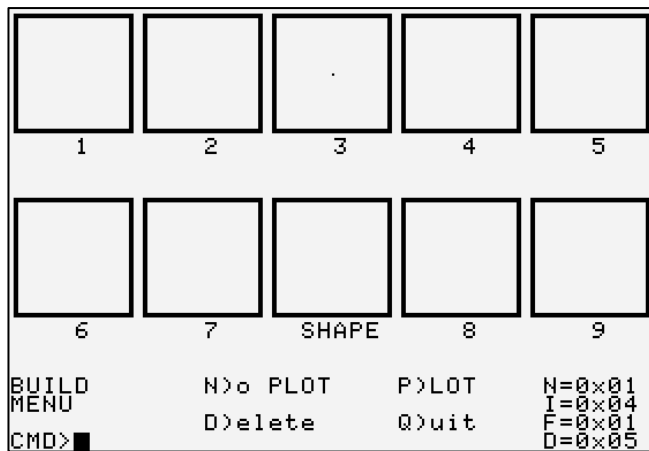


Figure 25. Enter Build Pixel ON/OFF

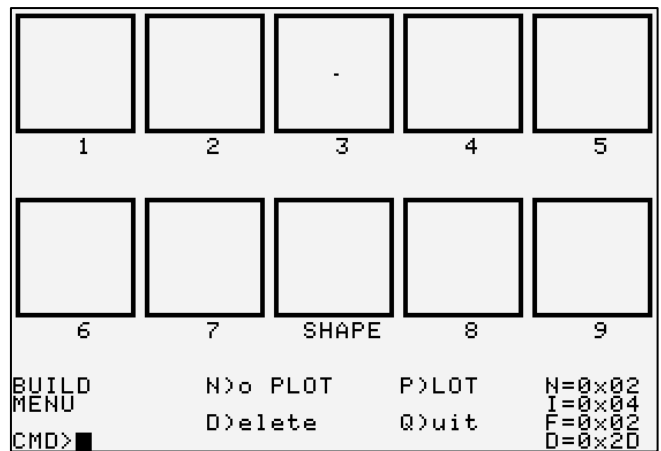


Figure 26. Enter Build Move Direction

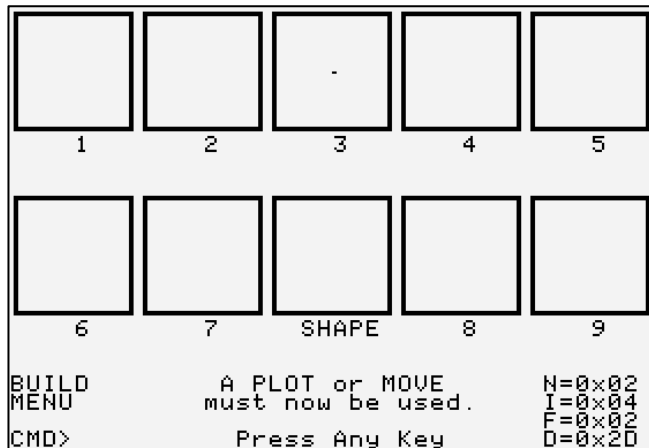


Figure 27. Build Info Message

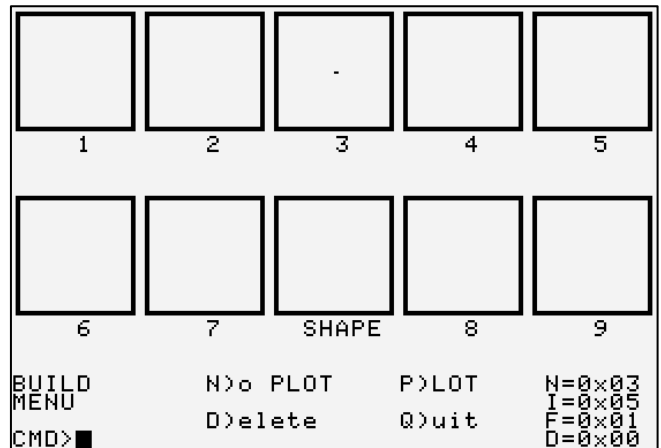


Figure 28. Enter Build Pixel ON/OFF

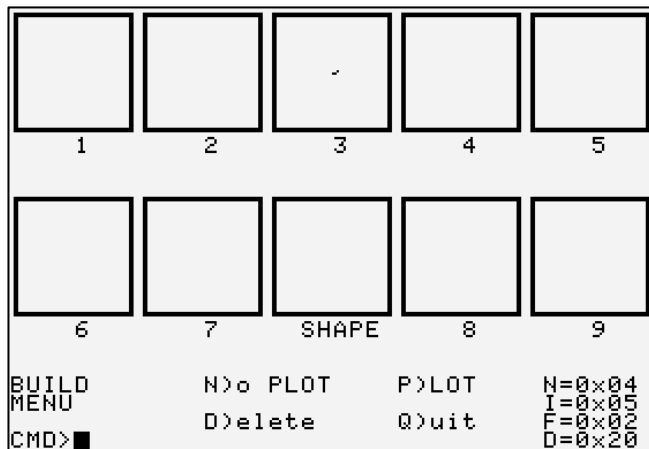


Figure 29. Enter Build Pixel ON/OFF

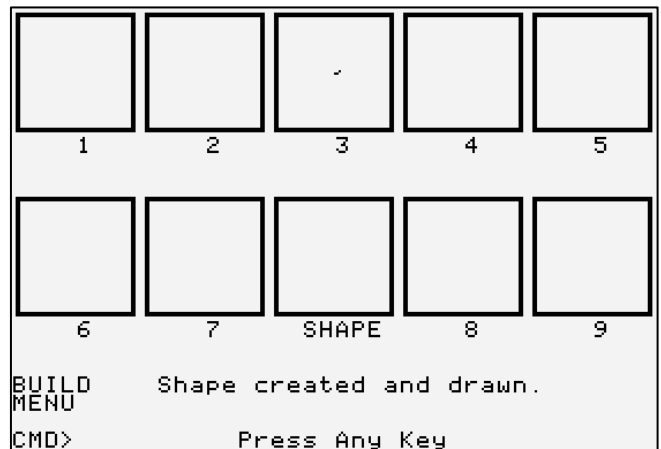


Figure 30. Build Completion Message

The last Plot or Move vector may be deleted from the SHAPE definition data by using the D option for delete as shown in Figure 29. The D option may be used successively until all data vectors have been removed from the SHAPE definition data. Each time the D option is entered in order to delete a data vector, *SHAPE Manager* recalculates the N, I, F, and D variables for display and the value of those variables assist *SHAPE Manager* in adding new Plot or Move vectors into the selected SHAPE definition data buffer.

When the data of a SHAPE definition is read into the buffer of a selected Build window, *SHAPE Manager* quickly pre-processes all of the data vectors in that buffer starting with the first vector. *SHAPE Manager* utilizes the N, I, F, and D variables for that pre-processing in order to arrive at their final value as if each Plot and Move vector had been entered manually. Doing this pre-processing ensures that the final value of each of the N, I, F, and D variables is correct and they can be used to either remove a data vector or to add a new data vector to the selected SHAPE definition. *SHAPE Manager* is not designed to manage any data vector or any set of data vectors that are inconsistent with how DRAWSHP is designed to draw Plot vectors and to process Move vectors. If inconsistent SHAPE definition data is submitted and the user expects Build to manage these data vectors, *SHAPE Manager* may very well abort its routines, mismanage the data vectors, or even fail to respond to further user input. It is always the responsibility of the user to provide SHAPE definition data that is consistent with the rules for the format of SHAPE definition Plot and Move vectors.

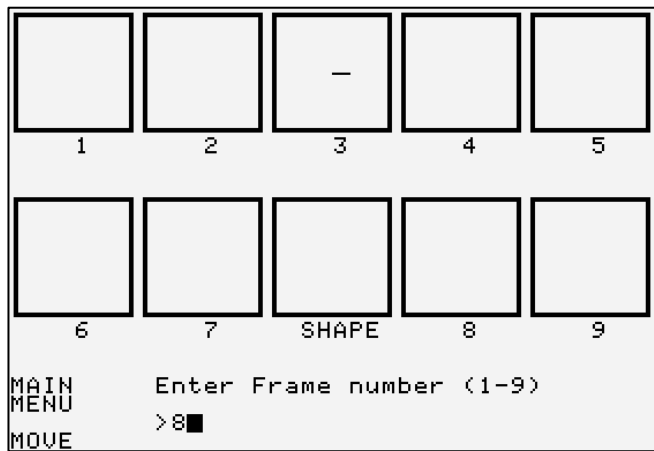


Figure 31. Enter Move Frame Number

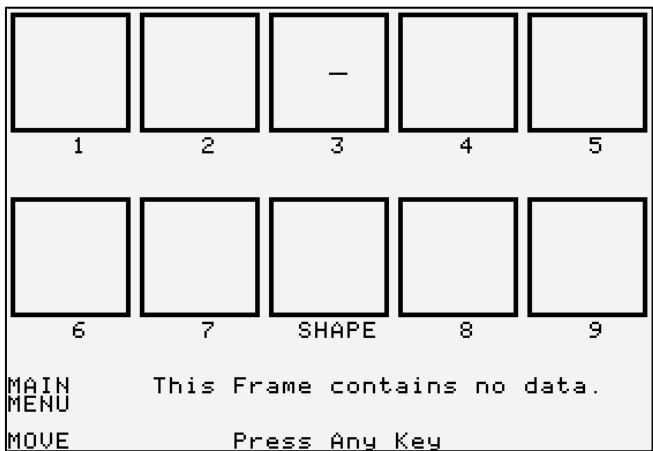


Figure 32. Move Warning Message

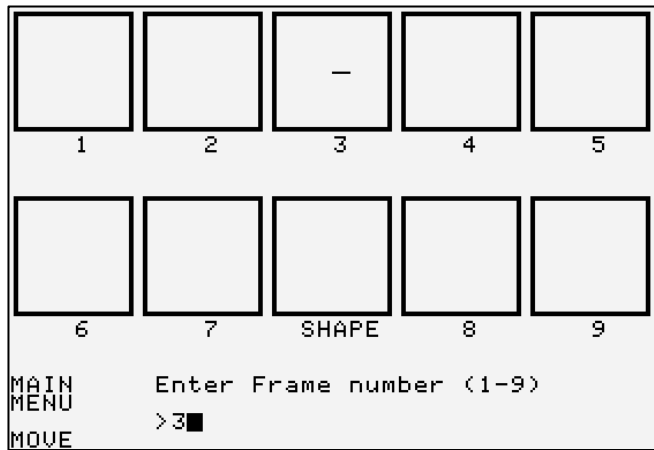


Figure 33. Enter Move Frame Number

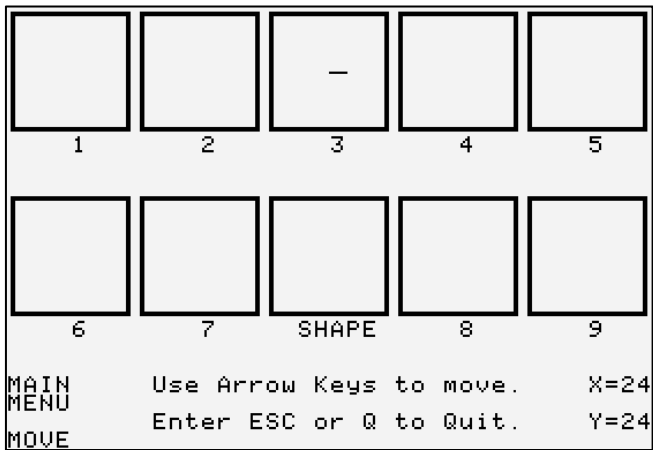


Figure 34. Enter Move Move Direction

Selecting Option 3 from the *SHAPE Manager* Main Menu allows the user to move the shape definition within the selected Build window as shown in Figure 31. If a shape definition does not exist in the selected Build window, *SHAPE Manager* issues the warning message that is shown in Figure 32. If a shape definition does exist in the selected Build window as shown in Figure 33, *SHAPE Manager* will move that shape definition one scan line in either the UP or the DOWN direction or one pixel in either the LEFT or the RIGHT direction solely based on which Arrow key is pressed. An Arrow key may be pressed any number of times. The start coordinates for the X and the Y locations that are relative to the upper left-hand corner are displayed in the lower right-hand corner as shown in Figure 34. These are the two coordinates that are utilized in order to draw the first pixel of the first Plot vector that is contained in the SHAPE definition that resides in the data buffer for the selected Build window. These coordinate values are also included by the List and by the Print Main Menu options. When the Q option is entered as shown in Figure 35, *SHAPE Manager* issues the completion message that is shown in Figure 36.

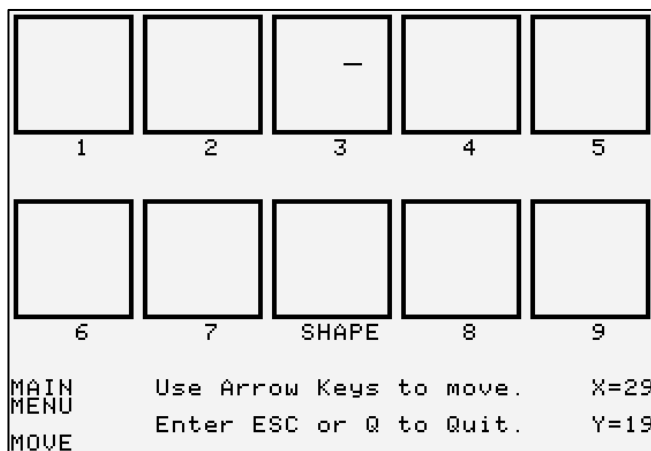


Figure 35. Move X and Y Coordinates

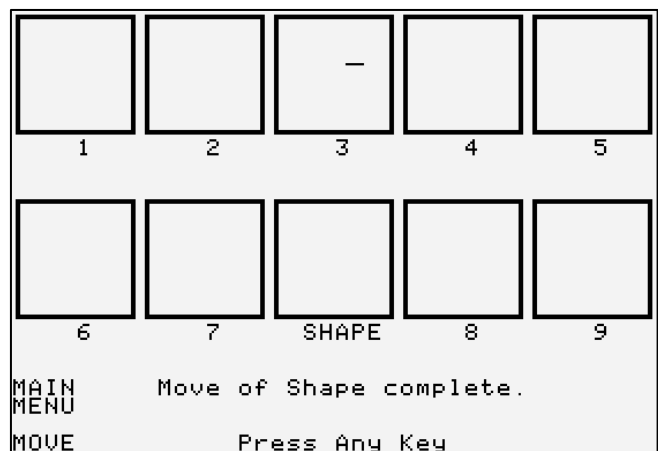


Figure 36. Move Completion Message

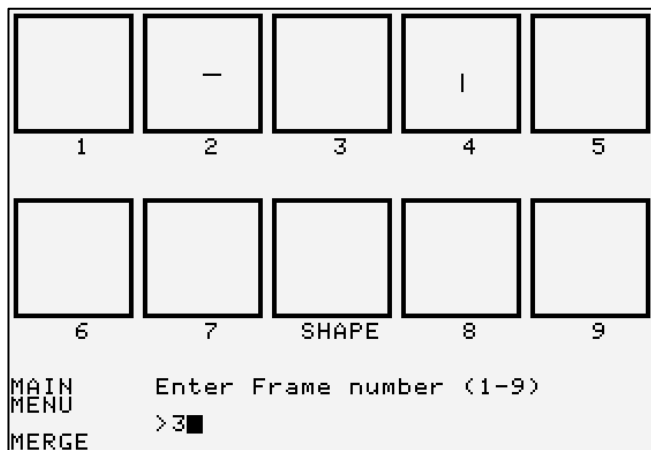


Figure 37. Enter Merge Frame Number

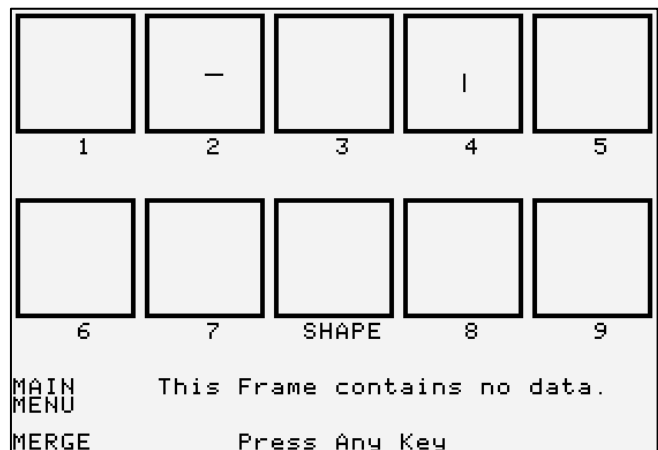


Figure 38. Merge Warning Message

Selecting Option 4 from the *SHAPE Manager* Main Menu allows the user to merge any of the shape definitions that are displayed in the Build windows and copy that shape definition into the Merge window

as shown in Figure 37. If a shape definition does not exist in the selected Build window, *SHAPE Manager* issues the warning message that is shown in Figure 38. If a shape definition does exist in the selected Build window as shown in Figure 39, *SHAPE Manager* will copy the SHAPE definition data that resides in the buffer for that selected Build window as a unique SHAPE definition into the SHAPE table data that resides in the Merge window data buffer as shown in Figure 40. If the Merge window data buffer is empty, *SHAPE Manager* copies in total the selected Build window SHAPE definition to the Merge window SHAPE table data buffer. If the Merge window SHAPE table data buffer is not empty, *SHAPE Manager* appends the selected Build window SHAPE definition data to the Merge window SHAPE table data, updates the number of shape definitions, and inserts a new index into the SHAPE table header that points to the appended data to the SHAPE table data according to the rules for SHAPE table data format. A second Merge is shown in Figure 41 and its completion message is shown in Figure 42. The data from any number of Build window SHAPE definitions may be appended to the SHAPE table that resides in the Merge window data buffer up to 255 SHAPE definitions which seems highly implausible. The more plausible limit would be the size of the Merge window data buffer which is initialized to be 1024 bytes. A buffer of this magnitude would be able to accommodate at least 2048 Plot and Move vectors which might very well address every single pixel location in a Merge window that is more than 45 pixels in width and more than 45 pixels in height, or $\sqrt{2048}$.

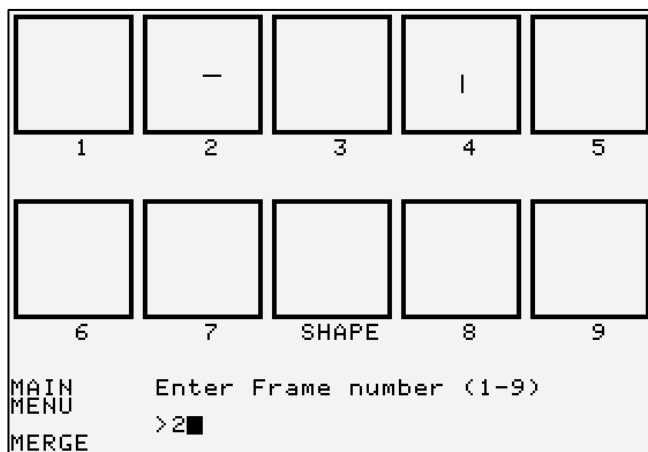


Figure 39. Enter Merge Frame Number

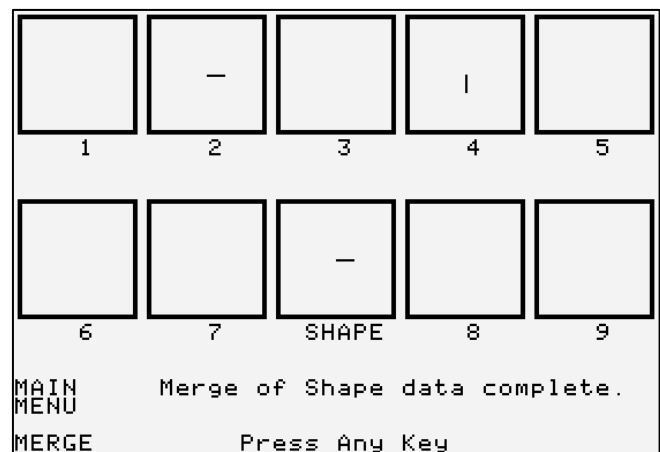


Figure 40. Merge Completion Message

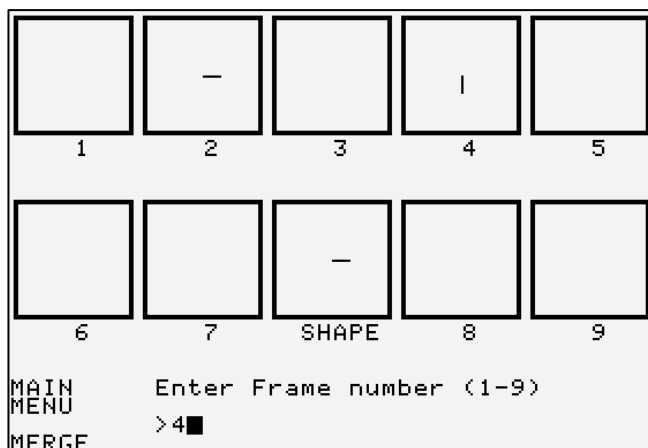


Figure 41. Enter Merge Frame Number

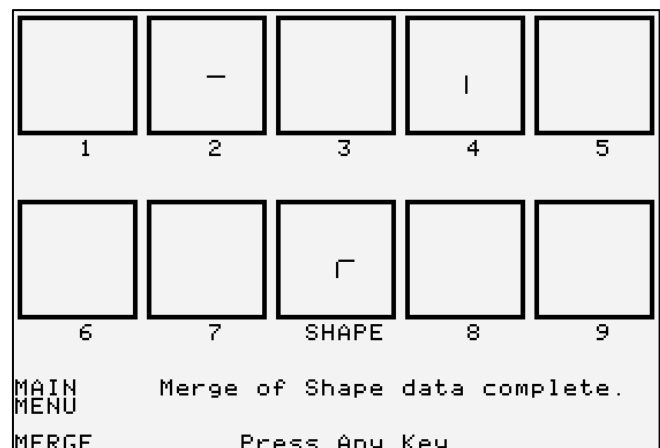


Figure 42. Merge Completion Message

Selecting Option 5 from the *SHAPE Manager* Main Menu allows the user to scale the shape definition that resides in the selected Build window as shown in Figure 43. If a shape definition does not exist in the selected Build window, *SHAPE Manager* issues the warning message that is shown in Figure 44. If a shape definition does exist in the selected Build window as shown in Figure 45, *SHAPE Manager* requests the value that DRAWSHP will use in order to scale the entire *SHAPE* definition that is shown in Figure 46. Using that scale value, the shape definition that is shown in Figure 46 is scaled, redrawn, and displayed in Figure 47 as well as issuing the completion message.

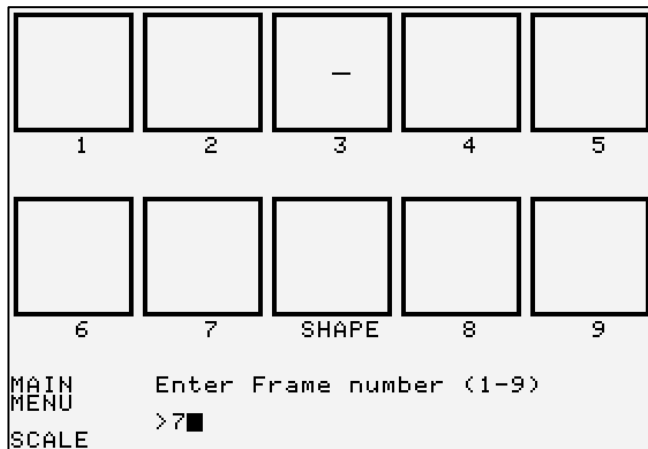


Figure 43. Enter Scale Frame Number

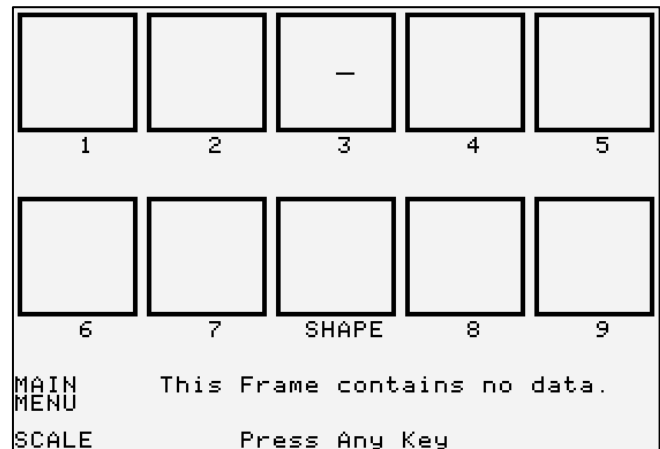


Figure 44. Scale Warning Message

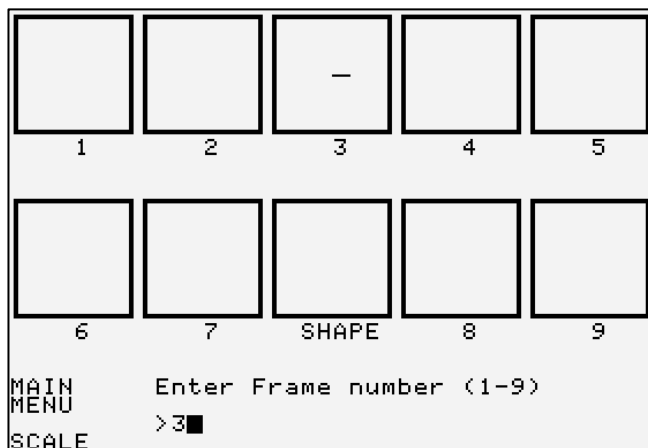


Figure 45. Enter Scale Frame Number

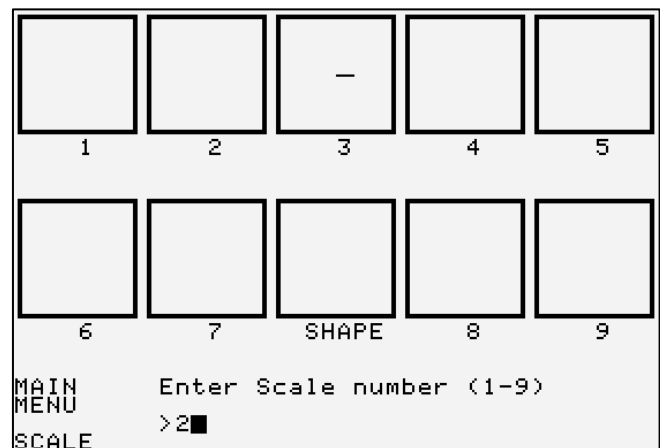


Figure 46. Enter Scale Scale Number

Selecting Option 6 from the *SHAPE Manager* Main Menu allows the user to rotate the shape definition that resides in the selected Build window as shown in Figure 48. If a shape definition does not exist in the selected Build window, *SHAPE Manager* issues the warning message that is shown in Figure 49. If a shape definition does exist in the selected Build window as shown in Figure 50, *SHAPE Manager* requests the value that DRAWSHP will utilize in order to rotate the entire shape definition that is shown in Figure 51.

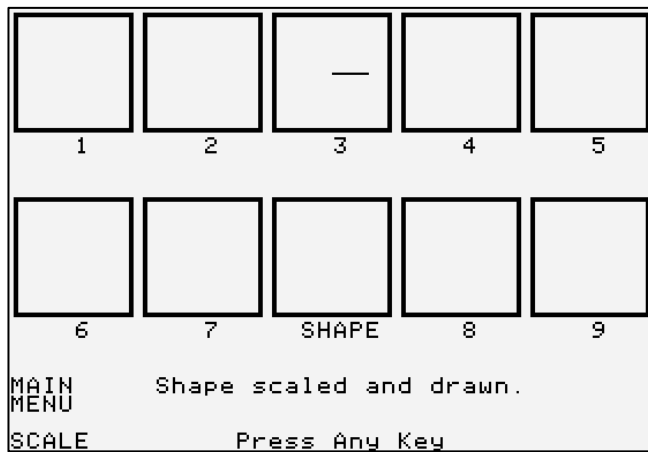


Figure 47. Scale Completion Message

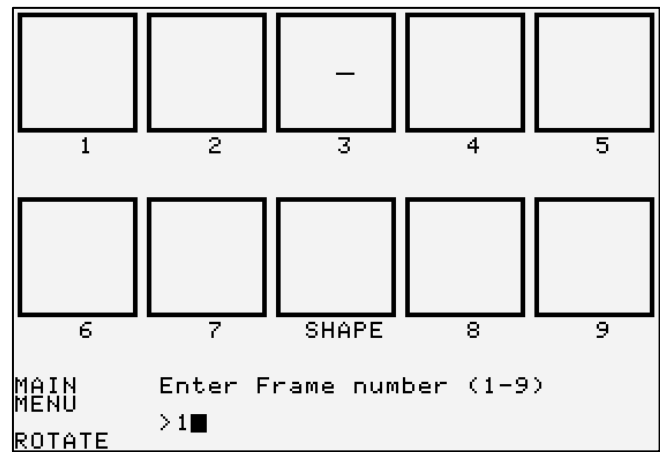


Figure 48. Enter Rotate Frame Number

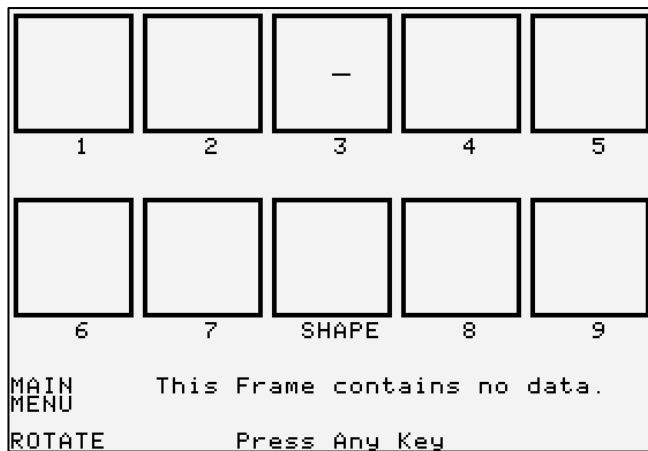


Figure 49. Rotate Warning Message

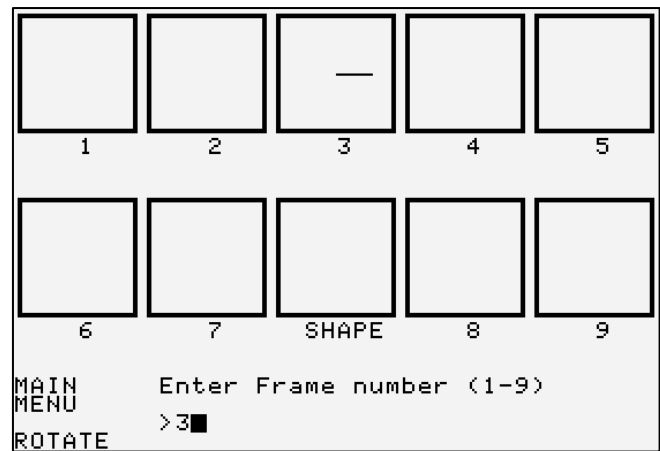


Figure 50. Enter Rotate Frame Number

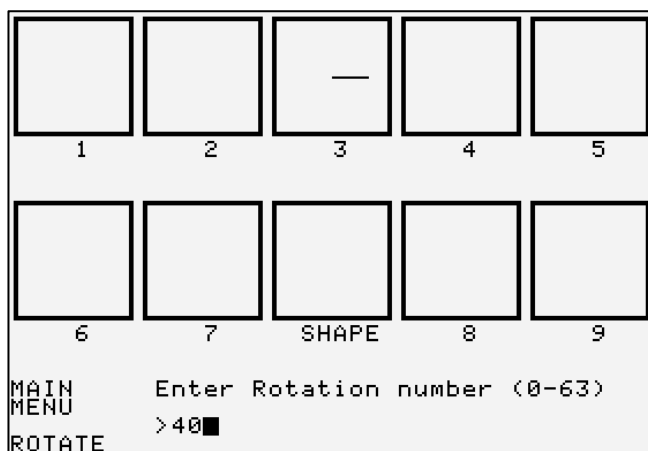


Figure 51. Enter Rotate Rotate Number

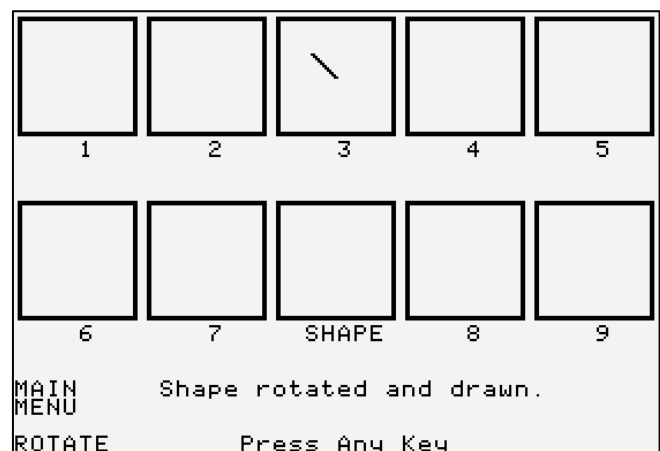


Figure 52. Rotate Completion Message

Using the rotate value that is shown in Figure 51, the shape definition is rotated, redrawn, and displayed in Figure 52 as well as issuing the completion message. There are sixteen possible values of rotation in every quadrant which is limited by the associated SCALE value and the size of the ROTATBL. Personally, I would have chosen fifteen or eighteen possible values to provide for five or six degrees of rotation per quadrant, respectively. However, the Applesoft design provides for a maximum of sixty-four possible values for rotation that is capable of rotating a shape definition within a full circle.

Selecting Option 7 from the *SHAPE Manager* Main Menu allows the user to load a SHAPE definition into the data buffer of a selected Build window from a DOS volume. However, if a ctrl-D is entered rather than a filename as shown in Figure 53 in order to display a volume catalog, *SHAPE Manager* requests a drive number as shown in Figure 54. The volume catalog is shown in Figure 55. Figure 56 shows the entry of a SHAPE table file name which may include its fully qualified pathname and Figure 57 shows the entry of the desired Build window number. The fully qualified pathname may contain a slot designation in order to change the default slot number. Finally, Figure 58 shows the entry of the color that *DRAWSHIP* will utilize for drawing the Plot vectors that are contained in the SHAPE definition data. The shape is drawn and displayed in Figure 59.

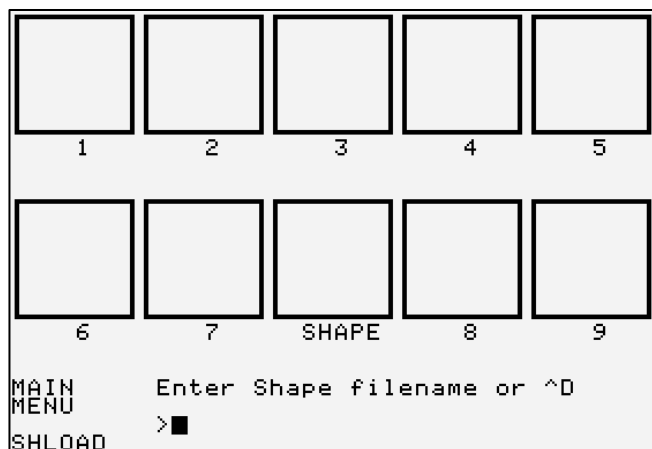


Figure 53. Enter SHLOAD ^D for Catalog

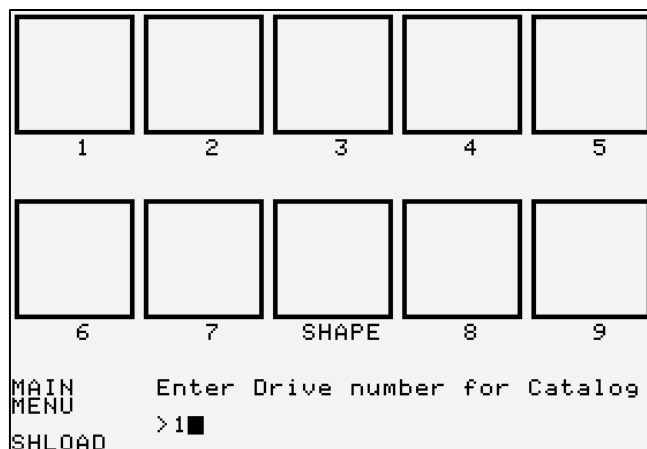


Figure 54. Enter SHLOAD Drive Number

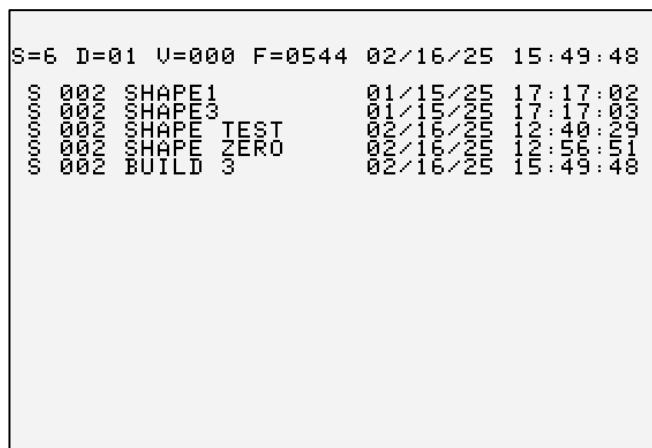


Figure 55. Display DOS Catalog

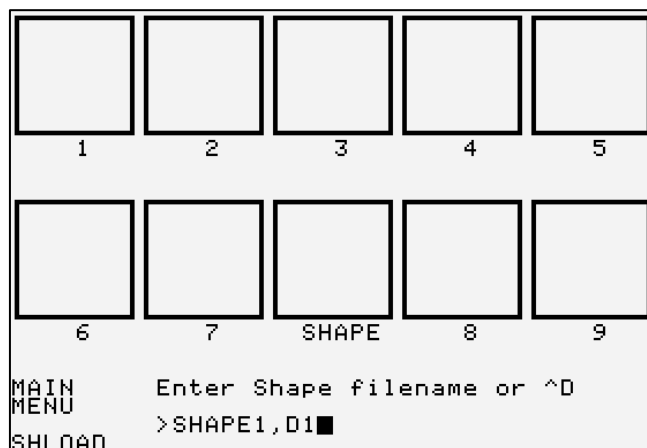


Figure 56. Enter SHLOAD File Name

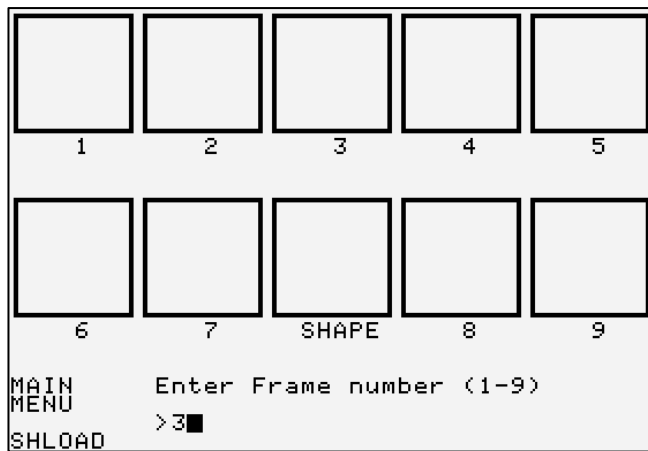


Figure 57. Enter SHLOAD Frame Number

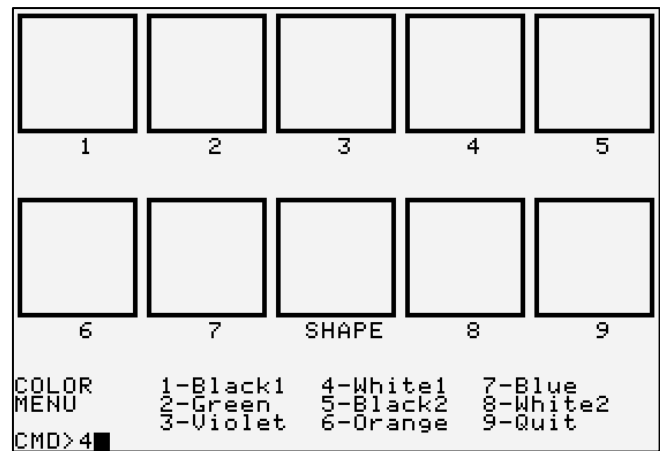


Figure 58. Enter SHLOAD Color Number

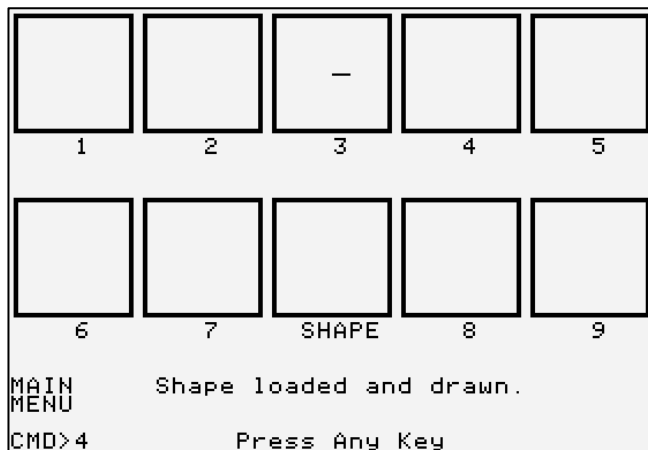


Figure 59. SHLOAD Completion Message

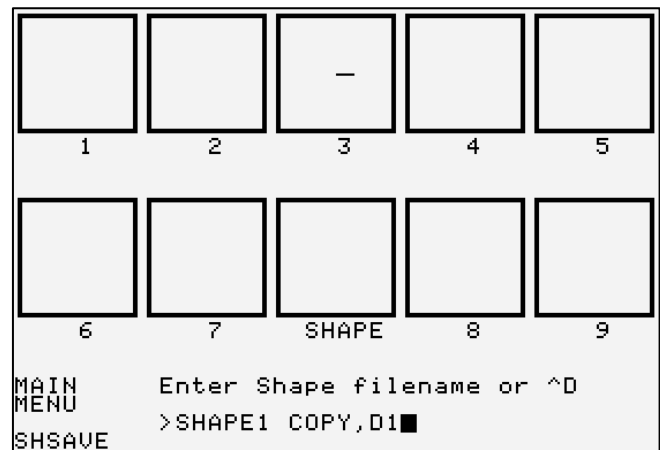


Figure 60. Enter SHSAVE File Name

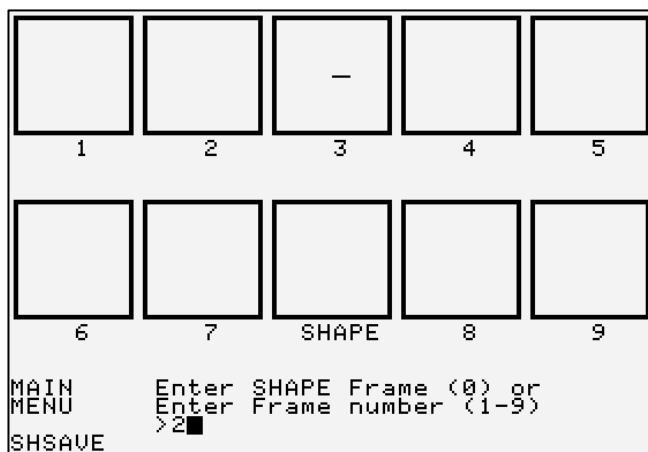


Figure 61. Enter SHSAVE Frame Number

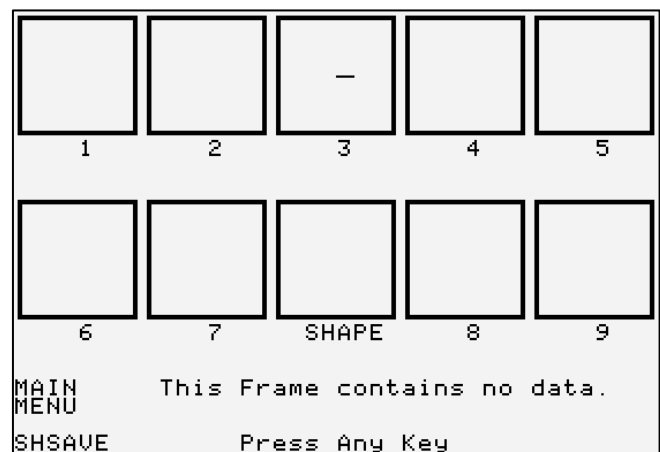


Figure 62. SHSAVE Warning Message

Selecting Option 8 from the *SHAPE Manager* Main Menu allows the user to save a SHAPE table onto a DOS volume from the data buffer of a selected Build window. As in the Load option, if a ctrl-D is entered rather than a filename, a volume catalog will be displayed for the selected drive number. Figure 60 shows the entry of a SHAPE table file name which may include its fully qualified pathname and Figure 61 shows the entry of the desired Build window. If the data buffer of the selected Build window contains no SHAPE definition data, *SHAPE Manager* issues the warning message that is shown in Figure 62. On the other hand, if the data buffer of the selected Build window does contain SHAPE definition data as shown in Figure 63, the selected SHAPE table data is saved to the selected DOS volume as shown in Figure 64.

The remaining four *SHAPE Manager* Main Menu options are selected by entering a specific letter for their option. These four options are L for List, P for Print, R for Refresh, and Q for Quit. The options List and Print are very similar in that List displays the content of the selected SHAPE definition data to the screen and Print sends the content of the selected SHAPE definition data to the printer. The option Refresh will refresh a Build or a Merge window whether its data buffer contains SHAPE definition data or not. The option Quit terminates all further processing and performs a DOS cold-start at 0x3D3.

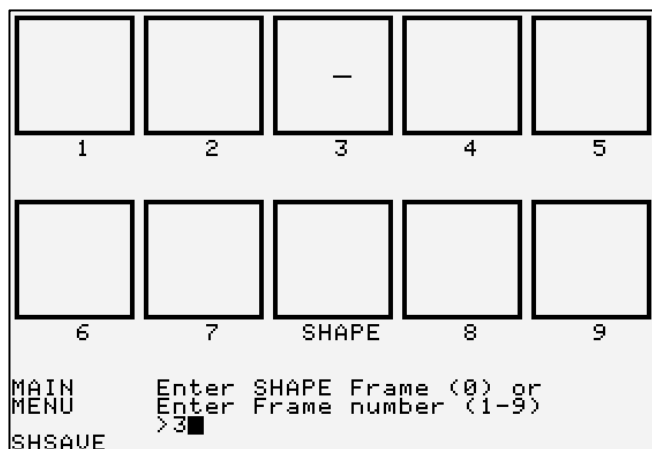


Figure 63. Enter SHSAVE Frame Number

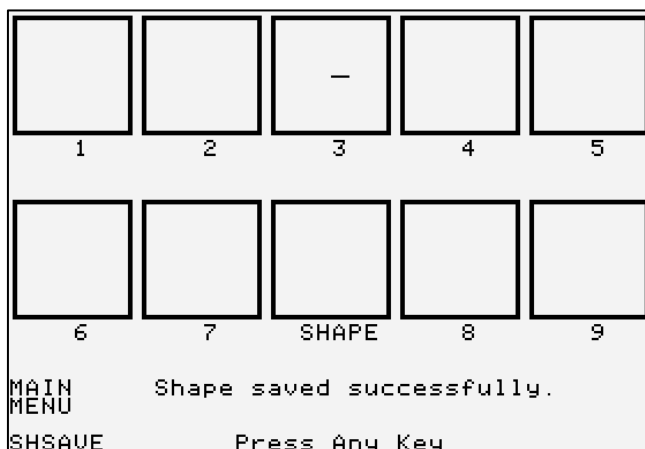


Figure 64. SHSAVE Completion Message

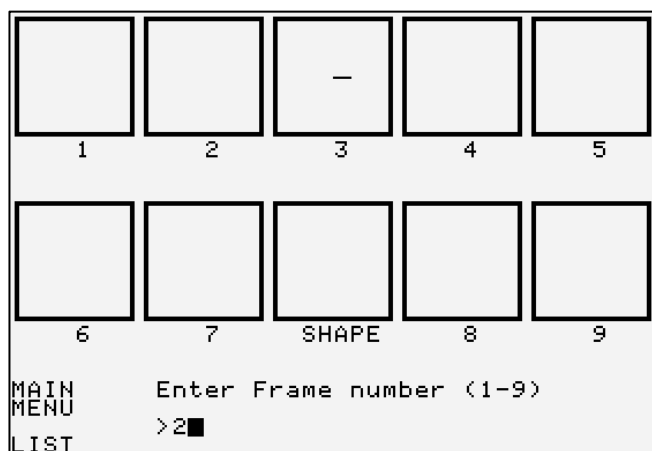


Figure 65. Enter List Frame Number

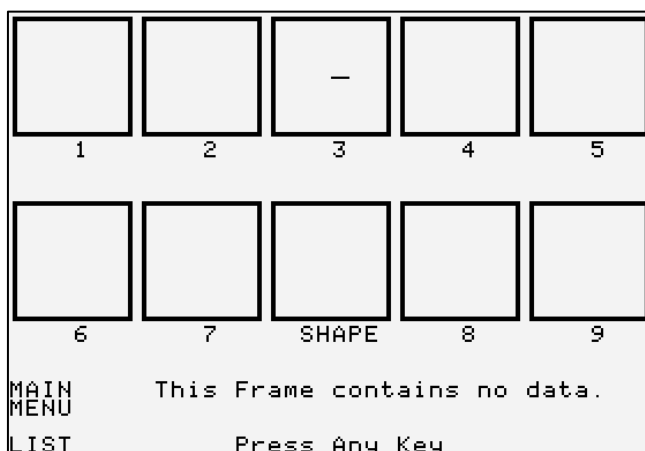


Figure 66. List Warning Message

Selecting Option L from the *SHAPE Manager* Main Menu allows the user to list the content of the selected SHAPE definition data to the screen. If the data buffer of the selected Build window that is shown in Figure 65 contains no SHAPE definition data, *SHAPE Manager* issues the warning message that is shown in Figure 66. The selected SHAPE definition data is formatted and the content of each byte of data is displayed on a single line showing, at most, all three vectors whether or not a pixel is drawn and the direction that its HIRES pixel cursor is moved as shown in Figure 67. After the format of all data bytes is displayed, the Build window specifications are listed showing the start coordinates, the color used to draw the shape, and the scale and rotational values that were used by DRAWSHP. Figure 68 shows the List completion message.

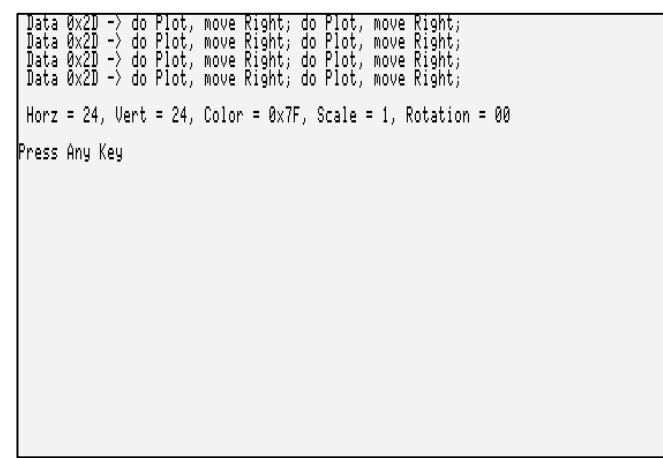


Figure 67. List Data Display

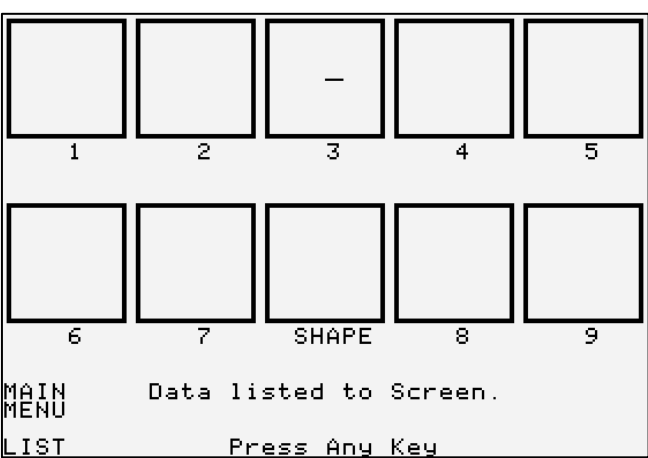


Figure 68. List Completion Message

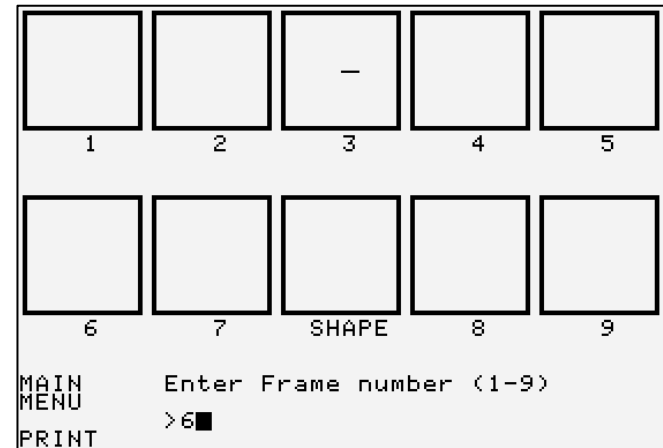


Figure 69. Enter Print Frame Number

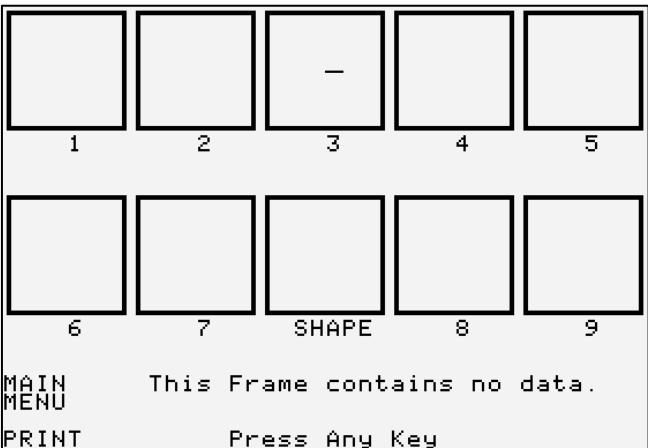


Figure 70. Print Warning Message

Selecting Option P from the *SHAPE Manager* Main Menu allows the user to list the content of the selected SHAPE definition data to the printer. If the data buffer of the selected Build window that is shown in Figure 69 contains no SHAPE definition data, *SHAPE Manager* issues the warning message that is shown in Figure 70. Otherwise, the SHAPE definition data that resides in the data buffer of the selected Build window as shown in Figure 71 is formatted as in the List option. That formatted data is sent to the printer whose slot interface card must reside in Slot #1 as shown in Figure 72. The printer slot number cannot be configured.

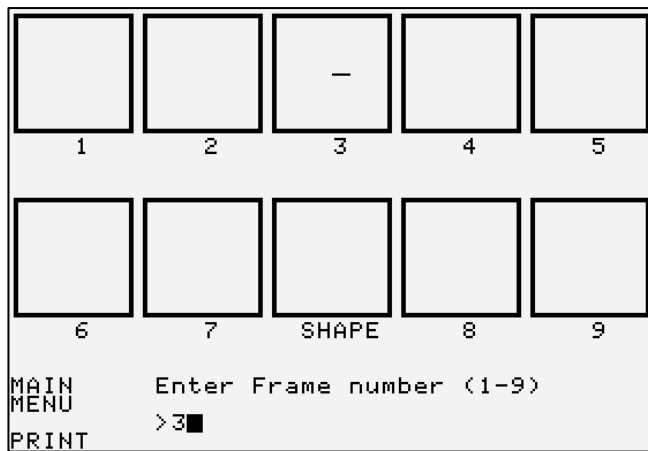


Figure 71. Enter Print Frame Number

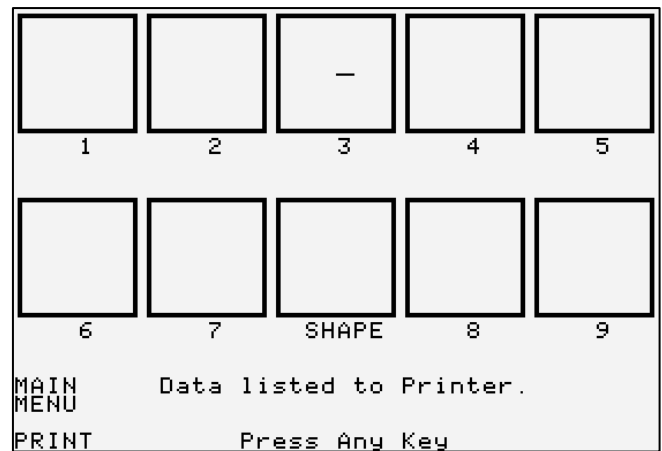


Figure 72. Print Completion Message

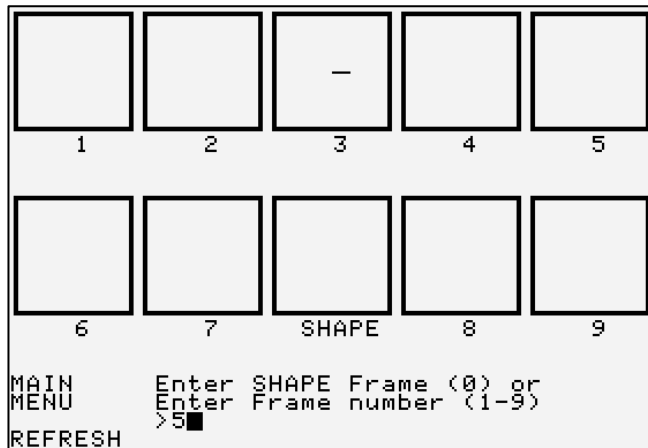


Figure 73. Enter Refresh Frame Number

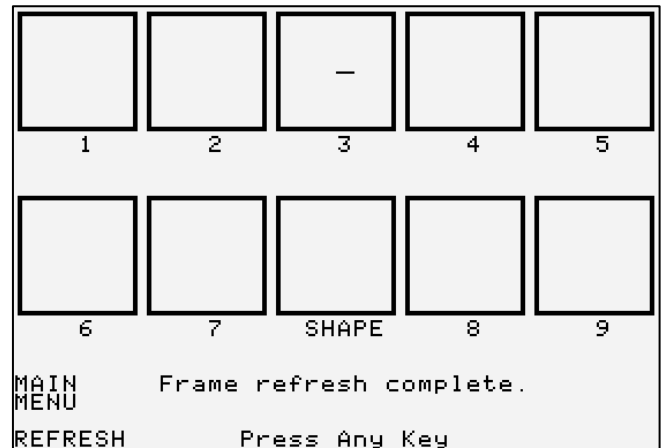


Figure 74. Refresh Completion Message

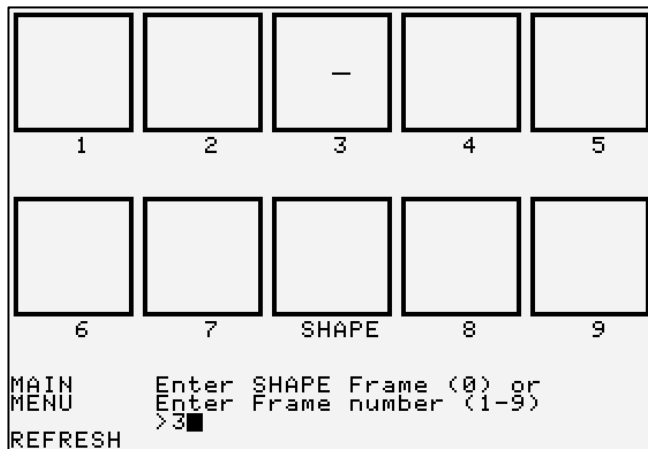


Figure 75. Enter Refresh Frame Number

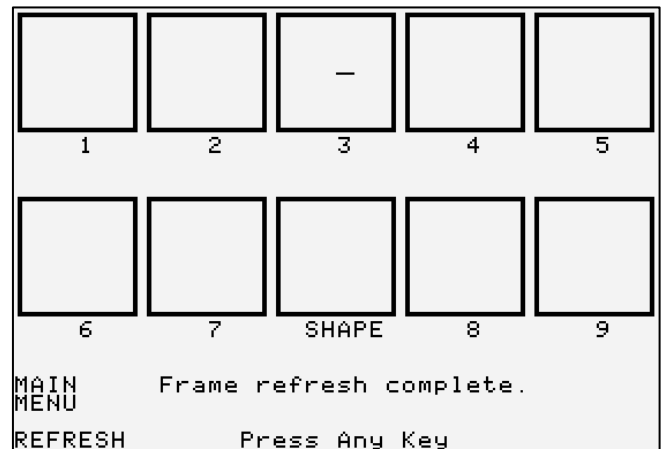
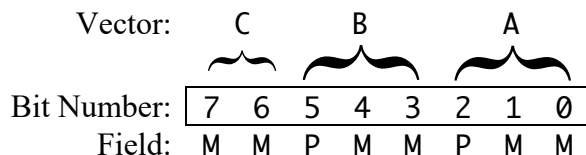


Figure 76. Refresh Complete Message

Selecting Option R from the *SHAPE Manager* Main Menu allows the user to refresh the selected Build or Merge window and the display of its data buffer content if that selected data buffer contains any SHAPE definition data. Figure 73 shows that either a Build window or a Merge window may be selected, the window frame and its content are erased, the window is redrawn, and the content of its data buffer is redrawn by DRAWSHP as shown in Figure 74. In these figures, Build window 5 contains no SHAPE definition data. Figures 75 and 76 show the refresh of Build window 3 which does contain SHAPE definition data.

Applesoft SHAPE Table Specifications

The Applesoft interpreter includes four statements which can be used to manipulate shapes by HIRES graphic routines: DRAW, XDRAW, ROT, and SCALE. Before these statements can be used in an Applesoft program, a shape must be defined, the shape definition must be stored in memory, and the memory location of the shape definition must be saved to HRSHTBL at 0xE8:E9. A shape definition consists of a sequence of vectors that comprise a series of data bytes. One or more shape definitions may be combined in order to build a SHAPE table. Each byte in a shape definition is divided into three vectors. Plot vector A uses bits 0:2, Plot vector B uses bits 3:5, and Move vector C uses bits 6:7. The last data byte in a shape definition is always zero. The DRAW and the XDRAW handlers use the DRAWSHP routine to step through each data byte of a shape definition, vector by vector, until it reaches the last data byte which is zero. Plot vectors A and B use their upper bit to draw a HIRES pixel if that bit is set or not to draw a HIRES pixel if that bit is not set and then use their lower two bits to define a move direction. Move vector C can only define a move direction other than UP. A data byte that is contained in a shape definition is arranged as follows:



Each bit pair MM field specifies the direction to move, and each bit P field specifies whether or not to draw a HIRES pixel before moving. The following defines the value for the MM field and the resulting direction of movement, and the value for the P field that results in whether or not to draw a HIRES pixel.

If MM = 00 move up = 01 move right = 10 move down = 11 move left	If P = 0 do not plot pixel = 1 do plot pixel
---	---

Plot vectors A and B both contain a plot field and a move field. Move vector C only contains a move field; it does not contain a plot field. Move vector C can only specify a move without drawing a HIRES pixel.

The DRAWSHP routine is used by the DRAW and the XDRAW handlers in order to process the shape vectors from right to left, that is, from least significant bit to most significant bit, beginning with vector A, then with vector B, and finally with vector C. After a vector is processed, *if the remaining vectors of the data byte are zero, the remaining vectors are ignored*. Thus, the data byte cannot contain a Move UP in vector C or %00 because that vector would be ignored after processing the previous vector. Similarly, if vector C is %00, then vector B cannot contain a No Plot UP or %000 as that vector would also be ignored. Lastly, a No Plot UP or %000 in section A will end the shape definition unless there is any bit set in vector B. These Plot vector, Move vector, move direction, and termination rules all determine what a data byte may contain and

what a data byte may not contain in order to process all of the shape vectors in a shape definition that are consistent with these rules.

Plot	Direction	Move Vector Value	Plot Vector Value
No	UP	%00	%000
No	RIGHT	%01	%001
No	DOWN	%10	%010
No	LEFT	%11	%011
Yes	UP	n/a	%100
Yes	RIGHT	n/a	%101
Yes	DOWN	n/a	%110
Yes	LEFT	n/a	%111

Table 7. Move and Plot Vector Values

It is easier to manually build a SHAPE table for a shape definition once that shape definition is drawn on graph paper. Simply put a circle in each square of the graph paper that represents the shape that is drawn using HIRES pixels. Then select the first HIRES pixel in where to begin drawing the shape. Draw a path through each circle on the graph paper using only a 90 degree angle for each turn. Next, re-draw the shape as a series of Plot vectors and Move vectors where each vector moves one square up, down, right, or left. The Plot vectors may or may not pass through a circle before they move, and the Move vectors never pass through a circle before they move. So, a circle on the graph paper requires a Plot vector that draws a pixel. Once all of the Plot vectors and the Move vectors are identified, transform their plot/move symbol into a bit value according to Table 7 and create a list of Move vector and Plot vector values.

From the list of Move vector and Plot vector values, create an 8-bit data byte by inserting each of the vector values into their proper field as a vector A first, then as a vector B, and then possibly as a vector C. Always be mindful of the rules that govern a Move UP %00 vector and a No Plot UP %000 vector from above. A No Plot UP for vector A cannot stand alone without a Plot or a Move vector for vector B. If it is not possible to create a data byte that is not zero, chose another location to start drawing the shape and create a new list of Move vector and Plot vector values. All of the data bytes that are derived from all of the Move vector and the Plot vector values become the shape definition. The SHAPE table is comprised of the shape header and the shape definition. Figure 77 shows the definition and the complete layout of a SHAPE table.

Once all of the data bytes are derived from the Move vector and the Plot vector values in order to create a shape definition having a termination byte of 0x00, preface the shape definition with a header having just four bytes: 0x01, 0x00, 0x04, and 0x00. This shape header defines one shape definition that resides at four bytes from the start of the shape header. These header bytes along with the shape definition data bytes that are terminated with 0x00 comprise a complete SHAPE table. This SHAPE table may now be entered into memory at some starting address. When all of the SHAPE table data bytes are in memory, the SHAPE table can be saved to a disk volume using the DOS SHSAVE command if DOS 4.5.08H currently resides in the computer. Whatever address you provide to the DOS SHLOAD command, DOS will automatically initialize the HRSHTBL page-zero variable at 0xE8:E9 with the address that DOS just loaded the SHAPE table. When the B keyword is used with the DOS SHLOAD command, FRETOP at 0x6F:70 and HIMEM at 0x73:74 will be

set to that same load address in order to easily protect the SHAPE table from changes to the Character String Pool. Applesoft in the modified ROM no longer provides the resources to load a SHAPE table into memory using a cassette tape recorder.

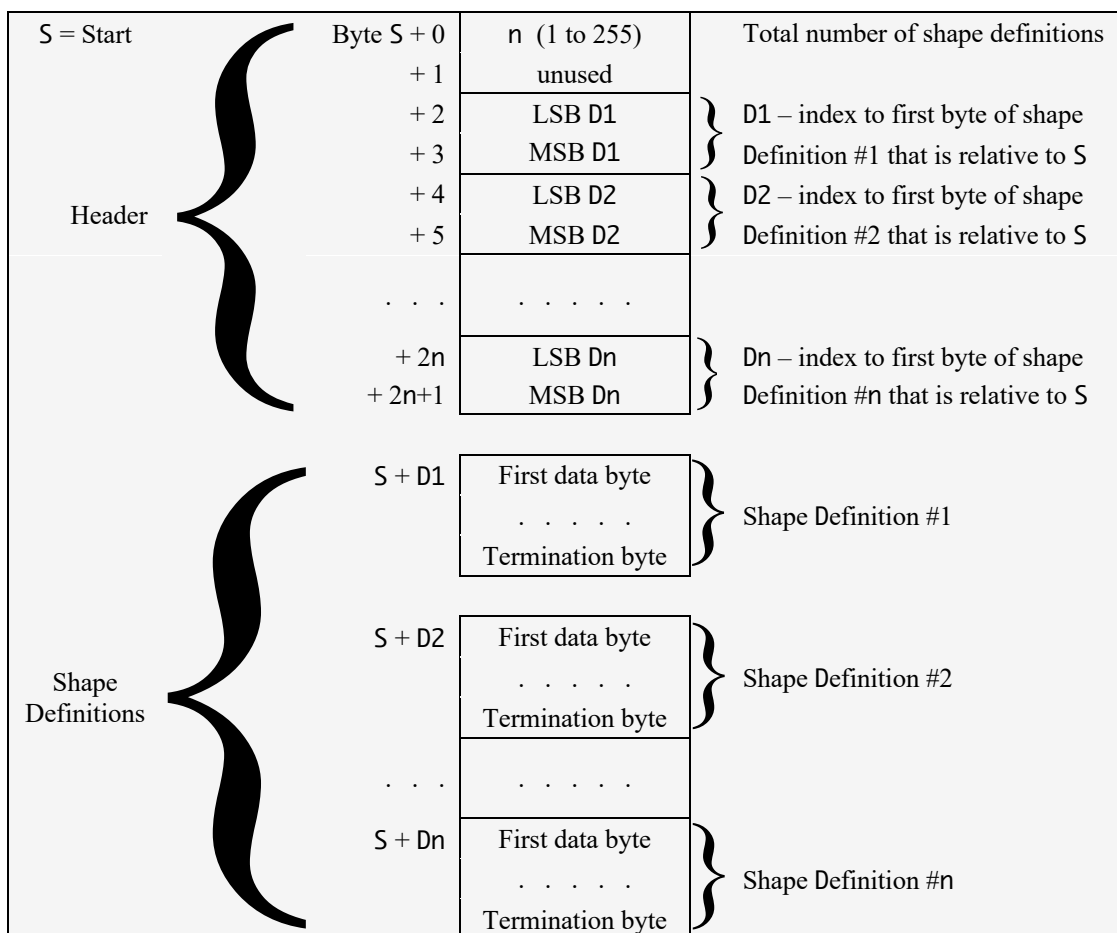


Figure 77. SHAPE Table Definition Layout

Using an Applesoft SHAPE Table with Applesoft Statements

The Applesoft Spoke program that is shown in Figure 9 is an excellent example in using an Applesoft SHAPE table. This program is also an excellent example in using the Applesoft statements HCOLOR, SCALE, and ROT. This Applesoft program first clears the Text screen and it defines the DOS character ^D, that is, control-D. It uses that DOS character along with the DOS SHLOAD command in order to place the SHAPE table that resides in the SPOKE SHAPE file at memory address $0xB000$. The DOS SHLOAD command automatically copies the address $0xB000$ to the HRSHPTBL page-zero variable at $0xE8:E9$. Because the B keyword is included with the DOS SHLOAD command, FRETOP at $0x6F:70$ and HIMEM at $0x73:74$ are also initialized to $0xB000$ in order to protect the SHAPE table from further changes to the Character String Pool. The Applesoft HGR statement clears HIRES memory from $0x2000$ to $0x3FFF$ and the bottom four TEXT lines are hidden by referencing the $0xC052$ address. The Applesoft HCOLOR statement sets HRCOLOR at $0xE4$ to 3 which selects WHITE1 or $0x7F$ for COLOR at $0x30$, and the Applesoft SCALE statement sets HRSCALE at $0xE7$ to 11 in order to draw the maximum sized shape possible that still fits within the HIRES

screen area. The Applesoft HPLLOT command in line #80 draws a very nice window frame around the outer perimeter of the HIRES screen. The Applesoft Spoke program utilizes the Applesoft FOR statement in order to draw sixty-four copies of the same shape definition from the same starting location in pixels by initializing the Applesoft ROT statement that sets HRRROT at 0xF9 to values from 0 to 63. After the conclusion of the FOR:NEXT loop, Applesoft line #140 simply waits forever for any keypress, and when a keypress is detected, the key buffer is cleared by referencing the 0xC010 address in Applesoft line #150. The program exits HIRES mode, returns to TEXT mode, and ends.

DRAW Statement

```
DRAW      n AT Xn, Yn
          n
```

```
Example:  DRAW 1 AT 140, 96
          DRAW 2 AT 100, 2
          DRAW 3
```

The Applesoft DRAW statement draws the specified shape definition number *n* by this HIRES graphic routine at the given horizontal and vertical coordinates *Xn* and *Yn*, respectively. If only the specified shape definition number *n* is given, the last coordinates that were calculated by the most recently executed HPLLOT, DRAW, or XDRAW statement are used for *Xn* and *Yn*. The variable *n* specifies the shape definition number that must exist within the current SHAPE table. The current SHAPE table must already reside in memory whose address is saved to HRSHTPTBL at 0xE8:E9. The specified shape definition *n* is drawn at the horizontal coordinate *Xn* and at the vertical coordinate *Yn*. The variable *n* must be in the range of 1 to 255 and *n* must exist within the value that is found in the first byte of the SHAPE table. The first byte of the SHAPE table contains the total number of shape definitions that exist in that particular SHAPE table. The variable *Xn* must be in the range of 0 to 279 and the variable *Yn* must be in the range of 0 to 191. If the specified values for these variables are not within these given ranges, an error message is displayed and Applesoft processing terminates unless an error handler has been previously specified. The Applesoft statements HRCOLOR, SCALE, and ROT must be previously processed in order for the DRAW statement to utilize their values. The values of HRCOLOR at 0xE4, HRSSCALE at 0xE7, and HRRROT at 0xF9 are never initialized and their initial values are indeterminate. If a SHAPE table does not exist in memory or HRSHTPTBL contains an indeterminate address, unpredictable results should be expected that may profoundly alter the contents of memory.

DRAW and XDRAW have no relationship or interdependencies and these two Applesoft statements are **not** designed to be used in conjunction with the other. DRAW 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 HIRES pixels that are currently being displayed. DRAW does not incorporate any of the old HIRES pixel information with any of the pixel information data of the new SHAPE definition.

The Applesoft DRAW statement draws colors to the HIRES screen such that the data that is drawn replaces whatever data may previously exist on the HIRES screen. The Applesoft DRAW statement may be used from both the Apple Command Line and from within an Applesoft program.

XDRAW Statement

XDRAW n AT Xn, Yn

Example: XDRAW 1 AT 140, 96
 XDRAW 2 AT 100, 2

The Applesoft XDRAW statement is nearly the same as the Applesoft DRAW statement except that the horizontal and the vertical coordinates Xn and Yn must always be given with the Applesoft XDRAW statement along with a shape definition number n.

XDRAW 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 coordinates. As with all graphic routines that make use of the exclusive-OR microprocessor instruction, color complements must be taken into consideration when using the Applesoft XDRAW statement.

The Applesoft XDRAW statement 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. As previously explained, Table 6 provides useful information in selecting the shape colors and the background colors in order to obtain the desired colored results for the shape. The main purpose in using the Applesoft XDRAW statement 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 Applesoft XDRAW statement may be used from both the Apple Command Line and from within an Applesoft program.

SCALE Statement

SCALE n

Example: SCALE 1
 SCALE 8

The Applesoft SCALE statement sets the size for a shape to be drawn by the Applesoft DRAW or XDRAW statements. The change in scale is specified by n which can be any value from 1 to 255. A scale of 0 is interpreted to be a scale of 256. The routine DRAWSHP utilizes the value n that is found in HRSCALE at 0xE7 by extending each Plot or Move vector n times. SCALE is not initialized by the Applesoft HGR, HGR2, or RUN statements. Until the first Applesoft SCALE statement is processed, the scale value for any HIRES graphic routine is indeterminate.

The Applesoft SCALE statement may be used from both the Apple Command Line and from within an Applesoft program.

ROT Statement

ROT n

Example: ROT 1
 ROT 8

The Applesoft ROT statement sets the angular rotation for a shape to be drawn by the Applesoft DRAW or XDRAW statements. The change in rotation is specified by n which can be any value from 0 to 255. A rotation of 0 will not change the orientation of the shape on the HIRES screen. The routine DRAWSHP processes the value n that is found in HRRROT at 0xF9 using modulo 64 such that the utilization of n repeats every instance of 64. An Applesoft ROT = 16 statement orients the shape 90 degrees clockwise, ROT = 32 orients the shape 180 degrees clockwise, and ROT = 48 orients the shape 270 degrees clockwise. For SCALE = 1, only four rotation values are recognized, that is, 0, 16, 32, and 48. For SCALE = 2, eight rotations are recognized, and so forth. ROT is not initialized by the Applesoft HGR, HGR2, or RUN statements. Until the first Applesoft ROT statement is processed, the angular rotation value for any HIRES graphic routine is indeterminate.

The Applesoft ROT statement may be used from both the Apple Command Line and from within an Applesoft program.

HCOLOR Statement

HCOLOR n

Example: HCOLOR 1
 HCOLOR 6

The Applesoft HCOLOR statement sets the HIRES graphic routine color to the specified HCOLOR value n. The HCOLOR value n must be in the range of 0 to 7 and that value resides in HRCOLOR at 0xE4. HRCOLOR is used as an index into the BITABLE table in order to select the desired value for COLOR at 0x30. HRCOLOR color values and their associated color names are shown in Table 8. The set of colors that have their MSB clear cannot be mixed with the set of colors that have their MSB set within modulo 2 bytes. That is, an even numbered byte and its associated odd numbered byte can support a set of colors that have their MSB clear and the next even byte along with its associated odd numbered byte can support a set of colors that have their MSB set. Furthermore, colors of one set may appear in the same bytes on lines above and lines below the other set of colors. HCOLOR is not initialized by the Applesoft HGR, HGR2, or RUN statements. Until the first Applesoft HCOLOR statement is processed, the plotting color for any HIRES graphic routine is indeterminate.

In advanced graphic routines and within a modulo 2 bytes, colors of the same set may be mixed such as Green, White1, Green, White1 or Blue, Black2, Blue, Black2. Thus, interlaced colors and dithered colors may produce useful background textures for a variety of intermediate colors and layered colors.

The Applesoft HCOLOR statement may be used from both the Apple Command Line and from within an Applesoft program.

HCOLOR	HRCOLOR	Name
0	0x00	BLACK1
1	0x2A	GREEN
2	0x55	PURPLE
3	0x7F	WHITE1
4	0x80	BLACK2
5	0xAA	ORANGE
6	0xD5	BLUE
7	0xFF	WHITE2

Table 8. HRCOLOR Values and Their Color Names

New SHAPE Table Commands in DOS 4.5 Build 08

A great deal of effort went into a project to modify DOS 3.3 that would load a Binary file directly into memory when I first began working at Sierra On-Line late in 1983. This effort produced a modified DOS BLOAD command that utilized additional keywords that would provide the necessary parameters in order to achieve its accelerated processing rate. Unfortunately, I have no further information on the additional keywords that were utilized and the extent of the modifications that went into DOS 3.3, the DOS BLOAD command, and 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 Build 06 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, and 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 since those four bytes are missing. The memory load address and the length in bytes for a *Special* Binary file must be already 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 ROM routines that specifically manage SHAPE tables, I have reconsidered the usefulness of both 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 to replace the excised Applesoft SHLOAD statement as well as developing a companion DOS SHSAVE command.

The Binary File commands in the DOS 4.5 Build 08 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 9. 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.

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

Table 9. Binary File Commands in DOS 4.5.08H

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 Build 08. 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 HRSHTPTBL at 0xE8:E9 in the same way as it is copied by the handler for the Applesoft SHLOAD statement. The 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.

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,\$B000,\$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 Build 08. 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. In DOS 4.5 these keywords are **optional**, but if they are included, **both** 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.

Modified Applesoft FILE Commands in DOS 4.5 Build 08

The Applesoft File commands in the DOS 4.5 Build 08 command repertoire consist of those commands that manage the general operation of Applesoft files. The DOS CHAIN command loads an Applesoft file into memory and preserves the variables of the previous Applesoft program. The DOS LOAD command loads an Applesoft file into memory. The DOS RUN command loads an Applesoft file into memory before

it begins processing the Applesoft statements that now reside in memory. The DOS SAVE command saves the Applesoft program that currently resides in memory to a file in a disk volume.

The syntax of the Applesoft File commands is shown in Table 10. The Applesoft 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. However, the DOS CHAIN command is not permitted to be used on the Apple Command Line.

Command	Command Syntax
CHAIN	f [,Ss][,Dd][,Vv][,Aa][,Ll][,R]
LOAD	f [,Ss][,Dd][,Vv][,Aa][,R]
RUN	f [,Ss][,Dd][,Vv][,Aa][,Ll]
SAVE	f [,Ss][,Dd][,Vv][,B][,R[1]]

Table 10. Applesoft File Commands in DOS 4.5.08H

CHAIN Command

CHAIN f [,Ss][,Dd][,Vv][,Aa][,Ll][,R]

Example: CHAIN CHAINPART2,D2
 CHAIN CHAINPART2,A\$1000
 CHAIN CHAINPART2,L10

This command is not available in DOS 3.3 for Applesoft File commands and this command was initially developed for DOS 4.1. This command was enhanced for DOS 4.5.08H to accept the A keyword. The DOS CHAIN command can only be used from within an Applesoft program or by an assembly language routine. This Applesoft File command reads into memory at 0x0801 or at the specified address a if the A keyword is given, the Applesoft program that is contained in the Applesoft file f in the specified volume. The byte length of the Applesoft program is found in the first two bytes of the first data sector of the Applesoft file f. Applesoft files are file Type 0x02. DOS begins processing the Applesoft file f in a unique way such that the DOS CHAIN handler does not clear the value(s) of any previous Applesoft program variable(s). Therefore, the Applesoft program can utilize the numerical and the string data results from the previous Applesoft program(s) and the current Applesoft program can provide its numerical and its string data results to any following Applesoft program(s) that may be sequentially processed by the DOS CHAIN command. If the A keyword is included with the DOS CHAIN command, the DOS CHAIN handler copies the specified address a to PRGTAB at 0x67:68. Applesoft initializes PRGTAB during its COLDSTRT initialization to the initial value of 0x0801. The address in PRGTAB may be changed by the DOS CHAIN command for an Applesoft program that has already been initialized to that same address. Refer to the DOS LOAD command for further details on *Applesoft program address initialization*. Applesoft does not provide any means to re-initialize PRGTAB to the COLDSTRT value of 0x0801. The address in PRGTAB remains constant unless it is changed by another DOS Applesoft command.

If the L keyword is included with the DOS CHAIN command, Applesoft processing will begin at that Applesoft program line number 1 only if that line number 1 exists, otherwise the Applesoft interpreter will report an error and terminate further Applesoft processing. Obviously, this capability opens up a multitude of selective programming functionality that could be based on its program processing for selective entry program line numbers for 1.

If the R keyword is **not** used with the DOS CHAIN command, the DOS CHAIN handler will automatically call the Applesoft GARBAG routine at memory address 0xE484 before the DOS CHAIN handler moves the Simple Variable and the Array Variable descriptors to their new memory location at the end of the Applesoft file f. Using the R keyword will bypass that call to the Applesoft GARBAG routine and it allows the user to utilize another method, process, or strategy in order to collect and remove character string data garbage before or after using the DOS CHAIN command. It is critical that the Applesoft program that invokes the DOS CHAIN command locate or *cause* to move all of its simple character string variables and character string array variables to the Character String Pool that are intended to be utilized by the chained Applesoft program. All character string data that safely resides in the Character String Pool will be available to the current Applesoft program or to another Applesoft program when the DOS CHAIN command is invoked. A more in-depth discussion of the DOS CHAIN command can be found in Section I.15 of *DOS 4.5 Volume and File Disk Management System Second Edition*. The DOS CHAIN handler concludes its processing and enters 0xD955 in the Applesoft ROM by means of the ASROMSET variable. This Applesoft ROM location initializes the start address of the Applesoft program if the L keyword is included with the DOS CHAIN command. This particular Applesoft processing utilizes the Applesoft FNDLIN2 routine at 0xD61E in order to locate the Applesoft program line number 1 that the DOS CHAIN handler saves to LINNUM at 0x50:51. Finally, the DOS CHAIN handler enters the Applesoft NEWSTT routine at 0xD7D2 by means of the DOS ASROMNEW variable. The Applesoft NEWSTT routine begins to process each Applesoft statement in the Applesoft program that currently resides in memory.

```

]load START
]list
10 D$ = CHR$ (4):AB = 123:CD% =
  456:EF$ = "Test Chain" + " "
20 PRINT : PRINT "This is the S
  TART program to test CHAIN":
  PRINT
30 PRINT "AB = ";AB;" CD% = ";
  CD%:" EF$ = ";EF$: PRINT
40 PRINT D$;"CHAIN PROGRAM2"
]load PROGRAM2
]list
10 PRINT : PRINT "Now running p
  rogram PROGRAM2": PRINT
20 PRINT "AB = ";AB;" CD% = ";
  CD%:" EF$ = ";EF$: PRINT
30 PRINT D$;"CATALOG": PRINT
]

```

Figure 78. CHAIN Program Listings

```

]run START
This is the START program to test CHAIN
AB = 123, CD% = 456, EF$ = Test Chain

Now running program PROGRAM2
AB = 123, CD% = 456, EF$ = Test Chain

S=6 D=02 V=000 F=0550 02/14/24 08:28:48
A 002 START 02/14/24 08:28:48
A 002 PROGRAM2 02/14/24 08:28:48
]

```

Figure 79. CHAIN Program Outputs

Figure 78 shows the program list of two Applesoft programs named START and PROGRAM2. The Applesoft START program defines four simple variables which are D\$, AB, CD%, and EF\$. The character string variable EF\$ is defined in such a way as to *cause* the Applesoft interpreter to relocate that string variable into the

Character String Pool where that variable can be safely stored and utilized by a chained Applesoft program. The Applesoft interpreter also moves the variable D\$ to the Character String Pool before that variable is utilized with the DOS CHAIN command. All four variables will be available to Applesoft PROGRAM2 program when the DOS CHAIN command is issued as shown in line 40 in Figure 78. Figure 79 shows the text output of the Applesoft START program after the program is RUN. The Applesoft PROGRAM2 program clearly shows the values of the four simple variables from the Applesoft START program, and not only have these variable names been absolutely preserved but also their respective values have been absolutely preserved. Even the value for the variable D\$ has been absolutely preserved, otherwise the DOS CATALOG command in the Applesoft PROGRAM2 program would totally fail.

A character string descriptor contains only the first two characters of a character string name, so care must be given in naming all Applesoft variables. The memory address that is in a character string descriptor or in a character string element is initially the memory location where the character string data resides within the contents of its Applesoft program. Once the DOS CHAIN command has replaced the current Applesoft program in memory with the next Applesoft program that is specified by file f, the resident program character string data will be overwritten by file f and lost, and its address or its location in memory will become invalid. Therefore, caution must be exercised when using character string variables whose memory address is still within the current Applesoft program. When those character string variables are **not** caused to be moved, that is, when they are **not** copied from within the contents of an Applesoft program to the Character String Pool and safely stored in that memory location, those character string variables will **never** be available for general access by the next or other chained Applesoft program(s).

LOAD Command

LOAD f [,Ss][,Dd][,Vv][,Aa][,R]

Example LOAD HELLO
 LOAD HELLO,A\$1000
 LOAD HELLO,R

This command is available in DOS 3.3 for Applesoft File commands and this command was enhanced for DOS 4.1 to accept the R keyword. This command was further enhanced for DOS 4.5.08H to accept the A keyword. This Applesoft File command reads into memory at 0x0801 or at the specified address a if the A keyword is given, the Applesoft program that is contained in the Applesoft file f in the specified volume. The byte length of the Applesoft program is found in the first two bytes of the first data sector of the Applesoft file f. Applesoft files are file Type 0x02. The DOS LOAD command will also process A Type files, or file Type 0x20, as an Applesoft program in the same way those files Types are processed in DOS 3.3.

If the A keyword is included with the DOS LOAD command, the DOS LOAD handler copies the specified address a to PRGTAB at 0x67:68. Applesoft initializes PRGTAB during its COLDSTRT initialization to the initial value of 0x0801. The address in PRGTAB may be changed by the DOS LOAD command for an Applesoft program that has already been initialized to that same address. Applesoft does not provide any means to re-initialize PRGTAB to the COLDSTRT value of 0x0801. The address in PRGTAB remains constant unless it is changed by another DOS Applesoft command. *Applesoft program address initialization is*

automatically performed on an Applesoft program after it has been loaded into memory using the DOS LOAD command. The DOS LOAD handler concludes its processing and enters the Applesoft ASENTER routine at 0xD4F2 by means of the DOS RESETADR address at 0xBEE8. The Applesoft ASENTER routine begins the Applesoft interpreter and it first loads the address that resides in PRGTAB, decrements that address, and copies that decremented address to DATPTR at 0x7D:7E and to TXTPTR at 0xB8:B9, it copies HIMEM at 0x73:74 to FRETOP at 0x6F:70, and it copies VARTAB at 0x69:6A to ARYTAB at 0x6B:6C and to STREND at 0x6D:6E. The Applesoft ASENTER routine also forces the stack pointer to 0xF8. After the routine has performed this initialization, the Applesoft ASENTER routine copies the address in PRGTAB to INDEX at 0x5E:5F and it uses INDEX to re-initialize the memory address to the next Applesoft line number throughout the entire Applesoft program. Every time the DOS LOAD command is used to read an Applesoft program into memory, that Applesoft program is processed from start to end in order to ensure that all of the addresses that point to their next Applesoft line number conform to the memory address that currently resides in PRGTAB. If a user wishes to select another address other than 0x0801 in order to chain or run Applesoft programs, the DOS LOAD command must be used along with the A keyword to load the Applesoft file f into memory at that memory location a and to save the modified *Applesoft program address initialized* file f using the DOS SAVE command. This will allow the Applesoft CHAIN and RUN commands to utilize the A keyword successfully. Therefore, when PRGTAB does not utilize 0x0801 for the beginning address of an Applesoft chained program, all Applesoft files that utilize the DOS CHAIN command must be loaded and saved at the new address a in PRGTAB in order to establish proper *Applesoft program address initialization* for those Applesoft programs so that they will successfully function at the selected address a other than 0x0801. All Simple Variables and Array Variable descriptors will be moved to their new memory location at the end of Applesoft file f.

If the R keyword is included with the DOS LOAD command, the memory load address and the number of bytes that are read into memory are displayed after the DOS command. Figure 80 shows the DOS LOAD command with the Applesoft HELLO file along with the R keyword. In this example, the HELLO file is loaded into memory at address 0x0801 and the number of bytes that are read from this file is shown to be 0x0494.

```

JCATALOG
S=6 D=02 V=000 F=0500 02/14/24 08:28:48
 A 006 HELLO          02/14/24 08:28:48
 A 006 HELLO2         02/14/24 08:28:48
JLOAD HELLO
JLOAD HELLO,R
A$0801,L$0494
JSAVE HELLO2
JSAVE HELLO2,R
A$0801,L$0494
JSAVE HELLO2,R1
A$0801,L$0494 = 005
J

```

Figure 80. LOAD and SAVE Commands

```

Jload HELLO
Jsave HELLO3
Jsave HELLO3,r1
A$0801,L$0494 = 005
Jconfig 2
Jconfig = 002
Jsave HELLO3,r1
A$0801,L$0494
Jconfig,r
Jconfig = 000
J*

```

Figure 81. SAVE Command Display

RUN Command

RUN f [,Ss][,Dd][,Vv][,Aa][,Ll]

Example: RUN
 RUN START
 RUN START,A\$1000
 RUN START,L10

This command is available in DOS 3.3 for Applesoft File commands and this command was enhanced for DOS 4.5.08H to accept the A keyword. This Applesoft File command reads into memory at 0x0801 or at the specified address *a* if the A keyword is given, the Applesoft program that is contained in the Applesoft file *f* in the specified volume and Applesoft begins processing the program statements. The byte length of the Applesoft program is found in the first two bytes of the first data sector of the Applesoft file *f*. Applesoft files are file Type 0x02.

If the A keyword is included with the DOS RUN command, the DOS RUN handler copies the specified address *a* to PRGTAB at 0x67:68. Applesoft initializes PRGTAB during its COLDSTRT initialization to the initial value of 0x0801. The address in PRGTAB may be changed by the DOS RUN command for an Applesoft program that has already been initialized to that same address. Refer to the DOS LOAD command for further details on *Applesoft program address initialization*. Applesoft does not provide any means to re-initialize PRGTAB to the COLDSTRT value of 0x0801. The address in PRGTAB remains constant unless it is changed by another DOS Applesoft command.

If the L keyword is included with the DOS RUN command, Applesoft processing will begin at that Applesoft program line number *l* only if that line number *l* exists, otherwise the Applesoft interpreter will report an error and terminate further Applesoft processing. Obviously, this capability opens up a multitude of selective programming functionality that could be based on its program processing for selective entry program line numbers for *l*. The DOS RUN handler concludes its processing and enters the Applesoft SETPTRS routine at 0xD665 by means of the DOS ASROMCLR variable. The Applesoft SETPTRS routine loads the address that resides in PRGTAB, decrements that address, and copies that decremented address to DATPTR at 0x7D:7E and to TXTPTR at 0xB8:B9, it copies HIMEM at 0x73:74 to FRETOP at 0x6F:70, and it copies VARTAB at 0x69:6A to ARYTAB at 0x6B:6C and to STREND at 0x6D:6E. The Applesoft SETPTRS routine also forces the stack pointer to 0xF8. Next, the DOS RUN handler enters 0xD955 in the Applesoft ROM by means of the ASROMSET variable. This ROM location initializes the start address of the Applesoft program if the L keyword is included with the DOS RUN command. This Applesoft processing utilizes the Applesoft FNDLIN2 routine at 0xD61E in order to locate the Applesoft program line number *l* that the DOS RUN handler saves to LINNUM at 0x50:51. Finally, the DOS RUN handler enters the Applesoft NEWSTT routine at 0xD7D2 by means of the DOS ASROMNEW variable. The Applesoft NEWSTT routine begins to process each Applesoft statement in the Applesoft program that currently resides in memory. An example in using the Applesoft RUN command is shown previously in Figure 79.

SAVE Command

SAVE f [,Ss][,Dd][,Vv][,B][,R[1]]

Example: SAVE HELLO2
 SAVE HELLO2,B
 SAVE HELLO2,R
 SAVE HELLO2,R1

This command is available in DOS 3.3 for Applesoft File commands and was enhanced for DOS 4.1 to accept the B and the R keywords, and this command was further enhanced for DOS 4.3. This Applesoft File command saves the Applesoft program that currently resides in memory to the Applesoft file *f* in the specified volume. The Applesoft variable PRGTAB at 0x67.68 specifies the beginning address of the Applesoft program that is currently in memory and the Applesoft variable PRGEND at 0xAF.B0 specifies the ending address of the Applesoft program that is currently in memory. The DOS SAVE command utilizes both the PRGTAB and the PRGEND variables in saving an Applesoft program that is currently in memory to the Applesoft file *f* in the specified volume. Applesoft files are file Type 0x02.

If the R keyword is included with the DOS SAVE command, the save address and the number of bytes that are written to the file are displayed as shown previously in Figure 80 which is 0x0494 bytes in that example. If a non-zero R keyword is included with the DOS SAVE command, the number of verified sectors is also displayed after the number of bytes saved, again as shown previously in Figure 80.

The B keyword can be used with the DOS SAVE command in order to implement the *File Delete/File Save* strategy. That is, the Applesoft 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 CONFIG Bit 1 is **set**, the Applesoft file is **not** verified after it is saved to the specified volume. Figure 81 shows that even when a non-zero R keyword is used with the DOS SAVE command, no sectors are verified when CONFIG Bit 1 is set. The VALSCNFG variable can be cleared by using the R keyword with the DOS CONFIG command followed by a comma that is also shown in Figure 81.

Summary

As in most new investigations, the learning curve is initially steep. A great effort was employed by Microsoft to design a floating-point BASIC for one of the early microprocessors and to port that software package to other microprocessors and their respective platform. In the software ports that I witnessed at Sierra On-Line, most of the generic software was portable and some of the unique software was hardware dependent to the Apple][, to the Commodore 64, to the TRS-80, and to various other Atari PROMs. I have no doubt that some concessions were made when Microsoft ported their 8080 BASIC to the 6502 microprocessor and specifically to the Apple][platform. The assembly code for Applesoft is highly compacted which contributes to the difficulty in exploring its routines and functions. Only when I designed my own similar routines did I fully appreciate how elegantly Microsoft designed many of their routines. Keeping fundamental routines at their established memory location in ROM is another contribution to the difficulty in investigating alternative processing choices. Would it really be worth the time and the effort

to verify improvements to floating-point handling if guard bytes were also pushed onto and popped from the stack, for example? Certainly, there still remains a few routines that continue to interest and amaze me and they propel my exploration into the further depths of Applesoft. Yes, there is bad code as well as brilliant code since Applesoft is a collection of routines that were written by a collection of very unique individuals. I would hope that eventually, someday, Apple Computer, Inc., may actually publish the Applesoft source code though the number of interested parties is exponentially dwindling to merely a hair above zero.

Walland Philip Vrbancic, Jr., 2025 February 14