PET® MACHINE LANGUAGE GUIDE

By ABACUS SOFTWARE

FOR 4K/8K AND 16K/32K PETS

PET MACHINE LANGUAGE GUIDE

FOR 4K/8K AND 16K/32K PETS

BY ARNIE LEE

ABACUS SOFTWARE
P. O. Box 7211
Grand Rapids, Michigan 49510

TABLE OF CONTENTS

INTRODUCTION

For those of you who are not satisfied with programming only in the BASIC language, for those of you who cannot make BASIC run fast enough for your applications, for those of you who are curious about the inner workings of your PET, or for those of you who just want to buy another manual, this guide is dedicated.

The information contained in this guide was not readily available from a single source when writing began on it. I have had to hunt, scrape, rummage, and experiment for most of it. The on-off switch on my PET is nearly worn out from use, having had to power it off and back on after crashing the operating system countless times.

This manuscript is entitled the "PET Machine Language Guide". You are probably familiar with the low level languages or the title would not have attracted your attention. This guide is not intended to be a tutorial on programming in machine language. It is intended to be a reference for the machine language programmer who wants to use the builtin features of the PET. There is no use in "reinventing the wheel" if you can use the wheels already supplied with your PET, referring of course to the builtin ROM routines. Many programmers are commanding more than $15 per hour for their services. If this guide saves you an hour's time by showing you new or improved ways to program, then I'd like to think that the guide has more than paid for itself.

The routines used in this guide are relatively simple. The routines use a very small portion of the 6502 instruction set. The beginning machine language programmer should have little trouble learning the instructions that are used here. The beginner should stick to simple and straight-foward routines at first. Save the fancy instructions and addressing modes for a later time. Loads and stores, compares and branches, and the routines outlined here will take the beginner a long way towards understanding machine language programming.

The routines presented here will run on either the 4K/8K Pets with the older operating system or the 16K/32K Pets with the newer revised operating system. Throughout this guide the text will reference memory locations that apply to the 4K/8K Pets. The corresponding memory locations for the 16K/32K Pets will appear in parenthesis.

Although many machine language monitors are currently available for the PET, I have chosen to use the Commodore version because it is either supplied on tape to all owners of the 4K/8K PETS or it is built into ROM for owners of the 16K/32K PETS. The monitors do a nice job of allowing me to quickly enter machine language routines into the PET's memory.

I would like to express my thanks to Jim Russo of Ann Arbor, Michigan who provided me with some of the updated information about the newer 16K/32K PETS. His efforts saved me much time in revising this guide.

For those of you with criticism, comments, corrections, questions, or praise, I would be glad to hear from you. I am hoping that you feel that you are getting your money's worth from this guide. With the prices of books and manuals as high as they are these days, ABACUS SOFTWARE is trying to provide you with practical and useful information at a reasonable price.

THANK YOU.

ARNIE LEE

ABACUS SOFTWARE
P.O. BOX 7211
GRAND RAPIDS, MI 49510
May 15, 1979

NOTE-PET is a registered trademark of COMMODORE BUSINESS MACHINES, INC.

MACHINE LANGUAGE PROGRAMMING

Shortly after the PET was born, it learned to speak in a machine language dialect called 6502ese. Commodore later on, taught the PET to speak a foreign language, BASIC. Now the PET uses BASIC to talk to you because it realizes that most of you are well-versed in the BASIC language.

Since PET learned to speak 6502ese first, it prefers to think in its native language. If you speak to it in BASIC, the PET will require a little extra time to translate the BASIC into 6502ese, but it will understand you. The PET is no different from those of you who are bilingual. You may speak more than one language, but all of your thought processes are carried out in your native language.

Now BASIC is not your native language, but it more closely resembles your native language than does 6502ese. So if I talk to you in 6502ese, it will probably take you a little time to translate the ideas into terms that you can understand.

A machine language program is a series of binary instructions that directs the microprocessor to carry out very elementary tasks. The instructions perform very primitive functions - reading and writing, adding and subtracting, shifting and rotating, anding and oring - functions that involve a single memory location.

Machine language programming involves considerable effort when routines of any sizeable length are to be written. The programmer must keep account of the memory locations that the instructions reference. He must insert the representation of those locations into the series of instructions. If the programmer later on decides to insert additional instructions into the program, then he may very well have to change several of the previously calculated memory locations. Programming in machine language becomes very tedious and error-prone.

Assembly language programming is a step up from machine language programming. Machine instructions are defined mnemonically. Memory locations are symbolically referenced. An assembler program processes the assembly language statements, converts the mnemonics into their machine language equivalents and converts the symbolic references to memory locations into their appropriate binary equivalents. Programming in assembler language is much less error-prone than in machine language but remains a tedious job.

Machine language and assembler language programs operate on very primitive data elements. High level languages such as BASIC are designed to operate on more sophisticated data structures. Programming in high level languages relieves the programmer of much of the tedious work associated with the lower level languages. Programs can be written in less time and are less error-prone. You must pay a price for the advantages which a high level language provides. The price is an increase in the amount of memory used and an increase in run time. Remember that the PET has to translate the foreign language BASIC into 6502ese before it can understand what you want it to do for you. BASIC therefore runs slower than pure machine language instructions.

Now that I've convinced you that machine language programs will run circles around BASIC programs, I will lead you through the steps of building a machine language program.

Firstly you must have a clear understanding of just what it is that your program is to accomplish. You must decide upon an approach to take in building that program. Unfortunately this step is often given too little thought. The programmer is too anxious to get into the programming step that the program suffers. The author hasn't clearly thought out the method. This step is vital regardless of the programming language used. It must be addressed for a program written in BASIC as well as machine or assembly language.

For example, I decide that I need to compute the logarithm (in base 10) of various arguments. I could approach this program in several ways: a) creating a table of the logarithms and looking them up as needed; b) finding the logarithm from scratch by using a complex series of formulas; c) using a derivative of a built-in function. By using the PET's built-in function I can create a very simple program. This eliminates the need for a complex program thereby simplifying the overall task to be done, namely to compute the common logarithm for a given argument.

Secondly you have to decide where in memory you will place the machine language program. Most machine language programs will run alongside a BASIC program. So you must make sure that the machine language program is protected from destruction by BASIC or the operating system. The article following this one will cover this in detail. For most machine language programs you can use the PET's second cassette buffer (memory locations 826 to 1017).

Thirdly you have to write the program itself. Without the aid of an assembler program, you will have to hand-assemble the assembler language source. I'll leave it to you to find a way of generating the resultant machine language code. I'd advise you to keep your routines as short as possible if you must resort to hand-assembly techniques. Also take advantage of the routines that are described in this manual. They will save you much time and effort. The bibliography at the end of this manual lists several sources for assembler programs for the PET.

Forthly you have to decide how you will put the resultant machine language program into memory. You can poke the program into memory from a BASIC program. An alternative way is to use a machine language monitor. This is by far the superior alternative. By using the monitor you can also alter and display memory, and you can save and reload your programs on cassette. Of course the assembler programs will probably assemble code directly into PET's memory.

Lastly you have to decide how you will test your new programs. It isn't too often that I write a program which runs correctly the first time that I try it. Once again the machine language monitor offers some help in testing these programs. By inserting special instructions into the program to be tested, you can cause the program to temporarily halt execution. At this time you can examine the contents of memory and registers and alter them if you desire. Then you can continue execution of the program from this breakpoint. If you don't use a monitor, then you will have to test blindly. A single bad instruction in the program could hang up the PET forcing you to turn it off and back on to recover from the error.

PROTECTION OF MACHINE LANGUAGE ROUTINES

When the PET is turned on, or the reset function is called, the operating system initializes PET's memory for BASIC. It sets up its work areas, constants, and pointers so as to maximize the number of BASIC statements which will fit into the limited amount of available memory.

No problems arise with this method of initialization unless you want to use a machine language routine. The problem is: "where in memory should the routine be placed so that BASIC does not destory the routine?". Commodore recommends that you use the cassette buffer for the second tape drive. This provides you with 192 memory locations into which you can place your routine.

But what if the routine is larger than 192 memory locations? Or what if the second cassette buffer is being used? The easiest way to insure that the machine language routine will not be destroyed is by making BASIC think that the amount of memory available to it is somewhat smaller than the actual amount of memory in the PET.

On reset, the operating system determines the actual amount of memory available. The operating system does this by writing a specific character to a single memory location, and then rereading this same location. If the character read is the same as the character written, then that memory location really exists. This same procedure is then applied to successively higher memory locations. When the character read is not the same as the character written, then it is determined that the previous write and read was to the highest available memory location. This location is then placed in the pointer at $86-87 ($34-35 for 16K/32K PETS). BASIC uses this pointer to determine how much working area it can use for itself.

If you alter this pointer before BASIC begins storing statements, variables, etc., then you can protect a machine language routine from destruction by BASIC. For an 8K PET, this pointer normally contains $00 20 (least significant byte, most significant byte). If you change the pointer to $00 1C, you will protect the 1024 memory locations from $1C00 to $2000. You have in effect made BASIC unaware of the existance of those memory locations. The most straight-foward way to chane the pointer is to POKE the pointer with the altered values. Locations

$85-86 ($34-35 in 16K/32K PETS) correspond to 134-135 (52-53 in 16K/32K PETS) for POKE statements. Thus to protect a machine language routine which begins at $1C00 you would do the following:

(For 8K PET only)

```
        POKE 134,0                LSB=$00=0
        POKE 135,28               MSB=$1C=28
```

The following table illustrates the protection method for the various models of the PET.

MODEL	HIGHEST MEMORY LOCATION	VALUE TO POKE TO PROTECT 1023 MEM. LOC.		PROTECTED LOCATIONS
4K	$1000	$0C00	POKE 134,0 POKE 135,12	$0C00-1000
8K	$2000	$1C00	POKE 134,0 POKE 135,28	$1C00-2000
16K	$4000	$3C00	POKE 52,0 POKE 53,60	$3C00-4000
32K	$8000	$7C00	POKE 52,0 POKE 53,124	$7C00-8000

These statements should be executed in direct mode before any BASIC statements are stored. With the pointer altered, BASIC is not aware that memory locations greater than $1C00 for the 8K PET. A machine language program can be placed anywhere between $1C00 and $2000 and be free from destruction by BASIC.

CLOCKS AND TIMERS

Every computer has at least one clock which paces the execution of its instructions. The PET user has access to several of these clocks. You are free to use the clocks for whatever purpose you desire providing that you understand the method of operation. The following lists several of these clocks:

0 to 255

4K/8K	16K/32K	
1) $0200	$008D	increments every 1092.1667 seconds
2) $0201	$008E	increments every 4.2667 seconds
3) $0202	$008F	increments every 1/60th of a second
4) $E848	$E848	decrements every microsecond(.000001 second)
5) $E849	$E849	decrements every 256 microseconds.

The clocks at memory locations $0200-0202 ($008D-008F for 16K/32K PETS) work together to form the "jiffie" clock. The register at $0202 ($008F in 16K/32K PETS) increments every 1/60th of a second. It counts upward from 0 to 255. When it rolls over from 255 to 0, it causes the register at $0201 ($008E in 16K/32K PETS) to be incremented by one. Similarly, the register at $0201 ($008E in 16K/32K PETS) counts from 0 to 255, and when it rolls over to 0, it causes the register at $0200 ($008D in 16K/32K PETS) to be incremented by one.

Thus the register at $0201 ($008E in 16K/32K PETS) increments every 1/60 * 256 = 4.2667 seconds and the register at $0200 ($008D in 16K/32K PETS) increments every 1/60 * 256 * 256 = 1092.1667 seconds.

When the BASIC user accesses TI, the jiffie clock, he is actually accessing the three contiguous memory locations starting at $0200 ($008D for 16K/32K PETS). The BASIC statement T = TI assigns to the variable "T", the value in registers $0200-0202 ($008D-008F in 16K/32K PETS). The built-in function converts the three byte binary value at those locations to the floating point variable "T".

When you, as the BASIC user access TI$, the time of day clock, the PET software is actually converting the jiffie clock to the time of the day. The following algorithm is similar to the conversion routine that is performed by the PET in evaluating TI$

```
HH = INT(TI/(60*60*60))

MM = INT((TI-(HH*60*60*60))/(60*60))

SS = INT((TI-(HH*60*60*60)-(MM*60*60))/60)
```

The jiffie clock begins counting when the PET is turned on. It initially has a value of zero when first powered on. It continues counting upwards from zero unless reset by a BASIC assignment - TI$="HHMMSS" which converts the HHMMSS of TI$ to jiffies as below:

```
TI = (HH*60*60*60) + (MM*60*60) + (SS*60)
```

You may use the jiffie clock to time various functions. Below is an example of a routine which will inform you when ten seconds have elapsed:

```
10 S=TI:                          REM  SAVE STARTING JIFFIE COUNT
20 PRINT"START OF INTERVAL HAS BEGUN"
30 IF TI-S<10*60 THEN 30:         REM  LOOP UNTIL 10 SECS ELAPSE
40 PRINT"TEN SECONDS HAVE ELAPSED"
```

You may also use the jiffie clock to time short intervals. Below is an example of a routine which computes the time that it takes you to react to a message that is displayed on the screen. The routine will flash a message on the screen and wait for you to depress any key.

```
10 PRINT"(CLR CD CD CD)WHEN YOU SEE THE NEXT MESSAGE APPEAR
   ON"
20 PRINT"(CD)THE SCREEN, DEPRESS ANY KEY AND I WILL"
30 PRINT"(CD)MEASURE YOUR REACTION TIME."
40 PRINT"(CD)";TAB(12);"GET READY"
50 DELAY=TI:                     REM  START OF WAIT PERIOD
60 IF TI-DELAY<60*3 THEN 60:     REM  WAIT A FEW SECONDS
```

```
70 PRINT"(CLR)"
80 DELAY=TI
90 IF TI-DELAY<60*3 THEN 90:        REM  WAIT A FEW MORE SECONDS
100 PRINT"(CD CD CD CD)"
110 POKE 525,0:                     REM  IGNORE KEYS PRESSED
120 REM  POKE 158,0  for 16K/32K Pets
130 PRINT"PRESS ANY KEY NOW"        REM  GIVE GO AHEAD SIGNAL
140 S=TI:                           REM  START TIMING
150 GET A$:IF A$=""THEN 150:        REM  LOOP UNTIL A KEY PRESSED
160 P=TI:                           REM  END OF TIMING LOOP
170 PRINT"(CD CD)YOUR REACTION TIME WAS"(P-S)/60"SECONDS"
180 END
```

The above routine is suitable for measuring intervals which do not require more resolution than several jiffies. A BASIC statement may require several milliseconds for execution, so the jiffie clock cannot resolve very small time intervals.

When high resolution timing is required, you must write routines at a machine language level. The register at $E848 counts down from 255 to 0 every microsecond. When it reaches zero, it rolls over to 255 again and causes the register at $E849 to count down by one. Thus every 256 microseconds, register $E849 is decremented. Technically we call these registers timers and not clocks. The timers are similar to the familiar "oven timer". Once set, it will count down. When it reaches zero, the little bell will go off. This is similar to how the PET's timers work.

The following routine is a rough estimate of the time that is required to count to 100 in machine language. The routine uses the microsecond timers at $E848-E849. Keep in mind that they count downwards from 255 to 0.

```
ADDR   VALUE      LABEL   OPC  OPERAND     COMMENTS

033A   A9 00              LDA  #$00        reset accumulator
033C   8D 49 E8           STA  $E849       reset timer
033F   8D 48 E8           STA  $E848         "     "
0342   C8                 CLC              clear carry flag
0343   D8                 CLD              insure binary mode
0344   69 01      LOOP    ADC  #1          add to accumulator
0346   C9 64              CMP  #100  $64   compare for 100
0348   D0 FA              BNE  LOOP        loop if not done
034A   AD 48 E8           LDA  $E848       save the two
034D   AE 49 E8           LDX  $E849       ..timers in A,X
0350   00                 BRK
```

After running the above program using the Commodore machine language monitor the following results appeared:

```
4K/8K PETS        PC    SR    AC    XR    YR    SP

                 0351   xx    3A    FD    xx    xx
```

$100 - XR = 100 - FD = 03 = 3_{10}$ $\quad 3 * 256 = 768$

$100 - AC = 100 - 3A = C6 = 198_{10}$ ------ $\underline{+\ 198}$

966 micsec.

```
16K/32K PETS   PC    IRQ    SR   AC   XR   YR   SP

               0351 E62E    xx   41   FD   xx   xx
```

$100 - XR = 100 - FD = 03 = 3_{10}$ $\quad 3 * 256 = 768$

$100 - AC = 100 - 41 = BF = 191_{10}$------ $\underline{+\ 191}$

959 micsec.

The 966 microseconds (959 microseconds with 16K/32K PETS) is an interval which could not be measured with the jiffie clock whose resolution is only .01600 seconds.

SCREEN DISPLAY AND CURSOR POSITIONING

The PET uses a memory mapped video display. Characters are displayed on the screen by storing the binary coded representation into reserved memory locations. Each reserved memory location is associated with a specific screen position. Ascending memory locations are displayed on the screen from left to right, and from top to bottom, in a pattern identical to reading a page in a book. These memory locations begin at $8000 and extend to $83E7. Memory location $8000 is displayed on the screen at the upper left-hand corner while memory location $83E7 appears at the lower right-hand corner. These memory locations account for the 1000 display positions on the PET's screen.

There are two basic ways to write to the screen: by storing characters directly into the video display memory or by using the operating system's routines which in turn write to the screen.

The first method of screen display is similar to poking memory from BASIC. One character's coded representation is stored into a mapped memory location. The exact location is determined by where you want the character to appear on the screen. POKE 32768,1 will cause the letter "A" to appear at the upper left-hand corner of the screen. Location 32768 corresponds to $8000 and '1' is the coded representation for the letter "A".

The screen display hardware expects the mapped memory to contains a coded representation different from PET's ASCII code. The screen display's coded representation closely resembles the ASCII code. Bit 1 of the ASCII code is dropped. These resulting codes represent the 64 normal printable characters. By using bit 1, an additional 64 graphic characters can be represented. This gives 128 printable characters. Finally by using bit 0, the above 128 printable characters can be reversed. This allows for a total of 256 different printable characters.

The following short routine will display the characters that result from storing values into the mapped memory. The values are in ascending sequence from 0 to 255 and cause a unique character to appear on the screen. Thus poking a '0' generates a "@", '1' generates a "A", '2' generates a "B", etc.

```
10 FOR I=0 TO 255:        REM    CODED REPRESENTATION
20 :                      REM    .FOR VIDEO HARDWARE
30 :                      REM    ..AND INDEX FOR SCREEN LOC.
40 POKE 32768,I:          REM    STORE INTO MAPPED MEMORY
50 NEXT I:                REM    REPEAT 255 TIMES
60 END
```

Of course if you prefer to do the same thing in a machine language routine, the following will accomplish the same goal:

```
ADDR   VALUE      LABEL  OPC  OPERAND    COMMENTS

033A   A2 00             LDX  #00        zero index register
033C   8A         LOOP   TXA             copy into accumulator
033D   9D 00 80          STA  $8000,X    display on screen
                  *                      .via mapped memory
0340   E8                INX             bump to next character/
                  *                      .next screen position
0341   D0 F9             BNE  LOOP       repeat 255 times
0343   00                BRK
```

The second method of display involves the use of the operating system's display routines. These routines make use of the cursor position registers. When calling an output to screen routine such as WRT or STROUT, the PET will begin writing at the screen position pointed to by the cursor position registers.

These registers are located at $E0-E2 ($C4-C6 in 16K/32K PETS). The register pair at $E0-E1 ($C4-C5 in 16K/32K PETS) contains the mapped memory location (least significant byte, most significant byte) of the screen line at which the cursor is currently positioned. The register at $E2 ($C6 in 16K/32K PETS) contains the number of positions into the line at which the cursor is currently positioned. The operating system calculates the screen position by adding the contents of the register at $E2 ($C6 in 16K/32K PETS) to $E0-E1 ($C4-C5 in 16K/32K PETS) to produce the mapped memory location for the cursor display. If the cursor were positioned at line 2, position 10 of the screen, then the cursor registers would contain:

REGISTERS

```
4K/8K PETS        $E0    $E1     $E2

16K/32K PETS      $C4    $C5     $C6

CONTENTS           28     80      0A

                  LSB    MSB     -----position within line
                 -addr of line-
```

If you want to write a string at a specific location on the screen, you would set the cursor position registers with the appropriate values before calling the STROUT subroutine. The example below demonstrates how you would write the string "ABC" to the screen starting at line 11, positon 20:

```
4K/8K PETS
ADDR   VALUE       LABEL  OPC  OPERAND    COMMENTS

033A  A9 68               LDA  #$68       set up the
033C  85 E0               STA  $E0        .cursor position
033E  A9 81               LDA  #$81       ..registers to point
0340  85 E1               STA  $E1        ...to line 11
0342  A9 14               LDA  #20        ....and to
0344  85 E2               STA  $E2        .....position 20
0346  A0 03               LDY  #>STRING   point to the string
0348  A9 4E               LDA  #<STRING   ..in Y,A registers
034A  20 27 CA            JSR  STROUT     go write the string
034D  00                  BRK
034E  41 42 43            ASC  'ABC'
0351  00                  DC   $00        delineate string

16K/32K PETS
ADDR   VALUE       LABEL  OPC  OPERAND    COMMENTS

033A  A9 68               LDA  #$68       set up the
033C  85 C4               STA  $C4        .cursor position
033E  A9 81               LDA  #$81       ..register to point
0340  85 C5               STA  $C5        ...to line 11
0342  A9 14               LDA  #20        ....and to
0344  85 C6               STA  $C6        .....position 20
0346  A0 03               LDY  #>STRING   point to the string
0348  A9 4E               LDA  #<STRING   ..in Y,A registers
034A  20 1C CA            JSR  STROUT     go write the string
034D  00                  BRK
034E  41 42 43            ASC  'ABC'
0351  00                  DC   $00        delineate string
```

You will note that the operating system's routines use the PET's ASCII character representation and not the representation required by the video display hardware.

The following table shows the correspondence between the mapped memory locations and the screen line numbers:

LINE NO	MEMORY LOCATION	LINE NO	MEMORY LOCATION
1	00 80	14	08 82
2	28 80	15	30 82
3	50 80	16	58 82
4	78 80	17	80 82
5	A0 80	18	A8 82
6	C8 80	19	D0 82
7	F0 80	20	F8 82
8	18 81	21	20 83
9	40 81	22	48 83
10	68 81	23	70 83
11	90 81	24	98 83
12	B8 81	25	C0 83
13	E0 81		
	($E0-E1 4K/8K PETS) ($C4-C5 16K/32K PETS)		($E0-E1 4K/8K PETS) ($C4-C5 16K/32K PETS)

The cursor position registers may also be used for cursor positioning during input. Before using the routine RDT, the operating system's input routine using the cursor, you would set the cursor position registers with the desired values. The flashing cursor would then signal that the input is required, but the cursor would be positioned at the screen position that you requested and not at the position which the PET would have otherwise used.

WRITE A SINGLE CHARACTER TO THE SCREEN

1) Load the accumulator with the character to be displayed.

2) Call subroutine WRT at $FFD2.

EXAMPLE FOR 4K/8K or 16K/32K PETS

```
033A   A9 41            LDA  #$41      letter 'A' into ACCUM
033C   20 D2 FF         JSR  WRT       call WRT subroutine
033F   00               BRK
```

WRITE A CHARACTER STRING TO THE SCREEN

1) String may be any length containing any characters but must end with X'00'.

2) Load the Y-register with the most significant byte of the string's beginning address.

3) Load the Accumulator with the least significant byte of the string's beginning address.

4) Call subroutine STROUT at $CA27 ($CA1C for 16K/32K PETS).

EXAMPLE FOR 4K/8K PETS

```
033A  A0 03              LDY  #>STRING      MSB of string addr
033C  A9 42              LDA  #<STRING      LSB of string addr
033E  20 27 CA           JSR  STROUT        call STROUT subroutine
0341  00                 BRK
0342  41 42 43   STRING  ASC  'ABCDEF'
0345  44 45 46
0348  00                 DC   $00           delimit  string
```

EXAMPLE FOR 16K/32K PETS

```
033A  A0 03              LDY  #>STRING      MSB of string addr
033C  A9 42              LDA  #<STRING      LSB of string addr
033E  20 1C CA           JSR  STROUT        call STROUT subroutine
0341  00                 BRK
0342  41 42 43   STRING  ASC  'ABCDEF'
0345  44 45 46
0348  00                 DC   $00           delimit string
```

CLEAR THE SCREEN

1) Call the subroutine CLRSCR at $E236 ($E246 for 16K/32K PETS).

EXAMPLE FOR 4K/8K PETS

```
033A  20 36 E2          JSR  CLRSCR         go clear the screen
033D  00                BRK
```

EXAMPLE FOR 16K/32K PETS

```
033A  20 46 E2          JSR  CLRSCR         go clear the screen
033D  00                BRK
```

CARRIAGE RETURN AND LINE FEED

1) Call the subroutine CRLF at $C9D2 ($C9E2 for 16K/32K PETS).

EXAMPLE FOR 4K/8K PETS

```
033A  20 D2 C9        JSR  CRLF         go retun carr/line feed
033D  00              BRK
```

EXAMPLE FOR 16K/32K PET

```
033A  20 E2 C9        JSR  CRLF         go return carr/line feed
033D  00              BRK
```

SCROLL SCREEN ONE LINE

1) Call the subroutine SCROLL at $E559 ($E53F for 16K/32K PETS).

EXAMPLE FOR 4K/8K PETS

```
033A  20 59 E5        JSR  SCROLL       go scroll screen
033D  00              BRK
```

EXAMPLE FOR 16K/32K PETS

```
033A  20 3F E5        JSR  SCROLL       go scroll screen
033D  00              BRK
```

GET A CHARACTER FROM THE KEYBOARD

1) Call the subroutine GET at $FFE4.

2) If zero flag is set, then no key was depressed. Go to step 1.

3) If zero flag is not set, then a key was depressed. The value of the key that was depressed is now in the accumulator.

EXAMPLE FOR 4K/8K PETS and 16K/32K PETS

```
033A  20 E4 FF  GETLP   JSR  GET      call GET subroutine
033D  F0 FB             BEQ  GETLP    if no key depressed
                *                     ..then go back to start
033F  20 D2 FF          JSR  WRT      repeat the depressed
                *                     ..key on the screen
0342  00                BRK
```

INPUT FROM THE KEYBOARD

1) Initialize the X-register for keeping count of the number of characters inputted.

2) Call subroutine RDT at $FFCF

3) ASCII code of the inputted character will be returned in the accumulator.

4) If "end of input" character has been inputted (usually the return key = X'0D'), then go to step 8.

5) Store accumulator value into an input buffer. On the 4K/8K PETS you may use memory locations $000A to $005A. On 16K/32K PETS you may use memory locations ~~$0220~~ $0200 to $0250. $024F

6) Increment the X-register.

7) Go to step 2.

8) Input is now in the input buffer with the length of the inputted string in the x-register.

EXAMPLE FOR 4K/8K PETS

```
                     BUFFER=$000A
033A  A2 00                 LDX  #00         zero length register
033C  20 CF FF       INPUT  JSR  RDT         call input subroutine
033F  C9 0D                 CMP  #$0D        end of input char.?
0341  F0 06                 BEQ  DONE        yes-branch around
0343  95 0A                 STA  BUFFER,X    no-save char in buffer
0345  E8                    INX              increment length
0346  4C 3C 03              JMP  INPUT       go back for more
0349  A9 00          DONE   LDA  #0          place a null char.
034B  95 0A                 STA  BUFFER,X    ...at end of string
034D  A9 0A                 LDA  #<BUFFER    point A,Y pair to
034F  A0 00                 LDY  #>BUFFER    ..the input buffer
0351  20 27 CA              JSR  STROUT      repeat string on screen
0354  4C 8B C3              JMP  BASIC       go back to BASIC
```

EXAMPLE FOR 16K/32K PETS

```
                                   200
                   BUFFER=$~~0220~~
033A  A2 00             LDX  #00         zero length register
033C  20 CF FF  INPUT   JSR  RDT         call input subroutine
033F  C9 0D             CMP  #$0D        end of input char.?
0341  F0 07             BEQ  DONE        yes-branch around
0343  9D 20 02          STA  BUFFER,X    no-save char in buffer
0346  E8                INX              increment length
0347  4C 3C 03          JMP  INPUT       go back for more
034A  A9 00     DONE    LDA  #0          place a null char.
034C  9D 20 02          STA  BUFFER,X    ...at end of string
034F  A9 20             LDA  #<BUFFER    point A,Y pair to
0351  A0 20             LDY  #>BUFFER    ..the input buffer
0353  20 1C CA          JSR  STROUT      repeat string on screen
0356  4C 89 C3          JMP  BASIC       go back to basic.
```

NOTE-These two routines will not run under the control of the machine language monitor because the monitor itself makes use of the X-register. You may use the monitor to load memory with the above programs. After the programs are loaded, the X command will return you from the monitor to BASIC. Once in BASIC you can then enter the command SYS(826) to begin execution of the above routines.

USR FUNCTION

The USR function provides a technique for calling a machine language routine from a BASIC program. The format of the function is:

```
100 B=USR(A)
```

where A is the argument to be passed to the machine language routine
and where B is the function to be returned to the BASIC program from the machine language routine.

To use a USR function:

1) Load the machine language routine into memory by poking, loading from tape or using a monitor.
2) Set up the USR vector by placing the entry point of the machine language routine into memory locations \$0001 (LSB) and \$0002(MSB).
3) Assign a value to the variable to be passed to the machine language routine as the argument(A in the above example).
4) Call the machine language routine using the USR function.
5) The machine language routine will compute its function and place it into the floating point accumulator(\$B0-\$B4).
6) The function value in the floating point accumulator is assigned to the variable on the left-hand side of the USR call(B in the above example).

EXAMPLE:

PET BASIC does not have a logarithm function for base 10. It does have a logarithm function for base e. We can use the following equality to produce the base 10 logarithm.

$$LOG_{10}(x) = LOG_{e}(x) * LOG_{10}(e)$$

Although this is a very simple example which could very easily be implemented entirely in BASIC, we will create a short machine language routine which demonstrates the USR function.

First we must load the following machine language routine into memory (use the machine language monitor to insert the code beginning at memory location X'33A').

```
4K/8K PETS
ADDR   VALUE      LABEL   OPC  OPERAND       COMMENTS

033A   20 BF D8           JSR  LOGE          call LOGe of FACC
033D   A9 03              LDA  #>CONST       point to constant
033F   A0 48              LDY  #<CONST       ..in A,Y registers
0341   20 5E D9           JSR  MVAFAC        move CONST to AFAC
0344   20 00 D9           JSR  FPMULT        FACC = FACC * AFAC
0347   60                 RTS                return to BASIC
0348   7F 5E 5B   CONST   =$7F5E5BD89A       .43429448(log10(e))
034B   D8 9A
```

```
16K/32K PETS
ADDR   VALUE      LABEL   OPC  OPERAND       COMMENTS

033A   20 F6 D8           JSR  LOGE          call LOGe of FACC
033D   A9 03              LDA  #>CONST       point to constant
033F   A0 48              LDY  #<CONST       ..in A,Y registers
0341   20 98 D9           JSR  MVAFAC        move CONST to AFAC
0344   20 3C D9           JSR  FPMULT        FACC = FACC * AFAC
0347   60                 RTS                return to BASIC
0348   7F 5E 5B   CONST   =$7F5E5BD89A       .43429488(log10(e))
034B   D8 9A
```

Then we create the BASIC program:

```
 10 POKE 1,58:                 REM   SET UP THE USR
 20 POKE 2,3:                  REM   ..FUNCTION VECTOR
 30 A=6027:                    REM   ARGUMENT VALUE
 40 B=USR(A):                  REM   CALL MACHINE LANGUAGE ROUTINE
 50 PRINT"LOG OF";A;"IS";B:    REM   PRINT OUT ANSWER
 60 END

RUN

LOG OF 6027 IS 3.78010118
READY.
```

When BASIC encounters statement 40, it evaluates the argument A and places it into the floating point accumulator. Next a linkage is set up so that control is passed back to the USR function for evaluation of B after the machine language routine completes. Finally control is passed to the machine language routine.

The machine language routine does the following:

1) The argument passed from BASIC has already been placed into the floating point accumulator. Subroutine LOGE is called at $D8BF ($D8F6 for 16K/32K PETS) to compute the natural logarithm of this function. The result is left in the floating point accumulator.

2) A pointer is set up in the A and Y registers for the subroutine MVAFAC which will move the value pointed to by the A,Y register pair into the alternate floating point accumulator. Subroutine MVAFAC is called at $D95E ($D998 for 16K/32K PETS) thereby placing the constant into the alternate floating point accumulator.

3) Subroutine FPMULT is called to compute the product of the floating point accumulator and the alternate floating point accumulator with the result being left in the former. Thus the floating point accumulator contains the logarithm in base 10 of the argument which was passed to the machine language routine .

4) Since the product is already in the floating point accumulator (where BASIC expects the function to be), we execute a return from subroutine to go back to the BASIC program. B is evaluated in statement 40 by assigning to it the value in the floating point accumulator. Execution continues with the remainder of the BASIC program.

In this case, when we run the program we find that the function returned to out BASIC program is 3.78010118. We can verify the answer by directly entering the command:

PRINT 10↑B ----which prints out a result of 6026.99979, the difference being caused by conversion and rounding.

NUMBER REPRESENTATION

PET BASIC uses two methods of representing numbers internally. They are referred to as fixed point representation and floating point representation. Each has its advantages and disadvantages and will be discussed separately. The fixed point numbers are the easier of the two to understand and will be introduced first.

FIXED POINT NUMBERS

Fixed point numbers are often referred to as integers. Variables of this type may assume only integer values, that is they may not have any fractional portions. Fixed point variables are specified by using a variable name with a '%', such as A% or B2%. A fixed point number is stored in two memory locations inside the PET. Another way of putting this is that 16 bits are required for each fixed point representation. One of these bits is the sign bit. Variables may therefore range in value from -32768 to +32767.

The high order bit is called the sign bit. A zero in this position indicates a positive number, while a one in this position indicates a negative number. The remaining 15 bits represent the magnitude of the number. Positive numbers are represented in true binary form. Negative numbers are represented in two's complement form.

EXAMPLE-

What is the fixed point representation of +1000?

$$1000_{10} = \underbrace{\overset{\substack{\text{sign bit}\\ \downarrow}}{0}\overbrace{000\ 0011}}_{\text{MSB}}\ \overbrace{\underbrace{1110\ 1000}_{\text{LSB}}}^{\text{true binary}} = \text{X'03E8'}$$

What is the fixed point representation of -1000?

$$-1000_{10} = \underbrace{\overset{\substack{\text{sign bit}\\ \downarrow}}{1}\overbrace{111\ 1100}}_{\text{MSB}}\ \overbrace{\underbrace{0001\ 1000}_{\text{LSB}}}^{\text{two's complement}} = \text{X'FF18'}$$

Fixed point numbers as used in PET BASIC have a 15 bit precision and would normally require only two memory locations. However single(non-array) fixed point variables actually occupy five memory locations thus wasting the remaining tree. Arrays of fixed point numbers occupy only two memory locations for each element with no waste.

You can save memory by using arrays of fixed point numbers if the limited range in values is suitable for your application. We can demonstrate the savings of memory between fixed and floating point numbers in two short programs.

```
 10 REM   ***FIXED POINT EXAMPLE***
 20 DIM A%(100)
 30 PRINT FRE(0)
RUN

6931
```

```
 10 REM   ***FLOATING POINT EXAM***
 20 DIM AA(100)
 30 PRINT FRE(0)
RUN

6628
```

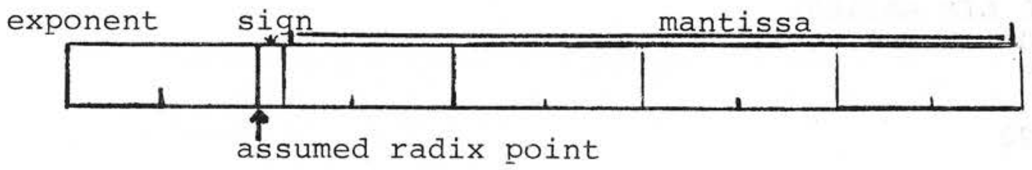
In each example we merely dimensioned two arrays and then displayed the amount of free space that remained. The difference in free memory locations is 6931 - 6628 = 303. The savings in memory of the fixed point program over the floating point program is the result of saving 3 memory locations for each fixed point element in the array (100 elements + 0th element).

Arithmetic operations using fixed point numbers are faster than when using floating point numbers. The 6502 instructions for addition and subtraction are designed to operate directly with numbers represented in the fixed point format.

FLOATING POINT NUMBERS

Floating point variables may assume fractional as well as integer values. They are useful because they may take on values with a very wide range. Floating point variables are specified by default. If they are not integer('%') nor string('$') then the variable is assumed to be of the floating point type. Each floating point number occupies five memory locations inside the PET whether it be a single element or an array.

The format of a floating point number is as follows:



The exponent occupies one memory location. The exponent is a power of two but is stored in excess 128 notation. This means that 128 has been added to the true exponent to allow for the easier handling of negative exponents. Thus if the exponent of a number is 16, then you would add 128 yielding 144 = X'90'. This last value, X'90' would be stored as the floating point exponent. Similarly if the exponent of a number is -12, then you would add 128 yielding 116 = X'74' which you would store as the floating point exponent.

The mantissa is the fractional part of the floating point number. It is always normalized. This means that the fraction has been adjusted and the exponent likewise adjusted until the most significant binary digit of the mantissa is to the right of the assumed radix point.

Since PET BASIC always normalized the mantissa, the leftmost bit of the fraction is always a one. This is redundant and so the position is used for another purpose. This bit is used as the sign bit for the number. Thus the mult fraction repeatedly by 2 and keeptrack of fract > 1

fraction has an "assumed" most significant bit. A zero sign bit indicates a positive floating point number, while a one sign bit indicates a negative floating point number.

The example below shows how you may derive the representation for a floating point number.

PROBLEM-What is the floating point representation for 1000_{10}?

1) 1000_{10} . = 0000 0011 1110 1000_2 . = X'03E8'
radix radix

2) Shifting the radix point 10 positions to the left in order to normalize the fraction while raising the exponent by a power of 10 gives us:

.1111 1010 0000 0000 * 2^{10}

3) The number is positive so we can set the sign bit to zero.

.0111 1010 0000 0000 * 2^{10}

4) The exponent is 10, but in excess 128 notation the exponent is 10 + 128 = 138 = 1000 1010_2

5) Combining the exponent and mantissa we find that the resultant appears as follows:

1000 1010 0111 1010 0000 0000 0000 0000 0000 0000_2

which if restated in hexadecimal would be:

8A 7A 00 00 00

6) A negative value would have a sign bit of one. Thus -1000_{10} would be represented as:

1000 1010 1111 1010 0000 0000 0000 0000 0000 0000_2

or in hexadecimal form:

8A FA 00 00 00

Of course PET BASIC carries the precision further than the examples above have shown, but the method is the same. As stated at the beginning of this section, floating point variables may take on an extremely wide range of values.

By experimenting with BASIC you will find what this range is for the PET.

```
PRINT 2↑126.99999995

 1.70141174 E+38


PRINT 2↑-127

 5.87747176 E-39
```

FLOATING POINT ARITHMETIC

The following descriptions are the fundamental arithmetic operations that PET BASIC performs on its floating point variables. The descriptions will allow you to use these routines from a machine language program.

ADDITION- FACC = FACC + AFAC

1) Place the first operand in the floating point accumulator.
2) Insure that the format of the number in the FACC is non-true binary. The most significant bit of $B1 ($5F for 16K/32K PETS) must be off if the number is positive or on if negative and the sign bit of $B5 ($63 for 16K/32K PETS) must be off if positive and on if negative.
3) Place the second operand in the alternate floating point accumulator.
4) Insure that the format of the number in the AFAC is non-true binary. The most significant bit of $B9 ($67 for 16K/32K PETS) must be off if the number is positive or on if negative and the sign bit of $BD ($6B for 16K/32K PETS) must be off if positive and on if negative.
5) Call the subroutine FPADD at $D73F ($D776 for 16K/32K PETS).
6) The sum will be found in the floating point accumulator.

SUBTRACTION- FACC = AFAC - FACC

1) Place the subtrahend into the alternate floating point accumulator.
2) Insure that the format of the number in the AFAC is non-true binary.
3) Place the minuend into the floating point accumulator.
4) Insure that the format of the number in the FACC is non-true binary.
5) Call the subroutine FPSUB at $D728 ($D736 for

16K/32K PETS).

6) The difference will be found in the floating point accumulator.

MULTIPLICATION- FACC = AFAC * FACC

1) Place the multiplicand into the alternate floating point accumulator.
2) Insure that the format of the number in the AFAC is non-true binary.
3) Place the multiplier into the floating point accumulator.
4) Insure that the format of the number in the FACC is non-true binary.
5) Call the subroutine FPMULT at $D900 ($D937 for 16K/32K PETS).
6) The product will be found in the floating point accumulator.

DIVISION- FACC = AFAC / FACC

1) Place the divident into the alternate floating point accumulator.
2) Insure that the format of the number in the AFAC is non-true binary.
3) Place the divisor into the floating point accumulator.
4) Insure that the format of the number in the FACC is non-true binary.
5) Call the subroutine FPDIV at $D9E4 ($DA1E for 16K/32K PETS).
6) The quotient will be found in the floating point accumulator.

EXPONENTIATION- FACC = AFAC ↑ FACC

1) Place the base into the alternate floating point accumulator.
2) Insure that the format of the number in the AFAC is non-true binary.
3) Place the exponent into the floating point accumulator.
4) Insure that the format of the number in the FACC is non-true binary.
5) Call the subroutine FPEXP at $DE2E ($DE68 for 16K/32K PETS).
6) The result will be found in the floating point accumulator.

ARITHMETIC FUNCTIONS

The following descriptions are the arithmetic functions that PET BASIC performs on its floating point variables. The descriptions will allow you to use these routines from a machine language program.

ABS - compute the absolute value of the argument.

1) Place the floating point argument into the floating point accumulator.
2) Call the subroutine ABS at $DB2A ($DB64 for 16K/32K PETS).
3) Function is returned in the floating point accumulator.

ATN - compute the arctangent of the argument.

1) Place the floating point argument into the floating point accumulator.
2) Call the subroutine ATN at $E048 ($E08C for 16K/32K PETS).
3) Function (expressed in radians) is returned in the floating point accumulator.

COS - compute the cosine of the argument.

1) Place the floating point argument (expressed in radians) into the floating point accumulator.
2) Call the subroutine COS at $DF9E ($DFD8 for 16K/32K PETS).
3) Function is returned in the floating point accumulator.

EXP - compute the exponential function of an argument.

1) Place the floating point argument into the floating point accumulator.
2) Call the subroutine EXP at $DEA0 ($DEDA for 16K/32K PETS).
3) Function is returned in the floating point accumulator.

INT - computes the largest integer less than or equal to the floating point argument.

1) Place the floating point argument into the floating point accumulator.
2) Call the subroutine INT at $DB9E ($DBD8 for 16k/32K PETS).
3) Function is returned in the floating point accumulator.

LOG - computes the natural logarithm of an argument.

1) Place the floating point argument into the floating point accumulator.
2) Call the subroutine LOG at $D8BF ($D8F6 for 16K/32K PETS).
3) Function is returned in the floating point accumulator.

SIN - computes the sine of an argument.

1) Place the floating point argument (expressed in radians) into the floating point accumulator.
2) Call the subroutine SIN at $DFA5 ($DFDF for 16K/32K PETS).
3) Function is returned in the floating point accumulator.

SQR - computes the square root of an argument

1) Place the floating point number into the floating point accumulator.
2) Call the subroutine SQR at $DE24 ($DE5E for 16K/32K PETS).
3) Function is returned in the floating point accumulator.

TAN - compues the tangent of an argument.

1) Place the floating point argument(expressed in radians) into the floating point accumulator.
2) Call the subroutine TAN at $DFEE ($E028 for 16K/32K PETS).
3) Function is returned in the floating point accumulator.

CONVERT INTEGER TO FLOATING POINT

1) Load the Y-register with the least significant byte of the integer to be converted.

2) Load the Accumulator with the most significant byte of the integer to be converted.

3) Call subroutine INTFLP at $D278 ($D26D for 16K/32K PETS).

4) The floating point number is returned in the floating point accumulator $B0-B4 ($5E-62 for 16K/32K PETS) with the sign in a separate location $B5 ($63 for 16K/32K PETS). A zero in the most significant bit of $B5 ($63 for 16K/32K PETS) indicates a positive number while a one in the bit indicates a negative number. Note that the mantissa is in true binary form (normalized bit is not assumed).

EXAMPLE FOR 4K/8K PETS

```
033A D8                CLD            insure binary mode
033B AD 51 03          LDA  INT+1     least significant
033E A8                TAY            ..byte to Y-register
033F AD 50 03          LDA  INT       most significant byte
0342 20 78 D2          JSR  INTFLP    convert to floating point
0345 00                BRK
 .
 .
0350 03 E8        INT  WORD 1000
```

EXAMPLE FOR 16K/32K PETS

```
033A D8                CLD            insure binary mode
033B AD 51 03          LDA  INT+1     least significant
033E A8                TAY            ..byte to Y-register
033F AD 50 03          LDA  INT       MSB into ACCUM
0342 20 6D D2          JSR  INTFLP    convert to floating point
0345 00                BRK
 .
0350 03 E8        INT  WORD 1000
```

RESULT:	4K/8K PETS		$B0	$B1	$B2	$B3	$B4	$B5
	16K/32K PETS		$5E	$5F	$60	$61	$62	$63
	+ 1000	03E8	8A	FA	00	00	00	00
	- 1000	FC18	8A	FA	00	00	00	FF

CONVERT FLOATING POINT TO INTEGER

1) Place the floating point number into the floating point accumulator $B0-B4 ($5E-62 for 16K/32K PETS).

2) Set $B1 ($5F for 16K/32K PETS) most significant bit on to make the number a true binary representation.

3) Place the sign of the number into the floating point sign location $B5 ($63 for 16K/32K PETS). A zero in the most significant bit indicates a positive number while a one in the bit indicates a negative number.

4) Call the subroutine FLPINT at $D0A7 ($D09A for 16K/32K PETS).

5) The integer will be returned at location $B3-B4 ($61-62 for 16K/32K PETS) in true binary form if positive or in two's complement form in negative.

EXAMPLE FOR 4K/8K PETS

```
                 FACC  EQU  $00B0
033A  D8               CLD            insure binary mode
033B  A2 00            LDX  #0        zero X-index register
033D  86 B5            STX  $B5       set FP sign to positive
033F  BD 60 03 LOOP    LDA  FPNUM,X   load part of FP number
0342  95 B0            STA  FACC,X    store in FP accum.
0344  E8               INX            increm. index register
0345  E0 05            CPX  #5        are we done?
0347  D0 F6            BNE  LOOP      no-go back for more
0349  A9 80            LDA  #$80      mask into accumulator
034B  0D 61 03         ORA  FPNUM+1   OR in the rest of byte
034E  85 B1            STA  FACC+1    make FACC true binary
0350  A9 80            LDA  #$80      pick up mask again
0352  2D 61 03         AND  FPNUM+1   AND to determine sign
0355  F0 02            BEQ  POSIT     skip if positive
0357  C6 B5            DEC  $B5       set sign to $FF
0359  20 A7 D0         JSR  FLPINT    call FLPINT subroutine
035C  00               BRK
 .
0360  8A 7A 00 FPNUM=$8A7A000000
0363  00 00
```

EXAMPLE FOR 16K/32K PETs

```
                    FPAC    EQU  $005E
033A   D8                   CLD            insure binary mode
033B   A2 00                LDX  #0        zero X-index register
033D   86 63                STX  $63       set FP sign to positive
033F   BD 60 03 LOOP        LDA  FPNUM,X   load part of FP number
0342   95 5E                STA  FACC,X    store in FP accum.
0344   E8                   INX            increm. index register
0345   E0 05                CPX  #5        are we done?
0347   D0 F6                BNE  LOOP      no-go back for more
0349   A9 80                LDA  #$80      mask into accumulator
034B   0D 61 03             ORA  FPNUM+1   OR in rest of byte
034E   85 5F                STA  FACC+1    make true binary
0350   A9 80                LDA  #$80      pick up mask again
0352   2D 61 03             AND  FPNUM+1   AND to determine sign
0355   F0 02                BEQ  POSIT     skip if positive
0357   C6 63                DEC  $63       set sign to $FF
0359   20 9A D0             JSR  FLPINT    call FLPINT subroutine
035C   00                   BRK
 .
0360   8A 7A 00 FPNUM=$8A7A000000
0363   00 00
```

```
RESULT -
                +1000                      -1000

FPNUM     8A 7A 00 00 00             8A FA 00 00 00
FACC      8A FA 00 00 00 00          8A FA 00 00 00 FF
AFTER              03 E8                      FC 18

 4K/8K -$B0 B1 B2 B3 B4 B5          $B0 B1 B2 B3 B4 B5
16K/32K-$5E 5F 60 61 62 63          $5E 5F 60 61 62 63
```

CONVERT ASCII NUMBER STRING TO INTEGER

1) The number to be converted must be in ASCII format and have a value less than 64000. The last character must be a blank.

2) Set the program pointer at $00C9-00CA ($0077-0078 for 16K/32K PETS) to point immediately in front of the ASCII string.

3) Call the subroutine CHRGET at $00C2 ($0070 for 16K/32K PETS).

4) Call the subroutine ASCINT at $C863 ($C873 for 16K/32K PETS).

5) The fixed point number will be returned at memory locations $0008-0009 ($0011-0012 for 16K/32K PETS).

EXAMPLE FOR 4K/8K PETS

```
033A  A2 4F        LDX #<NUMBER-1    set up pointer
033C  86 C9        STX $C9           ..at $C9-CA to
033E  A2 03        LDX #>NUMBER-1    ....point just in front
0340  86 CA        STX $CA           ......of ASCII number
0342  20 C2 00     JSR CHRGET        go scan string
0345  20 63 C8     JSR ASCINT        go convert to integer
0348  00           BRK
 .
0350  31 32 33     ASC '1234 '
0353  34 20
```

EXAMPLE FOR 16K/32K PETS

```
033A  A2 4F        LDX #<NUMBER-1    set up pointer
033C  86 77        STX $77           .at $77-78 to
033E  A2 03        LDX #>NUMBER-1    ..point just in front
0340  86 78        STX $78           ...of ASCII string
0342  20 70 00     JSR CHRGET        go scan string
0345  20 73 C8     JSR ASCINT        go convert to integer
0348  00           BRK
 .
0350  31 32 33     ASC '1234 '
0353  34 20
```

```
RESULT  $0008-0009  will contain  D2 04 (LSB,MSB) 4K/8K PETS
        $0011-0012  will contain  D2 04 (LSB,MSB) 16K/32K PETS
```

CONVERT ASCII TO FLOATING POINT

Through my experiments over the past few months, I have not found a direct way to convert a number from its ASCII representation to floating point representation. However the following method may be used as an alternate way to convert to floating point representation.

The method is based upon the BASIC USR function which is described elsewhere in this guide. The USR function evaluates an argument, converts it to floating point representation and places it into the floating point accumulator before giving control to a machine language routine. If you could examine the floating point accumulator after the evaluation of the argument then you would have let BASIC do the conversion for you.

The following BASIC program places a short machine language routine into the second cassette buffer. This routine moves the contents of the floating point accumulator to a save area where it may later be examine by the BASIC program. It is necessary to relocate the contents of the floating point accumulator because upon return to BASIC from the machine language routine its contents will be destroyed by subsequent BASIC statement execution.

The machine language routine for 4K/8K PETS is as follows:

```
                      FACC  EQU $00B0
033A  A2 05                 LDX #5       set up to save 6 locations
033C  B5 B0           LOOP  LDA FACC,X   load FACC indexed by X
033E  9D 47 03              STA SAVE,X   save value in 'safe' place
0341  CA                    DEX          decrement X-register
0342  10 F6                 BPL LOOP     if positive or
0344  F0 F8                 BEQ LOOP     ..or zero keep saving
0346  60                    RTS          otherwise return to BASIC
0347  00 00 00         SAVE=$000000000000     save area
034A  00 00 00
```

The machine language routine for 16K/32K PETS is as follows:

```
                          94d
                FACC  EQU $005E
033A  A2 05           LDX #5     set up to save 6 locations
033C  B5 5E     LOOP  LDA FACC,X load FACC indexed by X-reg
033E  9D 47 03        STA SAVE,X save value in 'safe' place
0341  CA              DEX        decrement index register
0342  10 F6           BPL LOOP   if positive
0344  F0 F8           BEQ LOOP   .or zero keep saving
0346  60              RTS        otherwise return to BASIC
0347  00 00 00  SAVE=$000000000000 save area
034A  00 00 00
```

The BASIC program which will place the machine language routine into the second cassette buffer and then display the converted numbers is shown below:

```
10 FOR I=0 to 18:                 REM  MACHINE LANG ROUTINE 19 LOC.
20 READ XX:                       REM  READ VALUE TO BE POKED
30 POKE 826+I,XX:                 REM  POKE INTO CASSETTE BUFFER
40 NEXT I:          94            REM  REPEAT FOR ALL DATA
50 DATA 162,5,181,~~176~~,157,71,3 REM  16K/32K PETS SUBSTITUTE
55 REM                                 '94' FOR '176' ABOVE
60 DATA 202,16,246,240,248,96
70 DATA 0,0,0,0,0,0
80 DIM B%(5)
90 HEX$="0123456789ABCDEF":       REM  STRING FOR HEX DISPLAY
100 PRINT"(CLR)ASCII TO FLOATING POINT DISPLAY"
110 PRINT:PRINT:PRINT"NUMBER TO CONVERT    F.P. VALUE":PRINT
120 POKE 1,58:POKE 2,3: 4926      REM  SET UP USR FUNCTION VECTOR
130 INPUT A:↑                     REM  READ A VALUE TO BE CONVERTED
140 PRINT"(CU)";TAB(20);
150 X=USR(A):                     REM GO SAVE F.P. NUMBER
160 FOR I=0 TO 5:                 REM  LOOP FOR RETRIEVING VALUES
170 B%(I)=PEEK(839+I):            REM  MOVE TO ARRAY B%
180 NEXT I
190 FOR I=0 TO 4
200 IF I=1 AND B%(5)<128 THEN B%(1)=B%(1)-128: REM  CONVERT
    SIGN BIT FROM SECOND BYTE
210 XH%=B%(I)/16:                 REM  CONVERT B-ARRAY TO HEX CHAR
220 XL%=B%(I)-(XH%*16):           REM       "          "    "    "    "
230 XH$=MID$(HEX$,XH%+1,1):       REM       "          "    "    "    "
240 XL$=MID$(HEX$,XL%+1,1):       REM       "          "    "    "    "
```

```
 250 XX$=XH$+XL$+" "
 260 PRINT XX$;:                REM  DISPLAY CONVERTED VALUE
 270 NEXT I
 280 PRINT
 290 GOTO 130

READY.
RUN

ASCII TO FLOATING POINT DISPLAY

 NUMBER TO CONVERT   F.P. VALUE
 ? 1000              8A 7A 00 00 00
 ? -1000             8A FA 00 00 00
 ? 6027              8D 3C 58 00 00
 ? 1.524             81 43 12 6E 97
 ? 1.5E12            A9 2E 9F 7B CC
 ? 1.5E-12           59 53 1B 32 10
 ? -1.5E-12          59 D3 1B 32 10
```

CONVERT FLOATING POINT NUMBER TO ASCII

1) Place the floating point number into the floating point accumulator $B0-B4 ($5E-62 for 16K/32K PETS).

2) Set $B1 ($5F in 16K/32K PETS) most significant bit on to make the number a true binary representation.

3) Place the sign of the number into the floating point sign location $B5 ($63 for 16K/32K PETS). A zero in the most significant bit indicates a positive number while a one in the bit indicates a negative number.

4) Call subroutine FLPASC at $DCAF ($DCE9 for 16K/32K PETS).

5) ASCII representation will be found beginning at location $0100 and continuing until X'00' character.

* The routine MVFACC will move the FP number to the floating point accumulator, generate the proper sign and make the number true binary.

EXAMPLE FOR 4K/8K PETS

```
033A  D8                CLD           insure binary mode
033B  A9 03             LDA #>FPNUM   MSB of FP number addr
033D  85 71             STA $71       save in pointer
033F  A9 58             LDA #<FPNUM   LSB of FP number addr
0341  85 72             STA $72       save in pointer
0343  20 78 DA          JSR MVFACC    move number to FACC
0346  20 AF DC          JSR FLPASC    convert to ASCII
0349  A9 00             LDA #$00      point A,Y pair
034B  A0 01             LDY #$01      ..to $0100
034D  20 26 CA          JSR STROUT    display string on screen
0350  00                BRK
 .
0358  8A 7A 00  FPNUM=$8A7A000000
035B  00 00
```

EXAMPLE FOR 16K/32K PETS

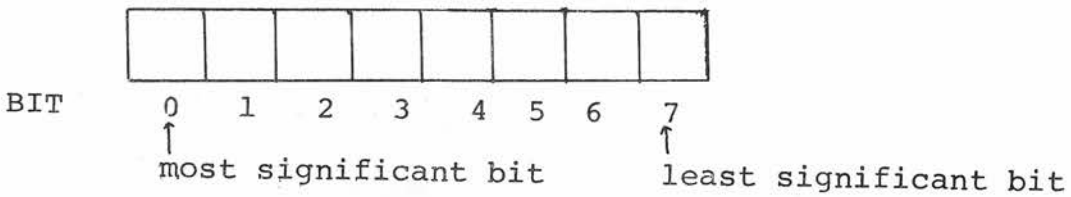
```
033A  D8                  CLD            insure binary mode
033B  A9 03               LDA #>FPNUM    MSB of FP number addr
033D  85 19               STA $19        save in pointer
033F  A9 58               LDA #<FPNUM    LSB of FP number addr
0341  85 20               STA $20        save in pointer
0343  20 AE DA            JSR MVFACC     move number to FACC
0346  20 E9 DC            JSR FLPASC     convert to ASCII
0349  A9 00               LDA #$00       point A,Y
034B  A0 01               LDY #$01       ..to $0100
034D  29 1C CA            JSR STROUT     display string on screen
 .
0358  8A 7A 00   FPNUM=$8A7A000000
035B  00 00
```

SYMBOLS

THAT ARE USED IN THS MANUAL

AFAC	-	ALTERNATE FLOATING POINT ACCUMULATOR
(CLR)	-	SYMBOL FOR CLEAR SCREEN CHARACTER
(CD)	-	SYMBOL FOR CURSOR DOWN CHARACTER
(CU)	-	SYMBOL FOR CURSOR UP CHARACTER
FACC	-	FLOATING POINT ACCUMULATOR
LSB	-	LEAST SIGNIFICANT BYTE
MSB	-	MOST SIGNIFICANT BYTE
$	-	SYMBOL INDICATING THAT THE FOLLOWING NUMBERS ARE IN HEXADECIMAL REPRESENTATION
>	-	SYMBOL MEANING MOST SIGNIFICANT BYTE
<	-	SYMBOL MEANING LEAST SIGNIFICANT BYTE

BIT CONVENTION



$E775 output hex in accum

ROUTINES THAT ARE

IDENTIFIED IN THIS MANUAL

NAME	4K/8K	16K/32K	
ABS	$DB2A	$DB64	computes the absolute value of the floating point argument
ASCINT	$C863	$C873	converts ASCII string to integer
ATN	$E048	$E08C	computes the arctangent of the floating point argument
CLRSCR	$E236	$E246	clears the screen
COS	$DF9E	$DFD8	computes the cosine of the floating point argument
CRLF	$C9D2	$C9E2	forces carriage return and line feed
EXP	$DEA0	$DEDA	computes the exponential function of the floating point argument
FLPASC	$DCAF	$DCE9	converts floating point to ASCII
FLPINT	$D0A7	$D09A	convert floating point to integer
FPADD	$D73F	$D776	adds two floating point numbers
FPDIV	$D9E4	$DA1E	divides one floating point number by another
FPEXP	$DE2E	$DE68	computes the value of one floating number raise to second floating point number
FPMULT	$D900	$D93C	multiplies two floating point numbers
FPSUB	$D728	$D736	subtracts one floating point number from another
GET	$FFE4	$FFE4	get a character from the keyboard
INT	$DB9E	$DBD8	computes integer function of a floating point number
INTFLP	$D278	$D26D	convert integer to floating point
LOG	$D8BF	$D8F6	computes the log function of a floating point argument
MVAFAC	$D95E	$D998	move floating point number pointed to by A,Y to AFAC
MVFACC	$DA78	$DAAE	move floating point number pointed to by A,Y to FACC
RDT	$FFCF	$FFCF	input character from screen
SCROLL	$E559	$E53F	scroll the screen up one line
SIN	$DFA5	$DFDF	computes sine of floating point arg.
SQR	$DE24	$DE5E	computes square root of floating point argument
STROUT	$CA27	$CA1C	write a character string to screen
TAN	$DFEE	$E028	computes tangent of a floating point argument
WRT	$FFD2	$FFD2	write a character to the screen

BASIC $C389 warmstart

MACHINE LANGUAGE INFORMATION

Foster, Caxton C., Programming a Microcomputer: 6502, Addison-Wesley, 1978

Fylstra, Daniel, "6502 Assembler in BASIC", computer program on cassette for the PET from PERSONAL SOFTWARE, P.O. Box 136, Cambridge, MA 02138

McCann, Michael J., "A Simple 6502 Assembler for the PET", Micro-The 6502 Journal, No. 6 AUG-SEPT 1978, pp 17-21

Zaks, Rodney, Microcomputer Programming: 6502, Sybex, 1978, from Sybex 2020 Milvia St., Berkeley, CA 94704

Zimmerman, Mark, "Assembler for the PET", Personal Computing, DEC 1978, pp 42-45.

OTHER SOURCES OF INFORMATION

CBM USER MANUAL, Commodore Business Machines, 3300 Scott Blvd., Santa Clara, CA 95051, free with the Pet, otherwise $10.

MICRO-The 6502 Journal, Micro Ink, Inc., P.O. Box 6502, Chelmsford, MA 01824 $15/yr (12 issues).

PET GAZETTE, 1929 Northport Dr. Room 6, Madison, WI 52704, free but donations accepted(6 issues/yr).

PET USER MANUAL, Commodore Business Machines, 3300 Scott Blvd., Santa Clara, CA 95051, free with the Pet, otherwise $10.

PET USER NOTES, P.O. Box 371, Montgomeryville, PA 18936, $6/yr (six issues).

RECREATIONAL COMPUTING, People's Computer Company, 1263 El Camino Real, Box E, Menlo Park, CA 94025, $10/yr (6 issues).

SHPINX PET NEWSLETTER, Lawrence Hall of Science, Computer Project, University of California, Berkeley, CA 94720, $4.50/yr (6 issues).

THE PAPER, Aresco, P.O. Box 1142, Columbia, MD 21044, $15/yr (10 issues).