Chapter 7

System Information

CBM COMPUTER SYSTEM ORGANIZATION

The CBM computer uses a 6502 microprocessor. The display screen, cassette tape unit, keyboard diskette drives and printer are physical devices that have been described in Chapter 2. The three external I/O ports are interfaced through the 2K block of memory-mapped I/O. The organization of the CBM computer system is shown in Figure 7-1. On 4K/8K PETs, the cassette tape unit connects directly to the I/O block, and the Cassette Tape Interface is available for connecting a second cassette unit. On 16K/32K PETs the cassette tape unit is connected through the Cassette Tape Interface; additional tape units, if any are desired, must be interfaced through the IEEE 488 port. Such tape units would operate under different protocol than standard tape units. The six ROM, RAM, and I/O blocks are allocated from the total 65K bytes of available memory (1K = 1024).

Memory allocation by 4K blocks is shown in Table 7-1. Each portion of the memory is described in more detail in the following text.

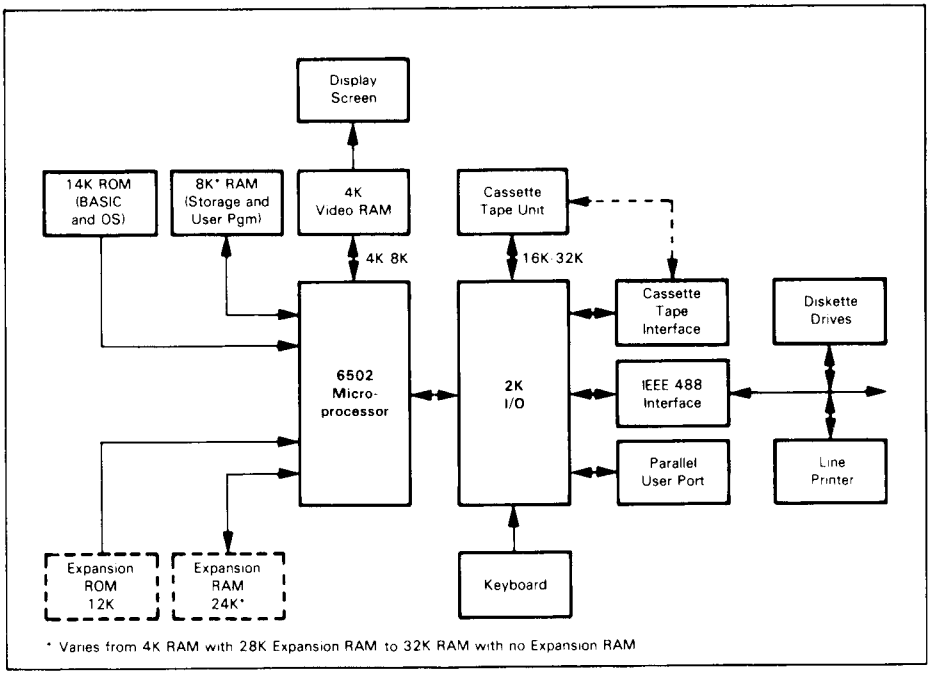


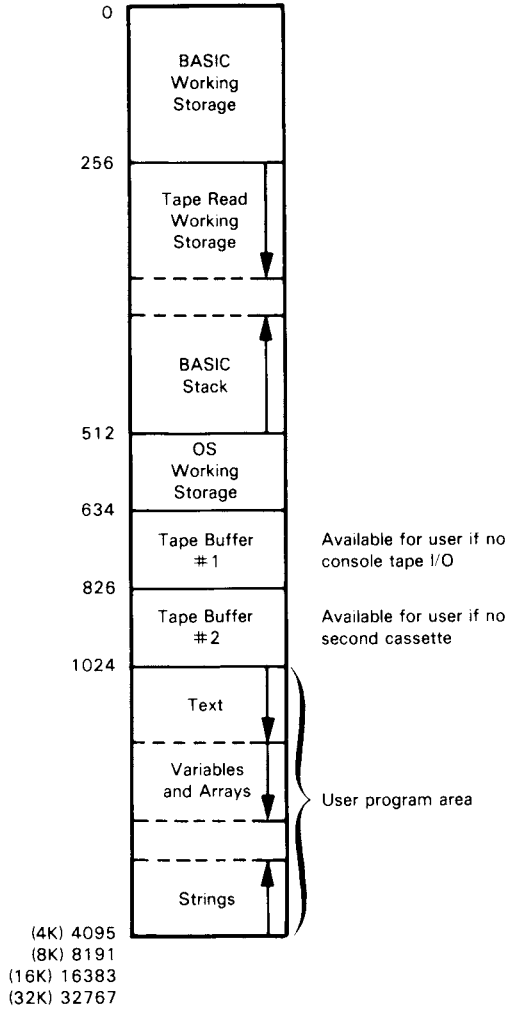
Figure 7-1. PET Block Diagram

Table 7-1. Memory Allocation by 4K Blocks

Block	Memory Type	Start Address		Description
		Decimal	Hexadecimal	
0	RAM	0	0000	Working storage, start of text
1	RAM	4096	1000	Text and variable storage (8K only)
2	—	8192	2000	Expansion RAM
3	—	12288	3000	
4	—	16384	4000	
5	—	20480	5000	
6	—	24576	6000	
7	—	28672	7000	
8	RAM	32768	8000	Screen Memory (and I/O — BASIC 4.0 only)
9	ROM	36864	9000	Expansion ROM
10	ROM	40960	A000	
11	ROM	45056	B000	Start of BASIC 4.0
12	ROM	49152	C000	BASIC (principally statement interpreter)
13	ROM	53248	D000	BASIC (principally math package)
14	ROM	57344	E000	Screen Editor (2K)
	I/O	59392	E800	I/O Memory (2K)
15	ROM	61440	F000	Operating System (OS)

Addresses 0-8191: 8K RAM (Storage and User Program)

The first block of RAM is allocated to working storage, the stack, tape buffers, and storage of user programs. The amount of active RAM may be 4K (addresses 0-4095), **8K** (addresses 0-8191), **16K** (addresses 0-16384), **or 32K** (addresses 0-32767). The first 1K allocation (to 1024) is fixed; the larger the memory size, the more space is available in the user program area.



Locations 0 through 255 are used by the BASIC interpreter as working storage locations. This area is detailed in Appendix F.

Locations 256 through 511 are used mainly by the BASIC Stack. A portion of the area beginning at location 256 and proceeding upward is used by the Tape Read routine for error correction and by BASIC as an expansion buffer. The stack begins at location 511 and proceeds downward. Storage is allocated dynamically as needed. An OUT OF MEMORY error occurs if the stack pointer reaches the end of available space in this area.

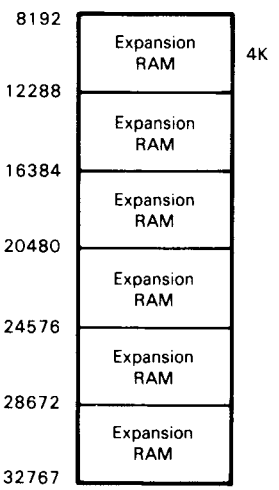
Locations 512 through 633 are used by the "Operating System" (OS) as working storage locations. This area is detailed in Appendix F.

Locations 634 through 825 form a 192-byte tape buffer for the console tape cassette. Locations 826 through 1023 form a second 192-byte tape buffer for the optional second cassette unit. User-written assembly language programs can be stored in tape buffers if there are no tape cassettes, or no second cassette in the system.

Locations 1024 through the end of available RAM are used to store user programs and variables. Programs begin at location 1024 and are stored upward toward the end of memory. Variable storage begins after the end of the program. Array storage begins at the end of variable storage. Strings are stored beginning at the end of memory and working downward. An OUT OF MEMORY error occurs if an upgoing pointer meets the downgoing pointer.

Addresses 8192-32767: Expansion RAM 24K

Memory addresses 8192 through 32767 are allocated for expansion of RAM to 32K.

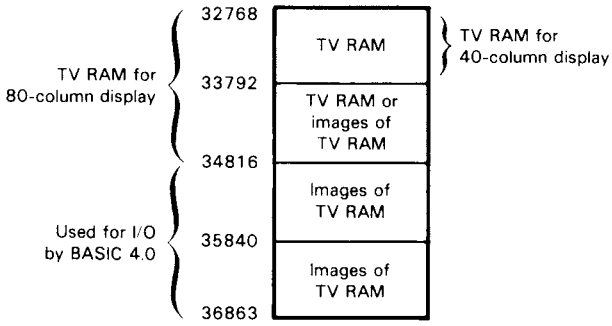


32K of RAM address space is allocated between active RAM and expansion RAM, as follows:

Active RAM	Expansion RAM
4K (0-4095)	28K (4096-32767)
8K (0-8191)	24K (8191-32767)
16K (0-1638)	

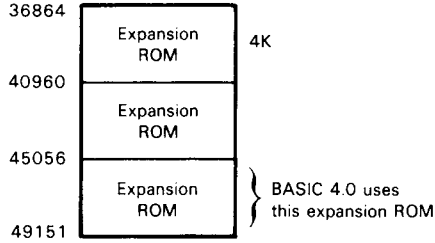
Addresses 32768-36863: 4K Video RAM

The first thousand locations of this block, from addresses 32768 through 33767, are allocated to screen memory. A POKE to any of these locations displays the character in the appropriate screen position.



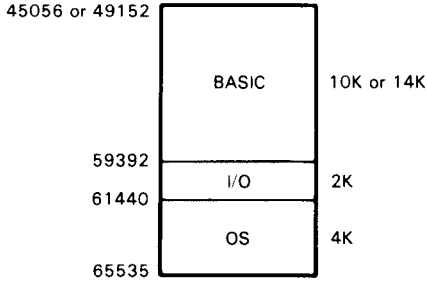
Addresses 36864-49151: Expansion ROM 12K

Memory addresses 36864 through 49151 are allocated for optional expansion of ROM to 26K.



Addresses 49152-65535: 14K ROM and 2K I/O

Locations 49152 (45056 for BASIC 4.0) through 59391 and locations 61440 through 65535 hold the BASIC interpreter and OS diagnostics. Memory-mapped I/O locations are from 59392 through 61439.



Location 65535 is the end of CBM memory.

MEMORY MAP

Detailed memory maps used by different versions of CBM BASIC are shown in Appendix F. Table F-1 describes the Revision Level 2 ROMs used in the original PET computers. Table F-2 shows the Revision Level 3 ROMs used in BASIC<3.0. Table F-3 shows the most recent memory map for BASIC 4.0.

Tables F-1 and F-2 show the memory address in decimal and hexadecimal. You should use the decimal value as the PEEK or POKE address. Tables F-1 and F-2 also show sample decimal and hexadecimal equivalent values in memory locations.

With the exception of pointers, these sample values are typical of what you might see if you PEEKed at the location; these are all byte values, in the range 0 to 255 ($0\text{-}FF_{16}$). **A pointer is a two-byte address, in the range 0 to 65535 ($0FFFF_{16}$), that is stored in the CBM in low-byte, high-byte order.** All two-byte locations in the table contain values stored in low-high order. Consider the first such location in the table:

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
1-2	0001-0002	826	033A	User address jump vector

If you PEEKed at these locations, the 16-bit address would be presented in two parts, first the low-order byte:

```
?PEEK(1)
58
```

and then the high-order byte:

```
?PEEK(2)
3
```

To convert the two values to the appropriate address, you can convert them separately to hexadecimal and then convert the hexadecimal address to decimal:

Low	High		Address
$58_{10}=3A_{16}$	$3_{10}=03_{16}$	⟶	$033A_{16}=826_{10}$

Note carefully that the sample value 033A means that the first memory byte =3A and the second (higher) memory byte = 03.

Or you can multiply the high-order byte by 256 and add it to the low-order byte. The following is a PEEK statement that will do this for you:

```
?PEEK(1)+PEEK(2)
826
```

Conversely, to convert a 16-bit memory address into two separate bytes for POKEing (in low-byte, high-byte order), you can convert the decimal value to hexadecimal and then convert the separated byte digit pairs to decimal, e.g., to convert the address 59409:

	High		Low
$59409_{10}=E811_{16}$ —	$E8_{16}=232_{10}$	and	$11_{16}=17_{10}$

Or you can convert using decimal arithmetic by first dividing the address value by 256 and discarding any fractional remainder:

```
                                High
59409/256=232.06641=232
```

Then subtract the high value multiplied by 256 from the original value (59409 in this case) to get the remainder, which is the low-order byte value:

```
232*256=59392
                    Low
59409 - 59392=17
```

(Of course, if you do the division by longhand, the remainder is directly available.)

For a block of byte locations, only the first byte value is shown in the table.

The column labeled DESCRIPTION in Table F-1 gives a short description of the location's use. There are multiple uses for some locations, in which case the primary one is indicated. While not exhaustive, the table illustrates the overall makeup of the CBM memory.

Table F-3 compares the BASIC 4.0 memory map with the BASIC 3.0 revision shown in Table F-2. The DESCRIPTION column provides the location description as currently used by Commodore; the label column shows the assembly language label currently assigned to the location by Commodore. The BASIC 4.0 column gives the hexadecimal address of each location, while the BASIC 3.0 column gives the equivalent BASIC 3.0 hexadecimal address. To find any BASIC 4.0 location, first find the hexadecimal address given in Table F-2. Find this hexadecimal address in the BASIC 3.0 column of Table F-3 and the comparable BASIC 4.0 hexadecimal address is in the adjacent column.

With the exception of the first two entries in Table F-3 which actually represent memory address 0000, all subsequent 0000 addresses identify entries which do not exist in one version of BASIC or the other. For example, if you see an address in the BASIC 3.0 column with 0000 in the BASIC 4.0 column, then BASIC 4.0 has no equivalent location in its memory map. Conversely, a 0000 address in the BASIC 3.0 column identifies a new entry in the BASIC 4.0 memory map for which there is no BASIC 3.0 equivalent.

CBM BASIC INTERPRETER

The CBM BASIC interpreter executes a user program by decoding each source line. Source lines are stored in memory in a compacted form. When you enter a line from the keyboard, the Line Editor has control, allowing you to edit the line until you press the RETURN key. Program lines are stored in memory in ascending line number order. When the RETURN key is pressed, the BASIC interpreter searches memory for the same line number. If there is one, it replaces the current line with the new line. If there isn't one, it searches for the next higher line number. The BASIC interpreter then inserts the new line into memory and moves the reset of the program up.

Program lines are stored at the beginning of the user program area of memory, which starts at memory location 1024. Variables are stored in memory above the program lines, and arrays are stored above the variables. All three areas begin at lower addresses and build upwards to higher addresses. Strings are stored beginning at the top of memory and work downwards. The BASIC interpreter builds all four areas, moving them as necessary and adjusting pointers for insertions and deletions. Eight pairs of

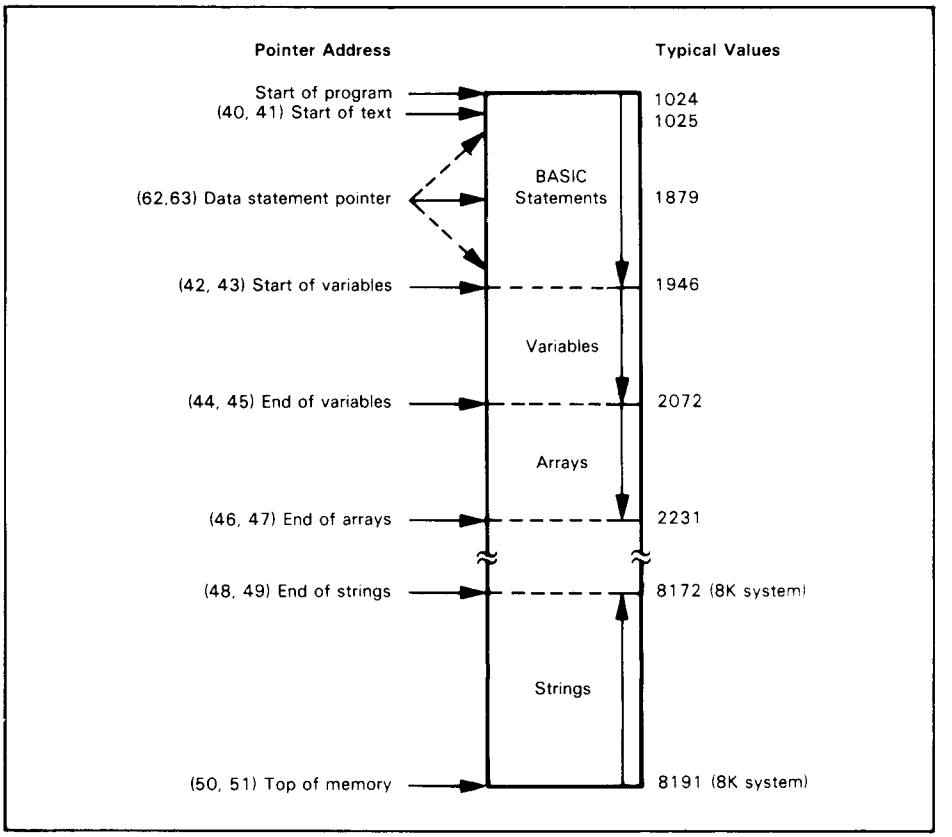


Figure 7-2. Principal Pointers In User Program Area

memory locations contain pointers to the division points in the user program area of memory. These are shown in Figure 7-2. (They are also listed in Appendix F tables).

The formats in which BASIC statements, variables, arrays, and strings are stored in their respective areas are discussed next.

BASIC STATEMENT STORAGE

BASIC statements are stored in the format shown in Figure 7-3.

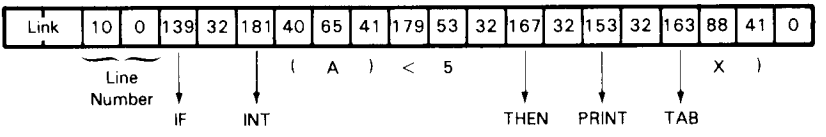
Memory location 1024 always contains a zero byte.

The next two bytes contain a pointer to the beginning of the first BASIC statement. The pointer, like all other addresses, is stored in low-byte, high-byte order. The pointer is a link to the memory address of the next link. **A link address of zero denotes the end of the text;** i.e., there are no more links and no more statements. BASIC statements are stored in order of ascending line numbers, even though there are links to the next statements. Links are used to quickly search through line numbers.

Following the link address is the line number of the statement, stored in low-byte, high-byte order. Line numbers go from 1 (stored as 1 and 0) to 63999 (stored as 255 and 249).

After the line number, the BASIC statement text begins. Keywords consist of reserved words (listed in Table 4-4) and operators (listed in Table 4-2). Reserved words and logical operator keywords are stored in a compressed format. A one-byte token is used to represent a keyword. All keywords are encoded such that the high-order bit is set to 1. Other elements of the BASIC text are represented by their stored ASCII code; these elements include constants, variable and array names, and special symbols other than operators. All are coded just as they appear in the original BASIC statement. Table A-1 shows the byte codes for all values from 0 to 255 that may appear in the compressed BASIC text. Codes are interpreted according to this table except after an odd number of double quotation marks enclosing a character string; within a character string the standard ASCII codes prevail, as shown in Table A-4.

Note that the left parenthesis is stored as part of the one-byte token for the functions TAB and SPC, but that the other functions use a separate byte for this symbol. For example, the following line would be coded as bytes (in decimal) as illustrated below.



The operators (the symbols +,−,*,/,<,=,> and the words AND, OR, and NOT) are given keyword codes (high-order bit set) since they "drive" the BASIC interpreter just as reserved words do (e.g., 179 for <). The standard ASCII codes for these symbols (e.g., 60 for <) appear only in the text of a string.

Spaces in the source line are stored except for the space between the line number and first keyword. This space is supplied on LISTing when a stored statement is expanded to its original form. **You can conserve memory storage space by eliminating blanks (but this makes the program harder to read). You can also conserve space by putting more than one statement on a line, since the five bytes of link, line number, and 0-end-byte are stored only once.**

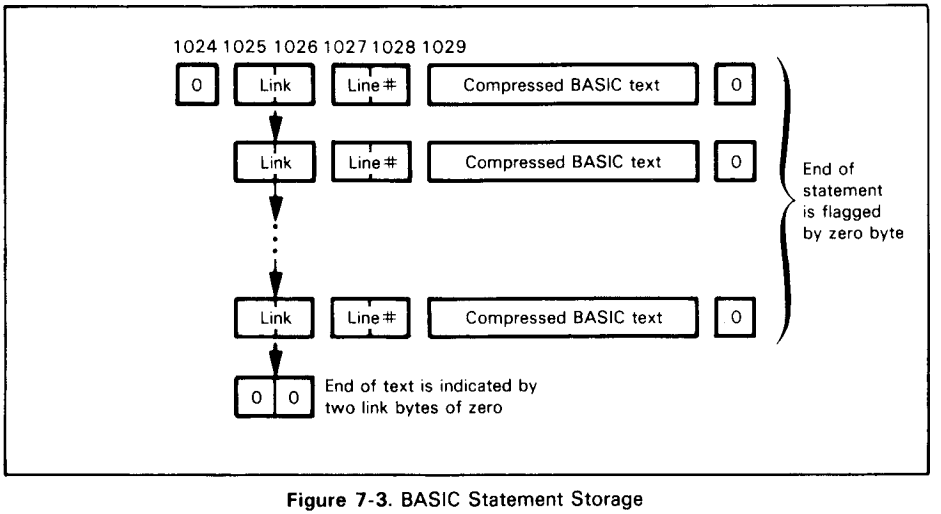


Figure 7-3. BASIC Statement Storage

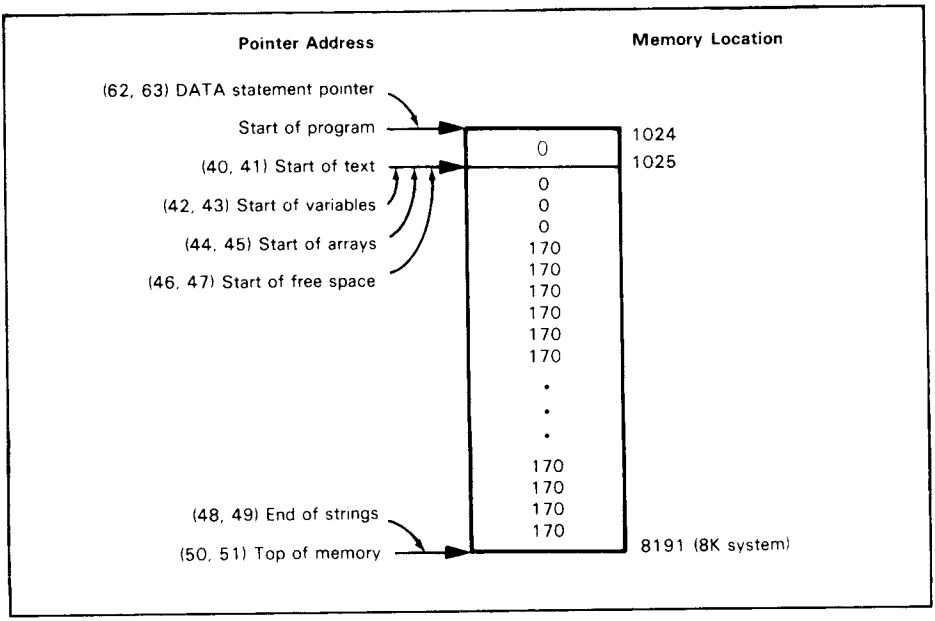


Figure 7-4. User Program Area on Power-Up

The size of each statement is variable and is terminated by a byte of zero to indicate the end of the statement. (A value of zero anywhere within the text is stored as 48.) 0-byte flags are used by the BASIC interpreter in executing a program when it goes through the compressed BASIC text from left to right picking out keywords and performing the indicated operations. A 0-byte indicates the end of the statement; the next four bytes are the link and the line number of the next statement. In contrast to searching through the text and using 0-byte indicators to locate the next statement, links are used when searching the statements for their line numbers. Three consecutive bytes of zero (the last statement's 0-byte followed by two zero link bytes) flag the end of text when executing the program.

A program is stored onto cassette tape in the same format as for memory storage (Figure 7-3). Thus, it is basically "dumped" onto tape in a continuous block, including link addresses and 0-end-bytes.

The use of tokens in place of keywords is not unique to the CBM BASIC, but there is no standard coding from one interpreter to another. Thus, **a BASIC source program SAVEd on tape by CBM BASIC is not compatible with other BASICs, nor can BASIC programs generated on other (non-CBM) machines normally be loaded by the CBM BASIC interpreter.**

USER PROGRAM AREA INITIALIZATION

On power-up, the user program area of memory is initialized to "+" characters (code 170) except for the first few beginning locations 1024 to 1026. Location 1024 is zero, the initial link in locations 1025 and 1026 is also zero. The pointers into the user area are initialized as shown in Figure 7-4.

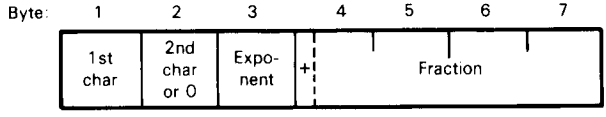
As lines are entered and edited and new programs loaded, the contents of memory locations throughout the user program area change. They change, however, only as necessary for the current program. The user area is not continuously reinitialized (to "+" or any other code). It is the pointers into the user area that determine the extent of the current program, if any. The action of a NEW statement is simply to readjust the pointers to the initial values shown in Figure 7-4. A CLR does the same thing except that it adjusts the variable and array pointers from the end of the program rather than the start of the program as NEW does. In fact, **if you have accidentally cleared the program or variables, you can reinstate them by "reading" through the user program area as needed and restoring the pointer values.**

DATA FORMATS

Variables

Variables are stored in the Variable Area of user program memory (see Figure 7-2). These are simple (unsubscripted) variables; arrays are stored in a separate area. The variables may be floating point, integer, or string and are freely intermixed in the Variable Area. **Each variable, regardless of its type, occupies seven bytes of memory.** The first two bytes contain the variable name, and the remaining five bytes further define the variable. Variables are entered into the variable table as they are encountered during execution of the user program. A variable that is not in the table is assumed to have a value of zero for numeric variables or null for a string variable.

Floating Point Variable Format

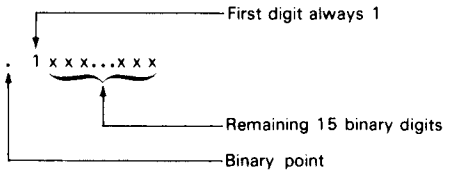


Byte 1 contains the first character of the variable name. Byte 2 contains the second character of the variable name or, if there is no second character, byte 2 contains a zero. The characters are stored in standard ASCII codes (see Appendix A). For example, the name A is stored as 65, 0 whereas the name A0 is stored as 65, 48. **A floating point variable is denoted by variable names having stored ASCII values of 90 or below.**

Bytes 3 through 7 contain the value of the floating point variable. **Byte 3 contains the exponent in excess 128 format.** The exponent determines the magnitude of the number. In excess 128 format, 128 is added to the true exponent (after normalization of the significant digits) so that the smallest exponent representation contains all zeros. The largest exponent representable contains all ones. A true exponent of zero is represented by an exponent value of 128 (0+128). Excess 128 format eliminates having to consider a sign in the exponent. Here are some examples:

Actual Exponent	Stored Exponent	Approximate Value
127	255	10^{38} (maximum exponent)
34	162	10^{10}
−1	127	10^{-1}
−126	2	10^{-38}
−128	0	10^{-39} (minimum exponent – number is zero)

Bytes 4 through 7 contain the significant digits of the number. The number is normalized such that the binary point is to the immediate left of the first non-zero binary digit. That is, it is represented as a fraction in the form:



The binary point is always assumed and is not stored. Further, **the most significant 1 digit is always assumed** (since it is always 1) **and** is not stored either. **Its bit position is used to hold the sign of the number,** 0=positive and 1=negative. To normalize a number, the point is moved to the left and the exponent decremented (smaller numbers), or the point is moved to the right and the exponent incremented (larger numbers), until the number is a fraction in the form shown above. **The number zero is** generally **represented by** all zeros in bytes 3 through 7, but the fraction may contain roundoff errors; **an exponent of zero** is sufficient to make the number zero.

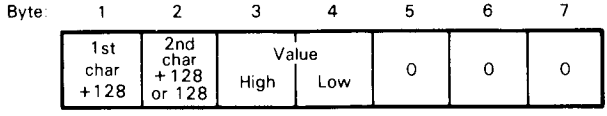
Some examples of floating point number representations stored in the Variable Area follow. 1E+38 has the maximum exponent of 255. This decreases down to zero as the numbers decrease to zero. Fractional floating point numbers (e.g., 5, .01, .006) have exponents below 129. For negative numbers, the exponent increases from 0 to 255 as the absolute value of the numbers increases. In byte 4 the high-order bit is the sign bit. In this column, decimal numbers less than 127 have bit 7=0 (positive numbers), and decimal numbers higher than this have bit 7=1 (negative numbers).

Byte:	3	4	5	6	7
Number	Exponent	±MSB		Fraction	LSB
1E+38	255	22	118	153	83
1E+10	162	21	2	249	0
1000	138	122	0	0	0
1	129	0	0	00	
0.01	122	35	215	10	62
1E-4	115	81	183	23	90
1E	62	60	229	8	101
1E-39	0	32	0	0	0
0	0	0	0	0	0
-1	129	128	0	0	0
-1000	138	250	0	0	0
-1E+10	162	149	2	249	0
-1E+38	255	150	118	153	83

The following short program allows you to examine floating point representations for any numbers. Line 10 inputs a number that you enter from the keyboard, terminating with a RETURN key. Line 20 points to the beginning of variables +2 to go past the two-byte variable name. Line 30 prints the number that was input, followed by the five bytes PEEKed from the variable table. The program is continuous; to end, enter a null line (RETURN key only).

```
10 INPUT A
20 X=PEEK(43)*256+PEEK(42)+2
30 PRINT A;"="PEEK(X);PEEK(X+1);PEEK(X+2);PEEK(X+3)PEEK(X+4)
40 GOTO 10
```

Integer Variable Format

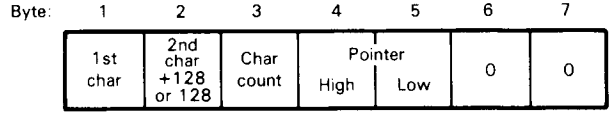


Byte 1 contains the first character of the variable name shifted (+128). Byte 2 contains the second character of the variable name shifted (+128), or if there is no second character, byte 2 contains 128. An integer variable is denoted by variable names having ASCII values of 176 or higher. The % notation is dropped from the variable name. Bytes 3 and 4 contain the value of the integer in high-byte, low-byte order. (Note that this value is not an address and does not conform to the reverse standard for pointers). **The value is stored in twos complement format** so that the high-order bit (bit 7 of byte 3) represents the sign, 0=positive, and 1=negative. **The remaining three bytes are not used and are set to zero.**

The following are some examples of integer representations stored in the Variable Area. You can use the same program as above to look at integer number representations after changing A to A% in lines 10 and 30.

Byte Number	3	4
32767	127	255 (256•127+255=32767)
32766	127	254
14000	54	176
256	1	0
255	0	255
1	0	1
-1	255	255 (FFF_{16})+1=1
-2	255	254
-32766	128	2
-32767	1281	1

String Variable Format



Byte 1 contains the first character of the variable name. Byte 2 contains the second character of the variable name shifted (+128), or if there is no second character, the second byte contains 128. This combination of ASCII ranges denotes a string variable entry. The $ notation is dropped from the variable name. **Byte 3 contains a count of the number of characters in the string (1 to 255). This is the value fetched for the LEN function. Bytes 4 and 5 contain a pointer to the beginning of the string itself, stored elsewhere in memory. This pointer is in the standard 6502 low-byte, high-byte order. The remaining two bytes are not used and are set to zero.**

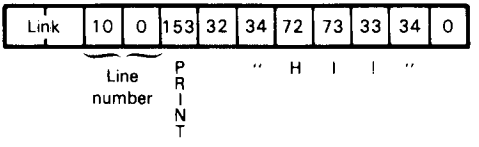
String storage is optimized by using the copy of the string already in memory if there is one. If there is not, a string is created and stored in the String Area in upper memory. A few examples are given below.

Constants

Constants are stored in the BASIC statement itself. They are not placed into a separate area of memory, and they are not stored in the Variable Area. **Floating point, integer and string constants are all stored as ASCII character source codes,** as described previously under BASIC Statement Storage. For example, the line:

```
10 PRINT "HI!"
```

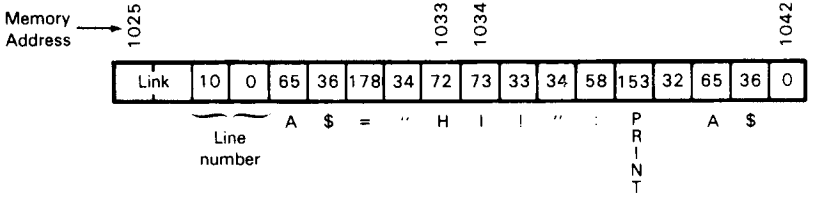
is stored **entirely in the BASIC Statement Area,** in the form:



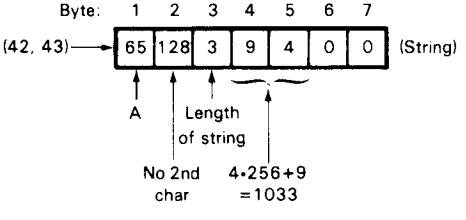
whereas the statement

```
10 A$="HI!":PRINT A$
```

is stored in two areas. The original statement is stored in the BASIC Statement Area:



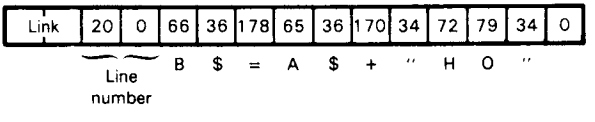
The illustrated memory addresses assume that this is the first statement in program memory, therefore it is stored at the beginning of the user program area (location 1025). In addition, when this statement is executed the following entry is made in the Variable Area:



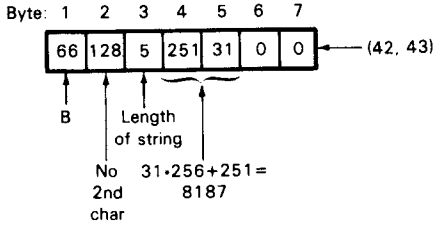
The string in the BASIC Statement Area is pointed to (beginning at memory location 1033 in this program) rather than storing a copy of it in upper memory. However, when you create a new string, as in:

```
20 B$=A$+"HO"
```

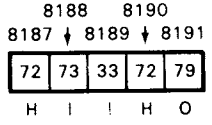
the BASIC Program Area entry is:



and the entry in the Variable Area is:



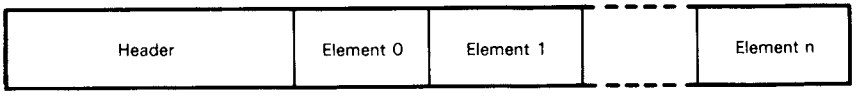
This time the pointer addresses a location in upper memory (8187 in this program) that contains the string:



The address 8187 assumes an 8K memory. The largest available address is then 8191.

ARRAY STORAGE FORMAT

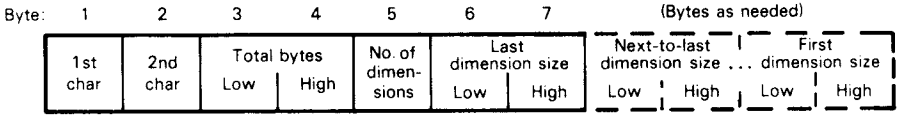
Arrays are stored in the Array Area of user program memory (see Figure 7-2). Arrays may be floating point, integer, or string, and are stored in the order in which they are created by the program. The type of array is distinguished by the way in which the two-character array name is stored. Array names and variable names are encoded in exactly the same way. **An array is stored with a header, followed by the elements of the array**, as follows:



Elements are stored in reverse order for strings.

Array Header

All types of arrays have the same header format. The header contains seven bytes, plus two additional bytes for each array dimension beyond 1.

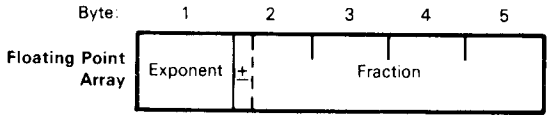


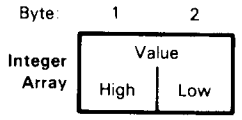
Floating point array elements are encoded using floating point variable format, therefore each floating point array element occupies five bytes. But array integers require just four bytes, while array strings require five bytes; in each case the zero bytes are discarded.

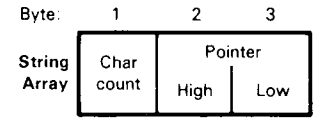
In the array header, bytes 1 and 2 contain the array name. Bytes 3 and 4 contain a count of the number of memory locations that the array occupies. For example,

A(0) would occupy 12 bytes: 7 for the header and 5 for the single element. The byte count is **stored in low-byte, high-byte order. Byte 5 contains a count of the number of dimensions in the array**. Thus, A(5) has one dimension (byte 5=1) and A(10,10,2) has three dimensions (byte 5=3). **For a one-dimensional array (or vector), bytes 6 and 7 contain the dimension size** — this is the number specified between parentheses in the DIM statement +1. For example, the dimension size = 61 for DIM A(60), = 101 for DIM A(100), etc. If the array does not appear in a DIM statement, the dimension size defaults to 11. The dimension size is **stored in low-byte, high-byte order. For a multiple dimension array, the header contains additional bytes in which additional dimension sizes are stored. Two additional bytes are used for each additional dimension. The dimension sizes are stored in reverse order as compared to the order in which they appear in the DIM statement.** For example, for DIM A(10,5) the dimension sizes are stored as bytes 6,7=6 and bytes 8,9=11. For DIM X(2,1,3) the dimension sizes are stored as bytes 6,7=4, bytes 8,9=2 and bytes 10,11=3.

Array element formats for each type of array are shown below. Formats are as described for variables, with bytes deleted.







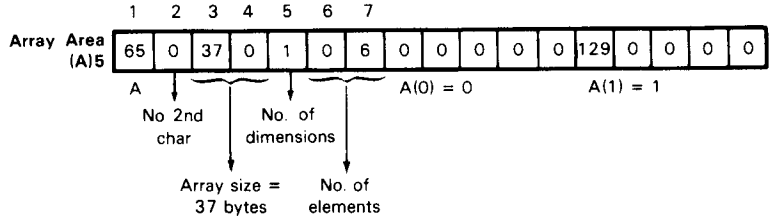
The size of the header may be calculated as five bytes plus twice the number of dimensions in the array. **Memory occupied by array elements may be calculated as the number of bytes per element (5 for floating point, 2 for integer, 3 for string) times the number of elements (the dimensions multiplied together + 1).** The total size of the array, header plus elements, is stored in byte 4 of the array header.

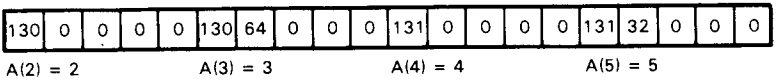
The following program examines Array Area entries:

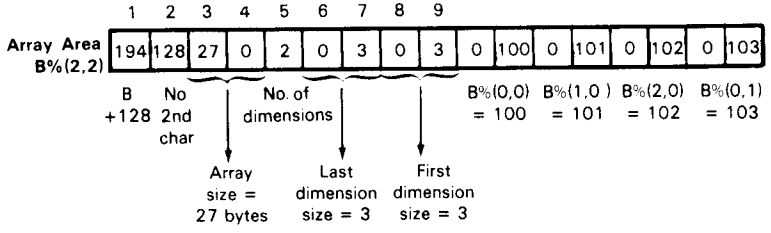
```
10 DIM A(5),B%(2,2),C$(10): REM SAMPLE ARRAYS
20 FOR I=0 TO 5: A(I)=I: NEXT I
30 FOR I=0 TO 2: FOR J=0 TO 2: B%(J,I)=100+3*I+J: NEXT J,I
40 FOR I=0 TO 10: C$(I)=CHR$(ASC("A")+I): NEXT I
50 X=PEEK(45)*256+PEEK(44): REM POINT TO ARRAY AREA
60 Y=PEEK(47)*256+PEEK(46): REM END OF ARRAYS
70 FOR I=X TO Y
80 PRINT I,PEEK(I)
90 GET D$: IF D$="" GOTO 90
100 NEXT I
```

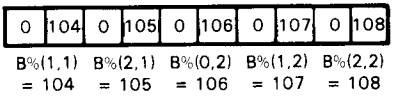
Each of the three types of array is dimensioned. Line 20 fills the floating point array A with the numbers 0 through 5. Line 30 fills the integer array C$ with the single strings A through K. Lines 50 and 60 fetch the pointers to the end of the Variable Area and the

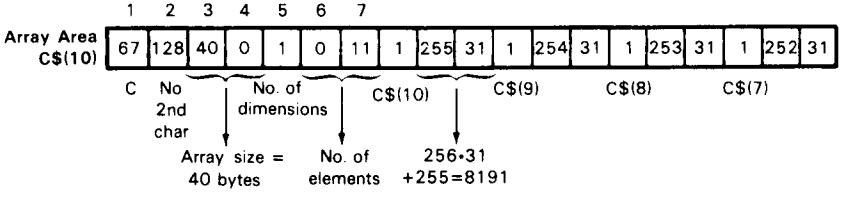
end of the Array Area. The display stops at each memory location; to print the next location, press any key (e.g., the RETURN key). You will need to locate the beginning of the arrays by the sequence for the first array shown below (the pointer addresses the end variable). **The memory locations will appear as shown below.**

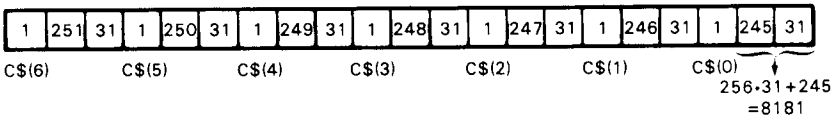


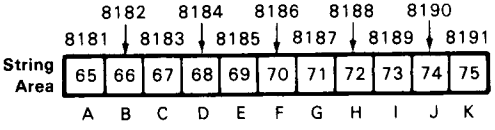






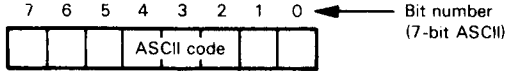






CHARACTER REPRESENTATION

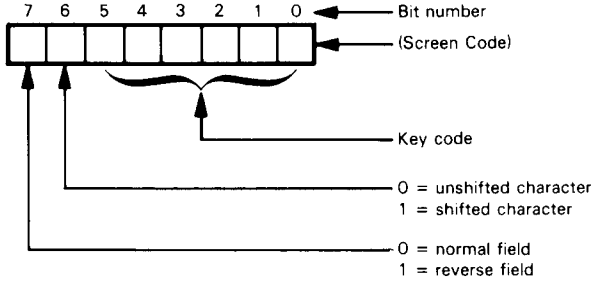
ASCII (American Standard Code for Information Interchange) is a widely used code for representing character data. It is normally a 7-bit code, allowing 128 characters ($7F_{16}=128_{10}$) to be represented. The standard ASCII 7-bit character set is shown in Table A-2 in Appendix A. Bits are numbered from 0 (least significant bit) to 6 (most significant bit):



The first 32 codes are reserved for non-printable control characters, intended for message formatting and print format control.

CBM computers store characters in an extended, 8-bit version of ASCII format. With eight bits normally available, rather than just seven, up to 256 characters can be represented. Within compressed BASIC text, the 8-bit character codes are interpreted as shown in Table A-1, where bit 8=1 signifies a keyword. Elsewhere in main memory the 8-bit character codes are interpreted as shown in Table A-4.

The screen memory, occupying memory locations 32768 through 33767, uses a different ASCII character representation than main memory. It is a 7-bit code as shown in Table A-3. The eighth bit is a normal/reverse field indicator. Note that the characters are arranged such that bits 0 through 5 represent one key on the PET keyboard, with bit 6=0 being the unshifted character and bit 6=1 being the shifted character of the same key.



The complete character set for screen memory is shown in Table A- 4 under the PEEK/POKE column.

The screen memory ASCII code may be derived from the CBM ASCII code by moving bit 7 of the main code into bit 6 and dropping the previous value of bit 6. The examples below illustrate the four cases of a 0 or 1 in bit 7 going into a 0 or 1 in bit 6:

Character	Main Memory Representation	Screen Memory Representation
	01000001	00000001
Shifted A (♠)	11000001	01000001
1	00110001	00110001
Shifted 1 (▒)	10110001	01110001

When PRINTing to the screen, the CBM computer automatically makes the conversion to screen codes. Only when you are PEEKing and POKEing in screen memory do you need to be concerned with character set differences.

Screen memory can be looked upon as having an additional bit that selects the alternate character set in response to a POKE 59468,14. POKE 59468,12 restores the standard set. The alternate set is also shown in Table A-4.

ASSEMBLY LANGUAGE PROGRAMMING

CBM BASIC can execute small programs written in 6502 assembly language. Assembly language programs execute faster and require less memory space for a given function than the equivalent BASIC program. **You might want to write an assembly language program to be run on the CBM computer if:**

1. The operation is not fast enough using a BASIC program.
2. The operation cannot be implemented in CBM BASIC.
3. The operation takes up too much memory space as a BASIC program.
4. Assembly language lends itself better to the task than the BASIC language. Some I/O operations probably fall into this category.

An assembly language program can be loaded into memory by POKEing the decimal values of the 6502 instructions that make up the program. There is no area set aside for use by assembly language programs. You have to make space, either by taking otherwise unused locations or by setting up a space in the user program area of memory. The following are possible locations:

1. **Cassette Buffers.** If you do not have a second cassette unit, then the 192-byte tape buffer for cassette #2 can be used to store an assembly language program. The buffer #2 extents are locations 826 to 1017 (see Appendix F). In addition, if the console cassette unit is not going to be used while the assembly language program is operating, then the other 192-byte tape buffer for cassette #1, at memory locations 634-825, is also available. No LOADs, SAVEs or other tape I/O can be performed accessing the particular cassette while its buffer is used by an assembly language program.
2. **Top of Memory.** Memory locations 52 and 53 contain the pointer to the top of memory. On 8K PETs this value is 8192. You can temporarily set the top-of-memory pointer to a lower address, thereby reserving a number of bytes from the new pointer value to the actual top of memory for storage of an assembly language program. To set the pointer, say, down 1000 bytes, you will need to store the value 7192 (8192−1000) converted into low address, high address order:

$$7192_{10}=1C18_{16} \rightarrow \begin{matrix}\text{High} \\ 1C_{16}=28_{10}\end{matrix} \text{ and } \begin{matrix}\text{Low} \\ 18_{16}=24_{10}\end{matrix}$$

So 24 is to be stored at location 52 (low byte), and 28 is to be stored at location 53 (high byte). The following instructions can be used:

```
10 AL=PEEK(52):AH=PEEK(53):REM SAVE CURRENT POINTER
20 POKE 52,24:POKE 53,28:REM TOP OF CORE = 7192
.
100 POKE 52,AL:POKE 53,AH:REM RESTORE POINTER
110 END
```

3. You may find usable locations in the **BASIC Statement Area**. You may create a block of dummy DATA statements and use those locations. There are generally a few locations free between the end of the program and the beginning of the Variable Area. But you must be very careful that your assembly language program and the BASIC interpreter do not get in each other's way.

The CBM BASIC interpreter can be used to load an assembly language program into the selected area of memory. The process is a rudimentary one, consisting of POKEing the decimal equivalents of the 6502 machine language instructions. To get the instructions in decimal, write your program in 6502 assembly language (reference manuals are listed in Appendix D), hand assemble it into hexadecimal, and then convert the hexadecimal codes to decimal. Commodore's Terminal Interface Monitor stores the hexadecimal codes directly. However, with the Monitor you must load the assembly language routine separately from the BASIC program, whereas by POKEing you can load the assembly language routine as part of executing the main program written in BASIC. DATA statements are used to define the machine language codes, which can be subsequently READ into the program and passed to a POKE loop.

Control is transferred to an assembly language program in one of two ways: the SYS or the USR function, which are more or less interchangeable. SYS is geared to turning control over to an assembly language program. USR is a true function reference that allows a value to be sent to the called assembly language routine and a value returned by it to the main program.

The assembly language program must return control to BASIC via a Return-from-Subroutine (RTS) assembly language instruction.

SYS

SYS is a system function that transfers program control to an independent subsystem.

Format:

SYS(address)

where:

address — is a numeric constant, variable, or expression representing the starting address at which execution of the subsystem is to begin. The value must be in the range $0 \leq \text{address} \leq 65535$.

Unlike other functions, SYS can be specified alone in a direct or program statement.

Example:

```
SYS(826)
```
In immediate mode transfer control of the system to the 6402 assembly language program beginning at memory location 826 (the 2nd cassette buffer)

```
55 SYS(826)
```
Same as above but executed in program mode. On return, execution proceeds with the first statement following the SYS statement

```
126 SYS(A+14)
```
Transfer control of the system to the computer address A+14

SYS is the assembly language subroutine equivalent of GOSUB, but with the important difference that the safeguards built in to CBM BASIC to protect the system from user program errors are no longer operable. The system will tend to crash even

more frequently while debugging assembly language programs than it does debugging BASIC programs.

Use the RTS assembly language instruction to return to BASIC.

Values can be passed between the BASIC program and the SYS subroutine using PEEKs and POKEs.

USR

USR is a system function that passes a parameter to a user-written assembly language subroutine whose address is contained in memory locations 1 and 2 and fetches a return parameter from the subroutine.

Format:

USR(datan)

where:

datan is the numeric parameter value passed to the subroutine

Example:

```
?USR(60)
```
Displays in immediate mode the value returned by the USR subroutine when passed a value of 60

```
105 A=USR(60)
```
Same as above but in program mode

```
210 IF USR(X)<4 GOTO 50

510 SM=USR(XA)+USR(3.4)+SQR(Y)+π
```

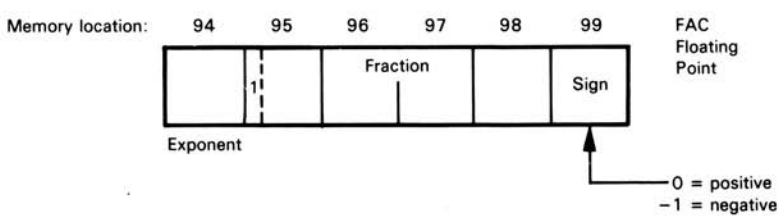
Before making a USR reference, the beginning address of the assembly language subroutine must be placed into memory locations 1 and 2. For example, if the subroutine is located in the cassette #2 area, you would include the instructions:

```
10 POKE 1,58
20 POKE 2,3
```

Low High

$826_{10}=033A_{16}=3A_{16}=58_{10}$ and $03_{16}=3_{10}$

The parameter value is passed to the USR subroutine in system locations that function as a floating point accumulator (FAC) for all functions. The FAC resides in six bytes, from memory locations 94 to 99 ($5E_{16}$-63_{16}). The FAC has the following format:



Like floating variables, the exponent is stored in excess 128 format and the fraction is normalized with the high-order bit of byte 95 (the high-order byte of the fraction) set to 1. The difference between this format and the variable format is that the high-order 1 bit is present in byte 95 of the FAC. An extra byte (99) is used to hold the sign of the fraction. (This is done for ease of manipulation by the functions that use the FAC.)

The USR subroutine must fetch the value passed to it from the FAC locations. It must deposit the value being returned into the FAC before terminating. If the USR subroutine does not alter the FAC, then the same value is returned to the program as was passed from it.

RANDOM ACCESS FILES

Random access files are created by directly addressing diskette data blocks and memory buffers.

Diskette data blocks each occupy a single sector. Random access files directly address diskette data blocks via their actual track and sector address. Diskette memory buffers, likewise, are directly addressed and assigned to logical file secondary addresses. (Recall that each diskette unit has sixteen 256-byte memory buffers.)

Random access files are created by using a number of subroutines that directly access the diskette surface and memory buffers. These are the same subroutines used to implement sequential and relative file logic; however, your program creates the field/record/file structure, whatever it may be.

You should not use random access files unless you are a very experienced programmer. You will be working at the same level as the people who designed the sequential and relative file logic found in standard CBM BASIC. These individuals are professional system programmers. Unless you are an equally experienced programming professional, you are unlikely to have much success with the information presented in this section.

Diskette random access is programmed using PRINT# statements with appropriately coded text strings in their parameter list. The PRINT# statements access the command channel, via secondary address 15. Random access logical files are opened with specific diskette memory buffers assigned to each logical file via its secondary address. The PRINT# statement parameter list uses the secondary address to identify logical files and assigned buffers.

The following standard OPEN statement format is used when opening a random access logical file:

```
100 OPEN lf,dev,sa,"#[bu]"
```

where:

lf	is the logical file number specified in the command channel OPEN statement
dev	is the device number (usually 8)
sa	is the secondary address, which should have a value between 2 and 14
bu,	if present, is the buffer number allocated to the specified secondary address. There are sixten 256-byte buffers; the first three buffers are used by the disk operating system. Buffer numbers 3 through 15 are therefore available. If bu is not specified, then the next available buffer is assigned to the secondary address

You can execute a GET# statement immediately after opening a random access file in order to determine the assigned buffer number. However, the GET# statement must be executed before any other input or output statement accesses the logical file. Here is an example program:

```
5 REM ASSIGN BUFFER 5 TO SECONDARY ADDRESS 4, USED BY LOGICAL FILE 2
10 OPEN 2,8,4,"#5"
20 PRINT DS$:REM CHECK I/O OPERATION STATUS
30 GET#2,A$:PRINT ASC(A$):REM DISPLAY THE BUFFER NUMBER TO CHECK OPERATION
40 PRINTDS$:REM RECHECK I/O STATUS
50 CLOSE 2
60 STOP
```

Random access file commands are subsequently issued using PRINT# statements with the following general format:

```
10 OPEN lf,8,15
20 PRINT# lf, "parameter"
```

parameter identifies the random access file operation. parameter has two parts: a command and a parameter list. The command has a long form which must end with a colon, or a short form, in which case the parameter list is assumed to begin at the fourth character position of the string. Parameters can be separated by comma, space or skip characters. The following abbreviations are used to describe parameters:

sa	The secondary address specified in the data logical file OPEN statement
dr	The diskette drive number (0 or 1)
t	The diskette track number
s	The sector number within the selected track
p	The buffer pointer, or character position selector, which may have a value between 0 and 255
adl	The low-order byte of a memory address
adh	The high-order byte of a memory address
nc	Number of characters. This number must be between 1 and 34
data	A data string with nc characters

adl, adh and nc must be specified as parameters of CHR$ functions. For example, if adl has the value 123, it must be specified as CHR$(123).

Block Read

This statment reads any diskette sector into a buffer. The BLOCK READ statement has the following format:

```
PRINT# lf, "BLOCK-READ:sa,dr,t,s"
PRINT# lf, "B-Rsa,dr,t,s"
```

The following example opens logical file 2, assigning buffer 5 to secondary address 4, then reads sector 0 of track 18 on drive 1 into buffer 5:

```
10 REM OPEN LOGICAL FILE 2, ASSIGNING BUFFER 5 TO SECONDARY ADDRESS 4
20 OPEN 2,8,4,"#5"
30 REM READ SECTOR 0 OF TRACK 18 ON DRIVE 1 INTO BUFFER 5
40 OPEN 15,8,15
50 PRINT#15,"B-R4,1,18,0"
60 REM DISPLAY THE BUFFER CONTENTS TO PROVE THAT DATA WAS FETCHED
70 REM DISPLAY 256 BYTE BUFFER AS 8 ROWS OF 32 NUMBERS PER ROW
75 PRINT"■";
80 FOR I=1 TO 8
90 FOR J=1 TO 32
100 GET#2,A$:IF A$="" THEN 100
110 PRINT ASC(A$);
120 NEXT J
130 PRINT
140 NEXT I
150 CLOSE 2
160 CLOSE 15
170 STOP
```

Block Write

This statement writes the contents of a buffer to a specified sector. It has the following format:

```
    PRINT # lf,"BLOCK-WRITE:sa,dr,t,s"
or  PRINT # lf,"B-Wsa,dr,t,s"
```

The following statements open logical file 2, assigning buffer 8 to secondary address 7. The contents of buffer 8 are written to sector 10 of track 35 on drive 0:

```
200 OPEN 2,8,7,"#8"
210 OPEN 15,8 15
220 REM STATEMENTS THAT WRITE TO BUFFER 8 MUST FOLLOW HERE
300 PRINT#15,"B-W7,0,35,0"
310 CLOSE 2
320 CLOSE 15
330 STOP
```

Block Execute

This statement is the same as a BLOCK READ, except that data read from the sector is assumed to be an assembly language program's object code. As soon as the program is loaded it is executed. The program must end with a Return-from-Subroutine instruction (RTS). It has the following format:

```
    PRINT # lf,"BLOCK-EXECUTE:sa,dr,t,s"
or  PRINT # lf,"B-Esa,dr,t,s"
```

Buffer Pointer

This statement moves the pointer from the beginning of the buffer to any character position within the buffer. It has the following format:

```
    PRINT # lf,"BUFFER-POINTER:sa,p"
or  PRINT # lf,"B-Psa,p"
```

The statement on line 55, shown below, if added to the BLOCK READ example, moves the buffer pointer to character 24:

```
55 PRINT#15,"B-P4,26"
```

Block Allocate

This statement updates the Block Availability Map (BAM) to show how the current block has been used. The block availability map is written to the diskette when the logical file is closed. If the requested block (sector) has already been allocated, the error channel identifies the next available block, while specifying a NO BLOCK error. If no blocks are available, then 00 is returned for the track and sector parameters. It has the following format:

```
    PRINT # lf,"BLOCK-ALLOCATE:dr,t,s"
or  PRINT # lf,"B-Adr,t,s"
```

Memory Write

The MEMORY WRITE statement writes data into a diskette buffer. It has the following format:

```
PRINT # lf,"M-W"adl/adh/nc/data
```

Table 7-2. Starting Address for Model 2040 and Model 8050 256-Byte Diskette Buffers

Buffer No.	Model 2040/8050	
	Hexadecimal	Decimal
0	1000	4096
1	1100	4352
2	1200	4608
3	1300	4864
4	2000	8192
5	2100	8448
6	2200	8704
7	2300	8960
8	3000	12288
9	3100	12455
10	3200	12800
11	3300	13056
12	4000	13312
13	4100	13568
14	4200	13824
15	4300	14080

Diskette memory buffer addresses are summarized in Table 7-2 for the Model 2040 and 8050 diskette drives. Note that buffer addresses are somewhat scattered.

Suppose the four data bytes 32, 0, 17 and 96 are to be written into buffer 2 of a Model 2040 diskette drive. From Table 7-2, note that this buffer starting address is 1800_{10}. Therefore the following PRINT# statement is needed:

```
100 PRINT#15,"M-W"CHR$(00)CHR$(18)CHR$(32)CHR$(0)CHR$(17)CHR$(96)
```

Memory Read

This statement allows a byte of data to be read from a diskette buffer. It has the following format:

```
PRINT# lf,"M-R"adl/adh
```

The address of the byte to be read is specified by the parameter list using CHR$ functions. The byte itself is then read using a GET# statement, via the control channel (15). Subsequently an INPUT# statement will not execute correctly until a random access statement other than a MEMORY READ, MEMORY WRITE or MEMORY EXECUTE has been executed.

For example, the following statements read a data byte from buffer address 1808:

```
100 PRINT#15,"M-R"CHR$(8)CHR$(18)
110 GET#15,A$
```

Memory Execute

This statement executes an assembly language subroutine. It has the following format:

```
PRINT# lf,"M-E"adl/adh
```

adl and adh are the decimal low- and high-order halves of the subroutine starting address in diskette buffer memory. The subroutine which gets executed must end with the following Return-from-Subroutine instruction:

```
RTS,$60
```

Table 7-3. Random Access File User Statements

User Designation	Alternate User Designation	Function
U1	UA	BLOCK-READ replacement
U2	UB	BLOCK-WRITE replacement
U3	UC	jump to $1300
U4	UD	jump to $1303
U5	UE	jump to $1306
U6	UF	jump to $D008
U7	UG	jump to $D00B
U8	UH	jump to $D00E
U9	UI	jump to $D0D5
U:	UJ	power up $E18E

User

There are ten special "user" statements. The first two substitute for BLOCK READ and BLOCK WRITE; seven are JUMP TO subroutines, while the eighth enters the power-up routine. User statements are summarized in Table 7-3. For U3 through U9 see the revision 3 memory map given in Appendix F in order to identify the routines jumped to.

For U1 and U2 use the following format:

```
PRINT # lf "Ux;sa,dr,t,s"
```

x is 1 for U1 or 2 for U2.