OSI

BASIC IN ROM

ALL ABOUT

OSI

MICROSOFT

BASIC-IN-ROM

Version 1.0 Rev.3.2

Edward H. Carlson

CONTENTS

INTRODUCTION

This book is intended for users of OSI MICROSOFT BASIC-IN-ROM, Version 1.0, Rev. 3.2. The material is presented on 2 levels. The first is pure BASIC. The complete set of commands, statements, functions and operators are listed, together with detailed explanations of their applicability and functioning. Many examples are given of their use to accomplish various results, and of pitfalls to be avoided. In addition, several other topics are treated in depth, including techniques to reduce the memory size required to store and run programs, techniques to make programs run faster, and cassette tape input and output of data from programs.

The second level looks in detail at the storage of program code and variables in RAM, as well as the pointers and flags stored in pages $ØØ, Ø1, and Ø2. Understanding this material allows exotic programs to be written to accomplish results not obtainable otherwise.

BASIC runs in two modes, the <u>immediate mode</u> and the <u>run mode</u>. Following a cold start or a warm start, the prompter OK appears on the screen to indicate that the machine is in the immediate mode and ready to accept keyboard input. To understand BASIC, we need to keep in mind 4 areas of memory containing code. They are the BASIC <u>interpreter</u> stored in ROM starting at $AØØØ, the <u>line buffer</u> stored in zero page from $13 to $59, the <u>source program</u> starting at $Ø3ØØ and the <u>variable tables</u> stored immediately after the source code. With the machine in the immediate mode, we enter a line of material from the keyboard. The entered material appears on the screen and in the line buffer. When we hit the (RETURN) key, one of two things will happen. If the line started with a line number, integer that can be stored without round off error is $256^{+3} - 1 = 16,772,215$. When large or small numbers are displayed on the screen, scientific notation is used and considerable accuracy is lost. Example: a one line program

```
1 PRINT 16772215
RUN
 1.6772EØ7
```

VARIABLE NAMES

There are two representations of each variable name that we will consider, the name you give it in the source program and the representation of that name in the variable table. They may not be the same. In the source program, names must start with a letter and may contain any number of letters, numbers and spaces. A name ending with the symbol $ is a string variable. Names must not contain BASIC reserved words such as SIN, FOR or TO. BASIC ignores all spaces in a line of program. In the variable table, the name is stored as 2 bytes of ASCII representing the first two characters of its name in the source program. If the variable in the source program is a single letter then in the table the second byte of the name is $ØØ. If the variable is a string, then $8Ø is added to the second byte of the name in the table. In these examples, remember that the ASCII code for A is $41 and for 1 is $31.

source name	in the table	table name
A	$41 ØØ	A
A$	41 8Ø	A$
A1	41 31	A1
AA	41 41	AA
A1$	41 B1	A1$
A11$	41 B1	A1$
AGOTOB	(illegal)	--
A 1 TIME	41 31	A1

Notice that no record in the table tells how long the name was in the source. All characters past the first 2 are ignored (except $ for a string). Notice the effect that truncation of the source name has in this program:

```
1 A 1 TIME$="WHO"
2 PRINT A1$
RUN
WHO
```

COMMANDS

We will divide commands into 3 groups. Editor commands are used in the immediate mode. All other commands can be used in the immediate mode or the run mode. Most have a natural use in one or the other and some will perform in a defective manner in other than their natural mode. We will list each command in its natural mode and perhaps comment on it again in the other mode.

We depart from the usual nomenclature because it is arbitrary and confusing. For example, NEW is usually called a "command" (it erases the source program) while CLEAR is called a "statement" (it erases the variable table). Similarly the two simultaneous keystrokes (CTRL/C) are called a "special character" (it causes a break in running) while STOP is called a "statement" (it causes a break in running too).

EDITOR COMMANDS

While in the immediate mode, a very simple capability is present for editing the lines of text. We will show key strokes in parentheses, e.g. (BREAK) and multiple, simultaneous key strokes will be separated with a /.

(SHIFT/O) Types a _ and "erases" the last character typed. May be repeated to "erase" several characters. (You still see them on the screen though.)

(SHIFT/P) Types an @ and "erases" the line. (You still see it on the screen).

(RETURN) Terminates the line. If the line did not start with a number, the line is interpreted in the immediate mode. If the line started with a number, the line is stored as source code.

(CTRL/O) Suppresses writing to the screen until another (CTRL/O) is typed.

132 (RETURN) A line number without a statement following it will erase the corresponding line in the source program.

IMMEDIATE MODE COMMANDS

RUN Enters run mode. Starts interpretation and execution of the source code starting at the first line (stored at $Ø3ØØ). Discards the old variable table and constructs a new one as it interprets.

RUN 31 Starts at line 31 of the source code. Discards the old variable table and makes a new one.

GO TO 31 Starts running at line 31. Keeps the old variable table.

LIST Lists the source program. May be stopped with (CTRL/C).

LIST 31 Lists line 31 only.

LIST 31-45 Lists lines 31 through 45.

LIST 31- Lists lines 31 to the end.

(CTRL/C) Interrupts execution of the source program, LISTing, LOADing or other procedure and returns to the immediate mode. (CTRL/C) may be disabled by POKE 53Ø,1 and enabled by POKE 53Ø,Ø.

CONT Continues any procedure that has been interrupted by a (CTRL/C) or a STOP, except LIST.

LOAD Accepts input from cassette tape and puts it into source memory. To exit from LOAD, hit (SPACE BAR).

NEW Deletes present program. It does not erase it from memory however. One thing it does is to load $ØØ into addresses $Ø3Ø1 and Ø3Ø2. This makes a termination signal for the program at a point where there are zero lines in the program. If you wish to recover the program, look up the address for the second line of the program and put it into $Ø301 and Ø3Ø2 in the format described later. This is not enough of a fix to be able to RUN the program, but you will be able to SAVE, LIST it to tape, then restart the machine and read the tape back in.

SAVE This slows down the cycle of displaying input to the screen so that when followed by LIST, the speed is appropriate for writing to tape. The information is also sent to the tape port. Exit from the SAVE mode by doing LOAD, (RETURN), (SPACE BAR). The procedure for saving basic programs to tape is: SAVE, (RETURN), LIST, start tape, and wait for a few seconds to give a leader, then (RETURN).

NULL Used to insert nulls at the start of lines of output to tape. Example: NULL 5. The number of nulls inserted can vary from Ø to 8.

RUN MODE COMMANDS

LET...=... The replacement command. LET is optional, and in fact, is not often used. Examples:

```
LET A=2
AB$="COAL"
```

REM... This statement allows comments to be included in the source program. These statements are ignored during running. Examples:

```
1Ø REM PROGRAM ITCH
2Ø A=2:REM A IS THE NUMBER OF BITES
```

Wrong:

```
2Ø A=2 REM A IS THE NUMBER OF BITES:B=3
```

Unlike some compilers, BASIC doesn't pack repeated characters into compact form. Every character takes one byte in memory. These two statements take the same space in source memory:

```
1 REM 123456789
2 REM        AA
```

There are quite a few commands that change the order of execution of statements in the program. These follow.

GO TO ... Example:

```
GO TO 15
```

Not allowed:

GO TO N

In fact, such variable addresses are not allowed in any of the other flow diverting commands below.

GO SUB   Subroutine calling command. Example:

```
5 A=2
7 GO SUB 13
8 B=3
1Ø END
13 A=A+1
15 RETURN
```

The statements are executed in the order 5,7,13, 15,8,1Ø.

ON... GO TO... Example:

```
5 ON M GO TO 1Ø, 2Ø, 3Ø
```

The flow is: if M=Ø go to next statement after 5
M=1 go to statement 1Ø
M=2 go to statement 2Ø
M=3 or more, go to statement 3Ø

There is no limit (except line length) to the number of addresses after the GO TO.

ON...GOSUB... Example:

```
5 ON Z GOSUB 1Ø,12, 15, 3
```

If Z=Ø or Z greater than 4 go to the next statement. If Z= 1,2,3,4 GOSUB 1Ø,12,15,3 respectively. Upon RETURN, go to the next statement. Note the difference in flow from the ON...GOTO... statement.

IF...GOTO...   Example:

```
5 Z=3
1Ø IF A=2 GOTO 1ØØ
15 Y=3
```

If A=2 then the next statement executed is line 1ØØ. If A≠2 then the next statement after the IF...GOTO... (here 15) is executed. In place of "A=2" there can be any expression that evaluates to a Boolean "true" or "false".

Examples:

```
IF A$="DA" GO TO 338
IF (INT(X) AND 12)=8 GOTO 4
IF 3*X > PEEK(Q) GOTO 66
```

(IF...GOSUB...)   Doesn't exist, use IF...THEN GOSUB... instead.

IF...THEN...   If the expression after IF is true, then all the statements after THEN are executed. If not, then the next line is executed. Example:

```
IF X > 7.8 THEN X=7.8:GOSUB 1Ø:GOTO 3Ø
```

FOR...=...TO...
...
NEXT...   Loops. There are several subtle points that are important for trouble free use of loops, so this discussion will be quite long. Example:

```
2Ø FOR I=1 TO 3
3Ø PRINT I
4Ø NEXT I
5Ø PRINT "I IS NOW ";I
RUN
 1
 2
 3
I IS NOW  4
```

The loop is always run at least once since the test for exit occurs at the NEXT statement, after the loop variable has been incremented.

Example:

```
20 I=1 TO 0
30 ?I
40 NEXT
50 ?"I IS NOW";I
RUN
 1
I IS NOW 2
```

Upon entering the FOR... statement from outside the loop, the initial value of the loop variable is calculated, then the value which determines the exit condition is calculated. The increment size is also determined (see STEP below). These values will not change during the rest of the time spent in the loop. The statements in the body of the loop will be repeatedly executed but the FOR... statement will not be again interpreted.

```
10 A=0.6
20 FOR I=2*A TO 3*I
30 ?I
40 NEXT
RUN
 1.2
 2.2
 3.2
```

In the body of the loop, the loop variable may be redefined:

```
20 FOR I=1 TO 3
30 I=2
40 NEXT
RUN
Loops forever
```

After entering the loop, you may jump out before the normal exit. The loop variable retains its current value:

```
2Ø FOR I=1 to 3
3Ø IF I=2 THEN 6Ø
4Ø NEXT
5Ø ?"NORMAL EXIT":END
6Ø ?I:END
RUN
 2
```

You may jump back into a loop you have jumped out of. But you may not jump into a virgin loop. Reading NEXT... without first going through FOR... causes an error break.

...STEP Increments other than 1 are implemented using STEP:

```
1Ø FOR X=2.1 TO 3.7 STEP Ø.35
1Ø FOR X=1ØØ TO -1ØØØ STEP -1Ø
1Ø FOR X=Ø TO 1Ø STEP Ø.1*X
```

(Nesting) Loops can be nested.

```
1Ø FOR I=1TO3
2Ø FOR J=1TO3
3Ø NEXT J
4Ø NEXT I
```

In the above example, the J could have been left off of line 3Ø since a NEXT without a variable name is assumed to apply to the last FOR... statement encountered.

```
1Ø FOR I=1 TO 3:FOR J=1 TO 3
4Ø NEXT I:NEXT J
RUN
?N\ ERROR IN   4Ø (NEXT without FOR error)
```

If the loops end together, a shorter NEXT statement can be used:

```
40 NEXT A,B,C,D,E,F,G,H,I,J,K,L
```

Up to 12 loops can be nested.

DATA... For storing initial data in a program. It is reasonably economical of storage space as it stands. This example uses 12 bytes. It becomes wasteful of space to transfer this to a dimensioned array as shown under READ... (below). DATA statements can contain string constants also:

```
DATA 1,2,3,"A","BIG"
```

Only the order of the data as it is stored in the program is important, not the number of DATA statements used. The following two sets are equivalent:

```
10 DATA 1,2,3,4,5   is the same as
10 DATA 1,2
11 DATA 3,4,5  except the latter takes up
```

more room in memory.

READ... The entries in DATA statements must be transfered to other statements for use:

```
10 DATA 1,2,3,4,5,1,2
20 FOR I=1 TO 7:READ A(I):NEXT
```

The 22 bytes used to store line 10 are now joined by many more, those in statement 20 as well as the 4 bytes/number in the A(I) array and its overhead bytes. If simultaneous use of these integers is not needed, much storage space can be saved. Example:

```
10 DATA 1,2,3,4,5,1,2
20 FOR I=1 TO 7:?READ A:NEXT
```

As READ statements "use up" data, a pointer is set to the next available data entry. The DATA statements are used in numerical order in the source program, no matter where the READ statements are located.

```
10 DATA 1,2
20 GOSUB 9000
30 READ B
40 ?A;B:END
9000 DATA 3,4
9001 READ A:RETURN
RUN
  1  2
```

RESTORE — This command restores the above mentioned pointer to the first entry in the first DATA statement in the program.

CLEAR — This statement cancels the variable table so that it will start being reconstructed from new as the program continues. It also has the effect of a RESTORE command on the DATA pointer.

PRINT... — The variable and expression values following the word PRINT are displayed on the screen. In writing a source program the symbol "?" can be substituted for the word PRINT. PRINT without any expressions prints a blank line. There are two kinds of separators in the list of items to be printed following a PRINT command. They are comma and semicolon. The comma organizes the material into 5 columns separated by 15 spaces. If the material in a given column is longer than 15 spaces or otherwise would overlap the next column, the next column is skipped. If there are more than 5 items in the list to be printed, then more than 1 line is used.

The semicolon puts the printed fields adjacent to each other. Thus strings would be printed without spaces between them. Example:

```
1?"A";"Z"
RUN
AZ
```

But numbers have a space attached to each side so:

```
1 ?1;2
RUN
 1  2
```

Comma and semicolon separators can be used in the same list. The combinations get complicated and it is advised that you experiment to see directly what effects can be obtained.

There are two functions that are used in PRINT statements so we take them up here.

SPC(X) This function is used in PRINT statements to add spaces between outputs from the list. The argument of the function is a numerical constant, variable, or expression that can take on values between Ø and 255. If it is not an integer value, it is truncated to an integer value. The value Ø is interpreted as 256. Large values will cause the printing to continue on the next line, or even later. Example:

```
1?"123456789"
2?SPC(3);"A"
RUN
123456789
   A
```

TAB(X) This function acts like the tab function of a typewriter. Example:

```
1 ?"123456789012345"
2 ?TAB(2);"A";TAB(10);"A"
RUN
123456789012345
  A        A
```

INPUT... This command allows input of data to the machine from the keyboard or tape. It can be preceded by a comment. Example:

```
1 INPUT "LOOK";A,B,C
2 ?A;B;C
RUN
LOOK? 1,2,3
 1  2  3
```

In the above example, the three numbers and 2 commas after LOOK? were entered from the keyboard. Strings can also be entered. Example:

```
1 INPUT A$
```

DEF FN... Used to define a "user defined" function. The function can be defined anytime before use. This is further explained under the heading " USER DEFINED FUNCTIONS".

POKE... This operator stores an integer N in a location W of memory. Example:

```
10 I=2:X=53256
20 POKE X+10*I,I+1
RUN
Stores 3 in address 53276
```

An error is reported if the number to be stored is out of range. Programs that unintentionally POKE values into pages $ØØ, Ø1, or Ø2 can cause very peculiar errors as the run continues, eventually BASIC may become so scrambled that RESET must be done. Since variables that haven't been defined have value zero, it quite often happens that address $ØØØØ is ruined. Then if the (BREAK) key is hit, a warm start cannot be accomplished. This can be corrected by using the MONITOR to put $4C back into $ØØØØ. Of course, expressions can be arguments of POKE. Example:

```
POKE Q*2+3,I+32
```

PEEK(X) This is a function, not a command. But it is the natural opposite of POKE so we discuss it here. PEEK returns the value (as a decimal integer between Ø and 255 inclusive) of the contents of address W. Example:

```
10 I=3
20 ?PEEK(I*256)
RUN
Ø
```

STOP STOP causes an exit to immediate mode with the printing of a break message. Example:

```
20 FOR I=1 TO 10
30 IF I=3 THEN STOP
40 NEXT
RUN
BREAK IN 30
OK
```

Now you may do various immediate commands such as PRINT I and get results. Just so long as you do not add any new statements or delete any statement you can continue with one of the RUN or GOTO commands. Examples:

```
?I
I=4:CONT   or
GOTO 2Ø
```

END This command is optional under many conditions. If the program reaches the last line of source code and that line doesn't transfer the flow to another program line, you may omit END. Each of these two programs yields the same results:

```
1Ø ?"END"        and        12 ?"END":END
RUN                         RUN
END                         END
```

The END statement is necessary if the program is to end in the middle of the source code. Example:

```
5A=1
1Ø IF A=1Ø THEN END
2Ø A=A+1:GO TO 1Ø
```

STRING OPERATOR

There is only one, concatenation, using a + sign:

```
1 A$="A":B$="V"
2 C$=A$+B$:?C$
RUN
AV
```

All strings that are not contained in BASIC source code statements are stored in "string memory" at the top of RAM memory.

NUMERICAL OPERATORS

```
-      Negation                          -5, -N1
^      (SHIFT/N) Exponentation           2^3=8
*      Multiplication
/      Division
+      Addition
-      Subtraction
```

The above numerical operators have their usual meanings in arithmetic and algebra and may be used with parentheses to make explicit the order of evaluation. Inappropriate order may give an error message. Consider the following examples done in the immediate mode:

```
?2*-3     get -6
?2-*3     get ERROR
?2+++3    get 5
?2^-1.5   get Ø.353553
?2-^1.5   get S┘ERROR
```

Parentheses can be nested up to 12 deep.

BOOLEAN OPERATORS

These operators return values of-1 for TRUE and Ø for FALSE.

```
>           Greater than
<           Less than
<> or ><    Not equal
=           Equal to
<= or =<    Less than or equal to
>= or =>    Greater than or equal to
```

Some examples in the immediate mode:

```
X=2:?2=X      get  -1
X=2:?X=2      get  -1
?2<3          get  -1
?2>3          get   Ø
```

Just after a warm start you may get an O┐ ERROR instead.

BIT MANIPULATION OPERATORS

Numbers that are in the range of -32768 to +32767 inclusive are treated as 16 bit 2's complement numbers by the following operators. (Truncation to integers is performed, if necessary.) Consult the appropriate section for an explanation of 2's complement binary numbers. Some examples in the immediate mode:

```
?NOT -2.1      get 2
?NOT 2E4       get -2ØØØ1
?NOT 2E6       get F/ ERROR
?1 OR 2        get 3
?1 AND 2       get Ø
?1 OR 3ØØØØ    get 3ØØØ1
```

AND For each bit in the pair of numbers connected by AND, the corresponding bit in the result is one only if both the bits are 1. This is most easily seen by an example in binary notation:

%Ø1Ø1 AND %ØØ11 = ØØØ1

OR Inclusive OR. The given bit is 1 if either (or both) numbers have a 1 for that bit position.

Ø1Ø1 OR ØØ11 = Ø111

NOT Each bit of the number is reversed, 1 for Ø and Ø for 1:

NOT Ø1Ø1 = 1Ø1Ø

USER DEFINED FUNCTIONS

Functions can be defined by using a DEF... statement anytime before use. The function has 1 variable but other parameters can also occur in the definition and will be given their current values at the time of use. Any number of functions can be used in one program.

```
10 DEF FNAX(X) = 3*X+B
20 Z=2
25 B=1
30 ?FNAX(Z-1)
RUN
 4
```

Not allowed: FNA$(X), FNA$(X$), FNA(X,Y), FNA(A$)

Function variables are stored in six bytes, among the numerical and string single variables. There is an $80 added to the first byte of the name to signify that the variable is a user defined function. Note that one is allowed to have all the following 5 variables in the same program because they are always stored under different names or in separate parts of the variable table.

AB, AB$, AB(I), AB$(I), FNAB(I)

STRING FUNCTIONS

String functions either have a string as an argument, or yield a string as a value, or both. Those that return a string value have a name that ends in $.

ASC(A$) Returns the ASCII value (decimal integer) of the first character in the string A$.

CHR$(A) Returns the character whose ASCII value is A. If you have the graphics chip, CHR$(A) will print the corresponding graphics character for A such that Ø<A<255. Example:

```
10 FOR I=0 TO 255
20 X$=CHR$(I)
30 Y=ASC(X$)
40 ?X$;Y
50 NEXT
```

This program prints all the graphics characters (except for I=Ø, because the CRT routine ignores nulls). When 1Ø, line feed is printed, a line feed occurs. When 13, CR is printed, a carriage return occurs.

LEFT$(A$,I) Gives the left most I characters of A$. If I=Ø there is an F/ ERROR reported.

RIGHT$(A$,I) Gives the right most I characters of A$. If I=Ø an ERROR is returned.

MID$(A$,I,J) This is intended to give a string J characters long, starting at the Ith character of A$ and continuing to the right. But in no case is MID$ longer than from the Ith character to the end of A$ inclusive, even for large J. If J is omitted, then MID$ goes to the end of A$. If I>LEN(A$) then MID$ is of zero length.

LEN(A$) Returns the length of A$.

STR$(X) Gives a string which is a representation of the number X. Example:

```
10 N=6.023E23
20 N$="AVOGADRO'S NUMBER IS "+STR(N)
30 ?N$
40 ?LEN(STR$(N))
RUN
AVOGADRO'S NUMBER IS 6.023E23
 10
```

VAL(A$) The opposite of STR$. If A$ is a string representing a number, VAL returns the corresponding value. If A$ does not represent a number, VAL returns 0. Example:

```
10 A$="-0.03E23"
20 ?VAL(A$)
RUN
-3E+21
```

Another:

```
10 A$="A"
20 ?VAL(A$)
RUN
0
```

FRE(A$) Not allowed unless A$ has been previously defined. Then it has the same effect as FRE(1) or any other numerical valued function or constant.

NUMERICAL FUNCTIONS

In the following functions, the argument may be any constant, variable or expression that has a numerical value. Example in immediate mode:

?EXP(NOT 1.1) get Ø.135335

ABS(X) Yields the absolute value of X. For X=2,Ø,-2 it returns 2,Ø,2 respectively.

INT(I) Truncates decimal number to an integer. For I=1.1,Ø,-1.2 it gives 1,Ø,-2 respectively.

SGN(X) Gives the sign of X. For X=Ø, there is no sign. For X=2,Ø,-2 it gives 1,Ø,-1 respectively.

RND(X) This is a pseudorandom number generator. If the argument is Ø it yields the same number as the previous call gave. If the argument is negative, it serves as a seed which resets the generator and changes its period. The number returned by the negative seed is not itself useful as a random number. In ordinary use the argument is a positive number and a pseudorandom number between Ø and 1 is returned. If not seeded, the generator has a period of 1861. That is, only 1861 separate "random" numbers are produced, and then further calls repeat this sequence in the same order. The generator should be tested with negative seeds to see if it remains a good generator. I have not done this.

SQR(X) Square root, for positive arguments only. Example:

?SQR(1ØØØØ9Ø) get 1ØØØ.Ø5

EXP(X) Exponential e^X where e=2.71828.

LOG(X) Natural log. You can obtain the log to base 1Ø by using LOG(X)/LOG(1Ø). The argument X must be positive.

SIN(X) Sine of X where X is in radians. The conversion that 180° is pi radians is needed to work problems given in degrees of angle. These trig functions seem accurate to within the number of digits shown on the screen.

COS(X)
TAN(X)
ATN(X) The cosine, tangent and arctangent are likewise defined for arguments in radians.

FRE(X) This function returns the number of bytes in RAM (that have been allocated to BASIC at cold start time) that have not yet been used to store source code, variable tables or strings in high memory. Example for a 4K machine whose memory was set to 1032 at cold start time:

```
1?FRE(1)            RUN
2A$="A"             199
3 ?FRE(Ø)           193
4 A$=A$+A$          191
5 ?FRE(Ø)
```

The value of the argument doesn't matter for this function. In the above example, the first FRE printing gives the bytes free after the source program is stored. The second allows for the variable table for A$, 6 bytes long. The third allows for the string "AA", 2 bytes long stored at $Ø3FD and Ø3FE. When FRE is called, it performs a "garbage compaction" of the strings stored in high memory, discarding the no longer used strings and compacting the rest into highest memory.

TAB(X) Discussed at the PRINT command.

SPC(X) Likewise

POS(X) Used with terminals. Gives the current location of the print head.

USR(X) See the separate discussion of the use of this function that allows one to interface machine language subroutines to BASIC programs.

PEEK(X) Used to return the numerical value (decimal) stored in a given memory address. See commands after POKE... .

WAIT I,J,K Used to interogate a memory location, especially an input or output port flag register. The memory location I (decimal) is exclusive OR'ed with K and then AND'ed with J. This is repeated until a non-zero result is obtained, upon which the execution of the next statement is begun. Examples of use are given under tape input and output. If K is omitted it is taken to be zero.

DIM(X,Y,...) Used to assign dimensions to the indices of an array. (See the discussion under ARRAY). Its most familiar use is with constant arguments at the beginning of a program:

```
5 DIM U12(16)
```

but it can be used with variable array sizes:

```
1Ø INPUT N
2Ø DIM ER(2*N+1)
```

ARRAYS

String arrays and numerical arrays are similar in all respects except for the value stored in the 4 bytes of each element. The value for a numerical variable is a 4 byte floating point number. The "value" for a string variable is information as to how long the string is and the address of its first byte. The string is usually stored in the source code statement as a string constant. If not, it is stored at the end of RAM memory.

Arrays can have from 1 to 11 indices. For example, A(I,J,K) has 3 indices, and XZ$(A) has one. The indices take on values Ø through a maximum given by a DIM statement. DIM A(2) sets up an entry in the variable table for A with 3 elements A(0), A(1), and A(2). If no dimension statement is encountered before an array is used, the dimension of each index defaults to 1Ø (so the index is allowed to take on values Ø through 1Ø). The maximum size any index can be assigned in a DIM statement is 32767, but with 4 bytes per element (plus overhead bytes), obviously real arrays must be much smaller than this. An array can be dimensioned only once, either by a DIM statement or a default. Space in the variable table is assigned to the array at the time of dimensioning. Any number of arrays, DIM statements and arrays per DIM statement can be used.

The total space an array occupies in the variable table is shown by considering DIM A(5,6,7):

3	overhead (name and number of indices)
2x3	2 bytes for each index (to give its length)
6x7x8	number of elements in the array
x4	4 bytes per element

Then the total size in the table is 3+2x3+(6x7x8)x4=1353 bytes
All arrays are stored after all single variables in the tables.

BUGS IN BASIC

There are 2 problems using string variables in BASIC. The first occurs when a string variable stored in high memory is redefined. BASIC doesn't know that the string has been abandoned and continues to hold space for it. If this cycle is repeated, memory eventually fills up.

```
1 A$="B"
10 FOR I=1 TO 100
20 B$=B$+A$
50 NEXT
60 B$=""
70 GO TO 10
```

There is a way out however. If 65 ?FRE(9) or even 65 X=FRE(1) is inserted, BASIC does an accounting when it encounters FRE and the unused strings are abandoned.

This leads to the second problem. If a string <u>array</u> has been defined, then when FRE is interpreted, the program may hang, with occasional screen flickers. The solution to this problem was provided by Mark Minasi and published in PEEK(65). Simply pick the dimension of the array as 3*(any integer)+2. This is no hardship, because there will be such a number near any desired array size.

Another bug in BASIC occurs just after a warm start. If you try to execute an immediate command, you may get an error message. The cure is to just repeat the command. You can avoid this problem by entering any keystroke and (RETURN), accept the error and go on to the desired command.

USR(X) FUNCTION
MACHINE LANGUAGE SUBROUTINES IN BASIC

USR(X) You can write a machine language subroutine which can be called from BASIC, do its stuff, and return to the BASIC program. This is done with the USR function. If desired, the argument X of USR(X) can take a 16 bit number to the subroutine. Two bytes can be returned to BASIC as the value of USR(X). Each of these transfers is a little involved, so first we will demonstrate the simplest case, where the subroutine is called, but no numbers are passed either way. Write a BASIC program:

```
20 Y=USR(X)
50  STOP
```

Now (BREAK) (M) to enter the MONITOR and place these numbers at the addresses shown:

```
$000B 22
 000C 02
 0222 60  RTS
```

The address Ø222 contained in the two bytes at ØB and ØC is the start of our program, which in fact only has one instruction, RETURN. Now do a (BREAK) (W) for a <u>warm</u> start of BASIC and RUN. If all is well you will get BREAK IN 50.

The next step is to pass a value, X, to the machine program. Write:

```
5 INPUT "X";X
20 Y=USR(X)
40 ?"X,Y";X;Y
50 ?
99 GOTO 5
```

(BREAK) (M) to MONITOR and enter code starting at

```
$Ø222  2Ø 4Ø Ø2  JSR
       A5 AE     LDA FACHI
       8D 2Ø D2  STA screen left
       A5 AF     LDA FACLO
       8D 22 D2  STA screen right
       6Ø        RTS
            . . .
 Ø24Ø  6C Ø6 ØØ  JMP indirect
```

The code whose address is stored at $Ø6 is a subroutine which takes X and converts it to a 16 bit 2's complement number and puts it in:

$ØØ AE	$ØØAF	
LO byte	HI byte	16 bit number
FACLO	FACHI	

Our subroutine must pick it up from there and in this case we poke it onto the screen as a graphics symbol which you can look up in the GRAPHICS MANUAL. Now (BREAK) (W) for a warm start and RUN to see the results. Notice that the value of X in BASIC is unchanged by all this, and Y has some peculiar value. It was necessary to do the two step JSR Ø24Ø and JMP indirect to get back to our machine code. Otherwise the JMP would take us to a subroutine that would return us to BASIC.

The last step is to return 2 bytes from the machine code. This is done by putting bytes into the Y register and the accumulator. These are transfered to the value of USR as a 16 bit signed number using another machine language program whose starting address is contained in $ØØØ8 and ØØØ9. This code will return us directly to the BASIC program. Add to the previous BASIC program:

```
5 INPUT "A,X,B";A,X,B
8 R=3*256
9 POKE R-2,A:POKE R-1,B
```

(BREAK) (M) to MONITOR and add to our previous program:

```
$Ø22F  AC FF Ø2    LDY B
       AD FE Ø2    LDA A
       6C Ø8 ØØ    JMP indirect
```

(BREAK) (W) for a warm start and RUN. The variable Y is now formed from the 2 bytes A and B in 2s complement form, A being the HI byte and B the LO.

To make the BASIC program's use of machine language trouble free to the user, the machine language instructions, as well as the starting address, can all be POKE'ed into memory.

SPACE SAVING

The most important attribute of a program (after requiring that it run correctly) is clarity, so that a reader can understand it easily. This requires careful structuring of subroutines and statements, many REMarks, spacing between characters (FOR M=1, not FORM=1), distinctive variable names, etc. When space in memory becomes tight, all this may go by the board. In addition, some other tricks to save space may be tried.

You can reuse variable names. If Z$ is used only once, in an INPUT statement for example, and a later string is called D$, then replace Z$ with D$. This saves 6 bytes. The same applies to numerical variables. Watch array use. DIM A(1) with elements A(Ø) and A(1) requires 15 bytes in the variable table, while A1 and A2 together require 12 bytes in the variable table, and may also save in the source program. In fact A,B instead of A1 and A2 would save 2 bytes in the source code.

Long arrays are more efficient than many equivalent single variables. It is wasteful to use floating point variables to store small integers. Sometimes they can be "packed". For example, instead of A=1,B=2,C=3 (18 bytes in the variable table) pack D=10203, so A=INT(D/10000) B=INT(D/100)-A*10000, etc. Of course, the decoding statements in the source code take up a lot of room, so there is a net loss, but if the variables are large arrays, the savings could be substantial.

The practice of initializing arrays using DATA statements is wasteful of space. Consider these 2 programs which do the same job:

```
1 DIM H(9)
2 DATA Ø,1,2,3,4,5,6,7,8,9
3 FOR I=Ø TO 9:READ H(I):NEXT
4 ?H(3)
```

and

```
1 H$="Ø123456789"
2 ?VAL(MID$(H$,4,1))
```

The second program saves 78 bytes in memory by storing the integer constants in a string, from which they can be recovered for use relatively easily.

Use multiple statements on each line number:

```
1 A=1:B=2
```

instead of

```
1 A=1
2 B=2
```

will save 4 bytes for each colon used. Put REM's on a functioning line for the same reason:

```
1 REM START
2 A=3
```

uses 4 more bytes than

```
2 A=3:REM START
```

Repeated characters are not stored in a packed manner in the source program.

```
1 AAAAA=1:REM GO
1 A=1:REM     GO
```

Both require the 15 characters you see (including the space characters). Both have the same variable table too.

Sometimes integers can be stored on the screen via POKE's and recovered via PEEK's. This may be possible in a game where the display itself can be data. Or if a C2 machine is using the 32x32 display, the blank half of the screen memory can be used for data storage. The margins of the 25x25 display of a C1 machine may also be used. Multiplexing the screen memory may also work, going to a short machine language routine via USR which uses the screen as memory but accomplishes its deeds very fast and then clearing the screen again, returns to BASIC in the twinkling of an eye. I haven't tried this multiplexing method yet. In 1 second you can do about 20,000 machine operations, 10 for each memory cell in the display.

Since most of page $Ø2 is unused, it is a good place to put your machine language subroutines that are accessed via USR. You can also change the vectors in page $ØØ so that BASIC memory starts at $Ø222 instead of Ø3ØØ. Do a cold start, then (BREAK) (M) to the monitor. Put $ØØ in $Ø222, $23 in $ØØ79 and $Ø2 in ØØ7A. Then (BREAK) (W) to warm start and NEW (RETURN) to reset the rest of page $ØØ. You are ready to go with BASIC with 7/8 of a page extra room!

TAPES, BASIC AND HOMEMADE

Ever wonder what is on the tapes of your programs that you have SAVEd? It is not what is in memory, exactly! It is more like what is on the screen as you LIST. Suppose your source program were:

```
1 AAAAA
2 BBBBB
```

Of course this program won't run, but its code is in memory. Suppose that you do a NULL 2 in immediate mode and then a SAVE, LIST to put the program on tape. The code on tape is ASCII which we here represent in decimal numbers.

```
13  0  0  0  0  0  0  0  0  0  0
10  0  0 13  0  0  0  0  0  0  0  0  0
10  0  0 32 49 32 65 65 65 65 65 13  0  0  0  0  0  0  0  0  0  0
10  0  0 32 50 32 66 66 66 66 66 13  0  0  0  0  0  0  0  0  0  0
```

10 line feed

32 space

13 return

The two nulls after the 10 (line feed) are the work of the NULL command. Default is zero nulls. Each line begins with a line feed and ends with a carriage return followed by 10 nulls. Two empty lines are sent before the BASIC program code starts.

The tape port address of a C2 is at $FCØØ=64512, and for a C1 or a Superboard II is at $FØØØ=6144Ø. You might want to read your BASIC tapes with a program like this:

```
1 Q=64512:R=Q+1
4 WAIT Q,1
5 ?PEEK(R):GO TO 4
```

But this program WON'T WORK for reading BASIC because the PRINT is too slow and so you will skip some bytes. This program will work for reading your own tapes if you space the bytes out when making the tape, more later.

You can read a BASIC tape by storing the bytes in an array:

```
1 DIM D(200)
2 Q=64512
3 R=Q+1
4 WAIT Q,1
5 D(I)=PEEK(R):I=I+1:GO TO 4
```

When you get an error break because you tried to fill D(201), you can go to immediate mode with

```
FOR I=1 TO 200: ?D(I);:NEXT
```

to see the output. The problem here is that the first part of D may be filled with noise. You may have trouble deciding where the taped program starts.

If you want to store some data on tape, you can go two routes. If the amount of data is relatively little, so that time to tape and read is not important, then you may use the functions already in BASIC, such as PRINT, INPUT, SAVE, AND LOAD. Here is a program to illustrate that.

```
1Ø DIM Y(2Ø)
3Ø FOR I=1TO2Ø:Y(I)=I:NEXT
4Ø SAVE
45 FOR I=1TO5;?Ø:NEXT:?255
5Ø FOR I=1TO2Ø:?Y(I):NEXT
6Ø LOAD:INPUT"HIT SPACE BAR TO CONTINUE";A$
99 END
1ØØØ DIM Y(2Ø):LOAD
1Ø1Ø INPUT X:IF X<>Ø THEN 1Ø1Ø
1Ø2Ø INPUT X:IF X=Ø THEN 1Ø2Ø
1Ø3Ø FOR I=1TO2Ø:INPUT Y(I):NEXT
1Ø4Ø INPUT"HIT SPACE BAR TO CONTINUE";A$
1Ø5Ø FOR I=1 TO 2Ø:?Y(I);:NEXT
9999 END
```

To write to tape do RUN. To read from tape do RUN1ØØØ.

Line 45 puts a leader on the tape that is recognized by lines 1Ø1Ø and 1Ø2Ø. Lines 6Ø and 1Ø4Ø allow one to get out of the LOAD mode. The LOAD in line 6Ø is to get you out of the SAVE mode.

A faster way to store data from an array to tape is to use this program.

```
1 DIM D(2ØØ)
2 GOSUB 1ØØ:REM TO PUT YOUR STUFF IN D
3 Q=64512:R=Q+1
4 FOR I=1TO2ØØ:WAIT Q,2
5 POKE R,D(I)
6 PRINT D(I):REM TO SLOW THINGS DOWN
7 NEXT
```

The resulting tape can be used with the first program we gave in this section.

Finally, this faster way to read and write tape will probably need to use the "leader" method that we used on the previous program.

AUTOLOAD TAPE

Machine language tapes from OSI use the autoload format. Each byte to be sent is broken down into the two ASCII characters that represent it in hexadeciaml notation. For example, if %1111ØØ11 is the form stored, it is sent as 2 bytes, F and 3, or in ASCII as $46 and $33. After each such pair of characters, a (RETURN)= $ØD is sent. Thus 1 byte in memory is recorded as 3 bytes on tape. This strange method is designed to use the monitor for tape in a way that mimics the keyboard, and allows the tape itself to switch to the keyboard mode, at the end of the loading process, so that an autostart feature is possible.

The characters to be found on the tape are the 16 hexadecimal digits Ø to F, and

.	$2E
(RETURN)	ØD
/	2F
G	47

which are familiar to you by your use of the monitor.

The tape format also includes the starting address of the code to be taped (or to be loaded) and the starting address of the code to be executed. This can be the program just loaded, some other program, warm start of BASIC (ØØØØ) or the monitor (FEØØ or FFØØ). The G for go is optional. Representing the 2 bytes by H and L (for high nybble and low nybble) and (RETURN) by R, the whole tape format is as follows:

```
.HL HL / HLR HLR HLR ...HLR.HL HL G
```

The left HL HL is the starting address, LSB (least significant byte) first. The right most HL HL is the starting address at which the monitor will start execution if the G is found on the tape (or entered from the keyboard).

FLOATING POINT NUMBERS

Single numerical variables require 6 bytes of table space, 2 for the name and 4 for the value. Numbers are stored in a floating point binary representation. The first byte gives the exponent. The next 3 bytes give the mantissa and sign. For example:

3 = %ØØ11 in one binary nybble

(% preceding a number indicates it is in binary, $ indicates hexadecimal.)
You can add as many binary zeros as you wish to the left (just as in decimal numbers).

3 = %ØØØØ ØØ11 in one byte

Make it a fraction by moving the "radix point":

3 = %Ø.11 x 2^{+2} in analogy with
3 = Ø.3 x $1Ø^{+1}$

So the internal representation of 3 could look like this:

$$3 = \underbrace{\$82}_{\text{exponent}}\ \underbrace{\%11ØØ\ ØØØØ\ \$ØØ\ ØØ}_{\text{3 byte mantissa}} \quad \text{but doesn't, quite.}$$

The exponent is +2, but it has been biased by adding $8Ø so that negative exponents can also be expressed. Then -2 is represented by $7E and an exponent of zero by $8Ø.

However, we have not yet represented the sign of 3 (+3 and not -3). Also, there is a redundancy, since the first digit of the mantissa will always be 1. So remove this redundant 1 and replace it with a sign bit, Ø for + and 1 for -. The final result is:

3 = %11
3 is stored as $82 %Ø1ØØ ØØØØ $ØØ ØØ = $82 4Ø ØØ ØØ
-3 as $82 %11ØØ etc. = $82 CØ ØØ ØØ

The largest number that can be represented by this system with no error is

$$2^{24}-1 = 256^3-1 = 16{,}772{,}215 = \%1111\ 1111\ 1111\ 1111\ 1111\ 1111$$

It looks like $98 7F FF FF in the table. When numbers are translated into decimal for presentation on the screen, considerable accuracy is lost. 16,777,215 is presented as 1.6772E+Ø7.

What happens if you try to store an undefined value? The 2 line program

```
1 A=B
2 ?A
RUN
 Ø
```

runs OK. The variable B, of course, is undefined and has no entry in the table. A is represented by the 6 bytes

```
41
ØØ
ØØ
ØØ
A5
7D
```

TWO'S COMPLEMENT BINARY NUMBERS

To represent signed numbers, the left most bit is reserved to be a sign bit (Ø for + and 1 for -). Then the best way to represent negative numbers is in the 2's complement form. Example:

4	%Ø1ØØ
3	ØØ11
2	ØØ1Ø
1	ØØØ1
Ø	ØØØØ
-1	1111
-2	111Ø
-3	11Ø1
-4	11ØØ

To get the negative of any number (+ or -) when in 2's complement integer form, first invert each digit (every 1 goes to Ø and Ø to 1). Then add 1 (with binary carry).

Example: 3 = %ØØ11

-3 = %11ØØ + 1 = 11Ø1

-2 = %111Ø

2 = %ØØØ1 + 1 + ØØ1Ø

in an 8 digit integer:

4 = %ØØØØ Ø1ØØ = $Ø4

-4 = %1111 1Ø11 + 1 = 1111 11ØØ = $FC

SOURCE CODE AND VARIABLE TABLES

The source code memory is rearranged as each line is entered so as to keep the lines in numerical order. Adding or deleting a line from source code "destroys" the variable table. (Pieces or all of it may be found by looking in memory with the monitor or PEEK.) We illustrate storage by some very simple programs:

```
    1 A=3
    RUN

$Ø3ØØ  ØØ   start of source program
       Ø9   address of next line
       Ø3
       Ø1   line number
       ØØ
       41   A
       AB   token for =
       33   3 in ASCII
       ØØ   line end symbol
       ØØ   when address of next line is zero, source ends.
       ØØ
       41   variable table starts.  First 2 bytes are name A.
       ØØ
       82   Next 4 bytes are value 3 in floating point.
       4Ø
       ØØ
       ØØ
       empty ...

    1 A$="B"
    RUN

$Ø3ØØ  ØØ   Start of source program
       ØC
       Ø3
       Ø1
       ØØ
       41   A
       24   $ token
       AB   = token
       22   " token
$Ø3Ø9  42   B in ASCII
       22   " token
       ØØ   line end
       ØØ   program end (2 bytes)
       ØØ
       41   A
       8Ø   $
       Ø1   length of string
       Ø9   address of first byte of string (2 bytes)
       Ø3
       ØØ
```

```
10 DEF FNAB(A)=A*2
RUN

$Ø3ØØ  ØØ          $Ø314  C1   FNAB
       12                 42
       Ø3                 ØE   address of definition of FNAB
       ØA                 Ø3
       ØØ                 1C   address of value of argument
       95   DEF           Ø3
       2Ø   space         41   A
       9E   FN            ØØ
       41   A             ØØ   4 byte value of A
       42   B             ØØ
       28   (             ØØ
       41   A             ØØ
       29   )             empty ...
       AB   =
 Ø3ØE  41   A
       A5   *
       32   2
       ØØ
 Ø312  ØØ
       ØØ
```

In the above example, if we add the line

```
20 Z=2:? FNAB(Z+3)
```

after RUNning the address value of the argument would still be that of the value of A, even though the execution of FNAB calculated the argument as the value of Z+3=5, and A is unchanged.

When strings are concatenated, they are stored at the end of memory. For a 16K machine the last byte is $3FFF. When the following program is run, its variable table looks like this:

```
1 A$="B"        $Ø31B   41
2 A$=A$+A$               8Ø
RUN                      Ø2   string is 2 bytes long
                         FE   its first byte is at $3F FE.
                         3F
                         empty ...

                $3FFE    42
                         42
```

ARRAY STORAGE

We illustrate the storage of array variables by showing the variable table for this program:

```
10 DIM A(1,2)
20 FOR I=Ø TO 1
30 FOR J=Ø TO 2
40 A(I,J)=10*I+J
50 NEXT
RUN
```

The Variable table starts at $Ø348:

```
$Ø348 49 I                   $Ø35D ØØ =0
      ØØ                           ØØ A(0,0)
      82 2                         ØØ
      ØØ                           ØØ
      ØØ
      ØØ                           84 =10
      4A J                         2Ø A(1,0)
      ØØ                           ØØ
      82 3                         ØØ
      4Ø
      ØØ                           81 =1
      ØØ                           ØØ A(0,1)
      41 A                         ØØ
      ØØ                           ØØ
  $21=33=6x4+9=
      ØØ  size of                  84 =11
          table                    3Ø A(1,1)
                                   ØØ
      Ø2 2 indices                 ØØ
      ØØ
      Ø3 J has 3                   82 =2
          values                   ØØ A(0,2)
      ØØ  I has 2                  ØØ
      Ø2                           ØØ

                                   84 =12
                                   4Ø A(1,2)
                                   ØØ
                                   ØØ
                                   empty...
```

Unlike a speedometer, the fastest changing digit is the one of the left. Note also that table size has its most significant digit last but the index size has it first!

BASIC TRACE

Knowledge of some of the things stored in pages Ø to 3 during the running of your programs allows you to write some subroutines to do exotic things. Here is a crude example of a TRACE routine.

```
 $ØØBC  4C 2A Ø2  JMP Ø22A
    BF  EA EA EA  NOP

  Ø22A  E6 C3     INC LO address
    2C  DØ Ø2     BNE
    2E  E6 C4     INC
    3Ø  A5 C3     LDA $C3 fetch address to this program
    32  8D 3B Ø2  STA $Ø23B
    35  A5 C4     LDA $C4
    37  8D 3C Ø2  STA $Ø23C
    3A  AD ØØ ØØ  LDA $---- load character
    3D  8D 1Ø D1  STA $D11Ø store on screen
    4Ø  2Ø ØØ FD  JSR $FDØØ wait for keystroke
    43  4C C2 ØØ  JMP $ØØC2 return
```

Cold start BASIC and write a short program. Then (BREAK) (M) to monitor and enter the code listed above. When finished loading and checking the code, (BREAK) (W) to warm start BASIC. Now RUN your program. The characters of RUN and your program will appear on the screen one by one. After each one, hit a (SPACE BAR) to go to the next. The tokens for BASIC reserved words will appear on the screen as graphics characters. You can use your GRAPHICS MANUAL and a list of tokens for reserved words to decode, but usually the letter and numerical characters alone will be enough (with careful attention) to keep you located in the program. At any point you can break to inspect various variables and, by going to monitor, to inspect memory locations for flag values, etc.

MEMORY MAP

C2-4P with 16 K of memory and a BASIC-IN-ROM Version 1.0, Rev. 3.2.
Most of these entries are due to Bruce Hoyt and to Jim Butterfield.

00	4C 74 A2	JMP to warm start. $BD11 earlier, cold start
03	4C C3 A8	JMP to message printer. A,Y contain lo,hi address of start of message. Message ends with a null.
06	05 AE	INVAR, USR get argument routine address
08	C1 AF	OUTVAR, address of USR return value routine
0A	4C 88 AE	JMP to USR(X) routine
0D	00	number of nulls after Line Feed, set by NULL command. Note! <u>not</u> the nulls after CR.
0E	00	line buffer pointer
0F	48	terminal width. $48=72
10	38	input col. limit
11	00 40	integer address
13 to 5A		line buffer
5B	22	used by dec. to bin. routine, search character, etc.
5C	22	scan-between-quotes flag
5D	--	line buffer pointer, number of subscripts
5E	--	default DIM flag
5F	FF	type: $FF=string, $00=numeric
60	--	DATA scan flag, LIST quote flag, memory flag
61	00	subscript flag, FNx flag
62	--	$00=input, $98=read
63	--	comparison evaluation flag
64	00	CNTL-O flag. $80 means suppress output
65	68 65 00	temporary string (descriptor stack) pointers
68	06 92 A1	stack of descriptors for temporary strings
6B	-- -- --	"
6E	-- -- --	"
71	92 A1	temporary variable pointer, also used by dec. to bin.
73	47 9B	pointers, etc
75	-- --	product staging area for multiplication
77	-- --	"

```
79   01 03    address of start of source program in RAM
7B   03 03                              single variable table
7D   03 03                              array variable table
7F   03 03                              empty BASIC memory
81   FF 3F                              high string storage space
83   -- --    temporary string pointer
85   00 40    address + 1 of end of BASIC memory
87   -- FF    current line number
89   -- --    line number at STOP, END or (CTRL/C) break
8B   -- 00    program scan pointer, address of current line
8D   -- --    line number of present DATA statement
8F   00 03    next address in DATA statements
91   -- --    address of next value after comma in present DATA
                statement
93   -- --    last variable name
95   12 --    last variable value address
97   -- --    address of current variable, pointer for FOR/NEXT
99   -- -- -- work area; pointers, constant save, etc.
9C   -- -- --    "
9F   -- 03       "
A1   4C -- 00 JMP, a general purpose jump
A4   -- -- -- misc. work area and storage
A7   -- FE 00 "
AA   -- --    pointer to current program line
AC to B0      first floating point accumulator.  E,M,M,M,S
AC   06 92    AD and AE are printed in decimal by $B962
AE   68       FACHI, byte transfered by USR(X)
AF   00       FACLO,     "
B0   20       sign of Acc. #1
B1   --       series evaluation constant pointer
B2   00       accumulator #1 high order (overflow) word
B3 to B7      second floating point accumulator. E,M,M,M,S
B3   80 00 00 10 00      E=exponent, M=mantissa byte
B8   92       sign comparison, acc. #1 vs. #2
B9   A1       acc. #1 low order (rounding) word
```

BA	98 A1	series pointer
BC to D3		routine copied from $BCEE. It is the start of a subroutine to go through a line character by character.
BC	E6 C3	INC lo byte of address of character
BE	D0 02	BNE
C0	E6 C4	INC hi byte if needed
C2	AD 00 03	LDA with a character of the line.
C5	C9 3A	CMP #$3A is it a colon?
C7	B0 0A	BCS branch is yes, statement done
C9	C9 20	CMP #$20 is it a space?
CB	F0 EF	BEQ branch if yes, get another character
CD	38	SEC set carry
CE	E9 30	SBC #$30
D0	38	SEC
D1	E9 D0	SBC #$D0
D3	60	RTS end of subroutine, character in A
D1 to D7		used by OSI extended monitor as well as BASIC
D4	80 4F	random seed
D6	C7 52	"
D8 to FF		unused by BASIC
FB		monitor load flag
FC		" data byte
FD		"
FE	-- --	" current address
100 to 10C		ASCII numerals built in this space
130		NMI interrupt location
1C0		IRQ " " , can be overwritten byBASIC
133 to 1FF		BASIC stack

200 to 20E			used to output to the screen and tape
200			cursor location, initialized to contents of $FFE0
201			save character to be printed
202			temporary
203			LOAD flag, $80 means LOAD from tape
204			temporary
205			SAVE flag, 0 means not SAVE mode
206			repeat rate for CRT routine
207 to 20E			part of scroll routine
207	B9 00 D7	LDA $D700,Y	
20A	99 00 D7	STA $D700,Y	
20D	C8	INY	
20E	60	RTS	
20F to 211			unused
212	00		CNTL/C flag, not 0 means ignore CTRL/C
213	OD 96 OD OD		used by keyboard routine
217			?
218 to 221			used in 600 board machines as follows:
218			input vector
21A			output vector
21C			CNTL/C vector
21E			LOAD vector
220			SAVE vector

A000 - A083	command jump table
A084 - A162	keyword table
A164 - A185	ERROR message table
A1A1	search stack for most recent GOSUB or FOR
A1CF	routine to open space in program for another line
A212	check stack size
A21F	check free memory left
A24C	contains offset from $A164
A24E	message out
A274	warm start
A295	tokenize and store in BASIC
A2A2	delete a line from program
A357	input a line to input buffer
A386	input a character, calls routine at FFEB
A399	toggles the CTRL/O flag
A3A6	convert keywords in input line
A432	find program line number less than number in $11-12 put address in $AA-AB
A461	NEW routine
A477	initialize
A491	clear stack, reset addresses
A4A7	initialize program scan pointer to beginning of program.
A4B5	LIST
A556	FOR routine
A5F6	execution routine
A61A	RESTORE
A629	CNTL/C routine
A638	STOP
A63A	END
A661	CONT
A67B	NULL
A686	CLEAR
A691	RUN
A69C	GOSUB
A6B9	GOTO

A6E6	RETURN
A70C	DATA
A73C	IF
A74F	REM
A75F	ON
A77F	decimal to binary, put asnwer in $11-12
A79B	LET
A82F	PRINT
A866	end of input line routine, puts out CR and LF & nulls
A8C3	string output routine, address in A,Y (lo, hi) end the string with a null
A8E5	output routine, calls $FFEE
A923	INPUT
A94F	READ
AA40	NEXT
AAC1	expression handler
ABAC	non-numeric expressions
ABD8	NOT
ABFB	SN errors
AC66	OR
AC69	AND
AC96	comparison
AD01	DIM
AD8B	create new variables
AE05	= command
AE17	create new arrays
AFAD	FRE
AFCE	POS
AFDE	DEF
B08C	STR$
B147	garbage collector
B2FC	CHR$
B310	LEFT$
B33C	RIGHT$
B347	MID$
B38C	LEN
B39B	ASC

B3AE	arithmetic expression, error if over 255
B3BD	VAL
B408	floating number in floating accumulator converted to fixed and put in $11-12
B41E	PEEK
B429	POKE
B432	WAIT
B458	- command
B46F	+ command
B5BD	LOG
B5FE	* command
B6CD	/ command
B7D8	SGN
B7E8	fixed to floating. fixed in $AD-AE to floating in $AC-AF
B7F5	ABS
B862	INT
B953	output line number
B95E	hex in A,X converted to decimal and printed
B962	output decimal value of number (binary) in $AC-AF
B96E	build ASCII number in $100-10C from number in $AC-AF
BAAC	SQR
BAB6	^ raise to a power
BB1B	EXP
BBC0	RND
BBFC	COS
BC03	SIN
BC4C	TAN
BC99	ATN
BCEE	Get character routine, moved to $BC
BD11	cold start
BE39	cold start messages
BF2D	CRT routine

```
FE00-   A2 28     LDX   #$28      MONITOR:  initialize
FE02-   9A        TXS               initialize stack to $28
FE03-   D8        CLD               clear decimal mode
FE04-   AD 06 FB  LDA   $FB06       initialize UART on 430 board
FE07-   A9 FF     LDA   #$FF          continue
FE09-   8D 05 FB  STA   $FB05         continue
FE0C-   A2 D8     LDX   #$D8        CLEAR TV SCREEN:  X hi byte of end address
FE0E-   A9 D0     LDA   #$D0        A holds hi byte of screen start address
FE10-   85 FF     STA   $FF         hi byte: current address of screen
FE12-   A9 00     LDA   #$00        lo byte
FE14-   85 FE     STA   $FE           store
FE16-   85 FB     STA   $FB           store
FE18-   A8        TAY               set FETCH flag to $00: means input from kybd
FE19-   A9 20     LDA   #$20        load space char. into A
FE1B-   91 FE     STA   ($FE),Y     store space on screen
FE1D-   C8        INY                 next
FE1E-   D0 FB     BNE   $FE1B       repeat
FE20-   E6 FF     INC   $FF         increment hi byte of current screen address
FE22-   E4 FF     CPX   $FF         done it 8 times?
FE24-   D0 F5     BNE   $FE1B       if not, branch and repeat
FE26-   84 FF     STY   $FF         if so, set hi byte of screen address to $00
FE28-   F0 19     BEQ   $FE43       branch always to IN: display for $0000
FE2A-   20 E9 FE  JSR   $FEE9     ADDRESS mode (.): fetch char from tape or kybd
FE2D-   C9 2F     CMP   #$2F        is it (/)?
FE2F-   F0 1E     BEQ   $FE4F       if yes, branch to DATA mode (/)
FE31-   C9 47     CMP   #$47        is it (G)?
FE33-   F0 17     BEQ   $FE4C       if yes, branch and GO: execute program
FE35-   C9 4C     CMP   #$4C        is it (L)?
FE37-   F0 43     BEQ   $FE7C       if yes, branch and set FETCH flag, read tape
FE39-   20 93 FE  JSR   $FE93       JSR to LEGAL:change char. from hex to binary
FE3C-   30 EC     BMI   $FE2A       branch if char. is illegal hex digit
FE3E-   A2 02     LDX   #$02        roll address in memory
FE40-   20 DA FE  JSR   $FEDA       IN: JSR to ROLAD
FE43-   B1 FE     LDA   ($FE),Y     load A from current address
FE45-   85 FC     STA   $FC         store in $FC
FE47-   20 AC FE  JSR   $FEAC       update screen display
FE4A-   D0 DE     BNE   $FE2A       branch always: get next char.
FE4C-   6C FE 00  JMP   ($00FE)     GO: execute program at current address
FE4F-   20 E9 FE  JSR   $FEE9     DATA mode (/): look for keyboard character
FE52-   C9 2E     CMP   #$2E        is it (.)?
FE54-   F0 D4     BEQ   $FE2A       if yes, go to ADDRESS mode (.)
FE56-   C9 0D     CMP   #$0D        is it (RETURN) key?
FE58-   D0 0F     BNE   $FE69       if no, roll in and display hex digit
FE5A-   E6 FE     INC   $FE         else increment address lo byte
FE5C-   D0 02     BNE   $FE60         need increment hi byte?
FE5E-   E6 FF     INC   $FF             if yes, do so
FE60-   A0 00     LDY   #$00        set Y for rolling data
FE62-   B1 FE     LDA   ($FE),Y     load data from current address in $FE,FF
FE64-   85 FC     STA   $FC         store data from memory in $FC
FE66-   4C 77 FE  JMP   $FE77       JMP to INNER: display on screen, then to(/)
FE69-   20 93 FE  JSR   $FE93       JSR to LEGAL: convert char. to binary
FE6C-   30 E1     BMI   $FE4F       branch if char. was not legal hex
FE6E-   A2 00     LDX   #$00        prepare to roll DATA nybble into memory
FE70-   20 DA FE  JSR   $FEDA       roll one nybble into $FC ($FD also changes)
FE73-   A5 FC     LDA   $FC         load current data byte from $FC
FE75-   91 FE     STA   ($FE),Y     store in next spot in memory
FE77-   20 AC FE  JSR   $FEAC       INNER: JSR to DISPLAY
FE7A-   D0 D3     BNE   $FE4F       branch always to DATA mode (/)
```

```
FE7C-   85 FB      STA   $FB       store L in $FB, FETCH flag
FE7E-   F0 CF      BEQ   $FE4F     branch to keyboard input if flag $00
FE80-   AD 00 FC   LDA   $FC00    OTHER: read tape from ACIA 6850
FE83-   4A         LSR             shift bit of status register to C
FE84-   90 FA      BCC   $FE80     if bit $00, ACIA is not ready
FE86-   AD 01 FC   LDA   $FC01     fetch char. from tape
FE89-   EA         NOP
FE8A-   EA         NOP
FE8B-   EA         NOP
FE8C-   29 7F      AND   #$7F      strip off parity bit, leaving ASCII char.
FE8E-   60         RTS             return
FE8F-   00         BRK
FE90-   00         BRK
FE91-   00         BRK
FE92-   00         BRK
FE93-   C9 30      CMP   #$30     LEGAL: hex to binary conversion, bit 7 set if
FE95-   30 12      BMI   $FEA9     branch if too small for hex           error
FE97-   C9 3A      CMP   #$3A      compare to $3A
FE99-   30 0B      BMI   $FEA6     branch if less than $3A: was hex 0 to 9
FE9B-   C9 41      CMP   #$41      compare to letter "A"
FE9D-   30 0A      BMI   $FEA9     branch if between ASCII : and @
FE9F-   C9 47      CMP   #$47      compare to letter "G"
FEA1-   10 06      BPL   $FEA9     branch if too large
FEA3-   38         SEC             set carry bit, char. is A to F
FEA4-   E9 07      SBC   #$07      subtract to form binary number
FEA6-   29 0F      AND   #$0F      mask off high nybble
FEA8-   60         RTS             return
FEA9-   A9 80      LDA   #$80      load A with neg. number for error flag
FEAB-   60         RTS             return
FEAC-   A2 03      LDX   #$03     DISPLAY: displays 4 bytes (erases 1 byte)
FEAE-   A0 00      LDY   #$00      set starting point on screen: $D0C6
FEB0-   B5 FC      LDA   $FC,X     byte to be displayed: $FF,FE,FD,FC in order
FEB2-   4A         LSR               shift
FEB3-   4A         LSR               shift
FEB4-   4A         LSR               shift
FEB5-   4A         LSR               shift
FEB6-   20 CA FE   JSR   $FECA     JSR DISNYB: display hi nybble
FEB9-   B5 FC      LDA   $FC,X     reload byte
FEBB-   20 CA FE   JSR   $FECA     JSR DISNYB: display lo nybble
FEBE-   CA         DEX               repeat above for next byte
FEBF-   10 EF      BPL   $FEB0     do 4 bytes altogether
FEC1-   A9 20      LDA   #$20      $20 is space
FEC3-   8D CA D0   STA   $D0CA     blank  out display of byte from $FD
FEC6-   8D CB D0   STA   $D0CB       continue
FEC9-   60         RTS             return
FECA-   29 0F      AND   #$0F     DISNYB: display 1 nybble on the screen
FECC-   09 30      ORA   #$30      AND the hi nybble to zero, add $30 to byte
FECE-   C9 3A      CMP   #$3A      compare to $3A
FED0-   30 03      BMI   $FED5     branch if hex is 0 to 9
FED2-   18         CLC             clear carry bit: number was 10 to 15
FED3-   69 07      ADC   #$07      add 7 to get ASCII letter A to F
FED5-   99 C6 D0   STA   $D0C6,Y   store on screen
FED8-   C8         INY             increment to next screen location
FED9-   60         RTS             return
FEDA-   A0 04      LDY   #$04     ROLAD: roll hex digits into 2 bytes of memory
FEDC-   0A         ASL             shift 4 times to put lo nybble in A to
FEDD-   0A         ASL               hi nybble in A
FEDE-   0A         ASL
```

```
FEDF-   0A         ASL
FEE0-   2A         ROL            roll A: bit 7 to C
FEE1-   36 FC      ROL  $FC,X     roll next memory
FEE3-   36 FD      ROL  $FD,X     roll next
FEE5-   88         DEY              next
FEE6-   D0 F8      BNE  $FEE0     do for 4 bits
FEE8-   60         RTS            return
FEE9-   A5 FB      LDA  $FB     FETCH: first check FETCH flag
FEEB-   D0 91      BNE  $FE7E     if not zero, read from tape
FEED-   4C 00 FD   JMP  $FD00     was zero, jump to keyboard (RTS from there)
FEF0-   A9 FF      LDA  #$FF    LOOK: looks for any keystroke
FEF2-   8D 00 DF   STA  $DF00     strobes all rows of keyboard at once
FEF5-   AD 00 DF   LDA  $DF00     records which col.s had keys down
FEF8-   60         RTS            return
FEF9-   EA         NOP
FEFA-   30 01                   Here are 3 addresses left over from when
FEFC-   00                        this code was in page $FF and these were
FEFD-   FE C0 01                  interrupt addresses
```

Changes from the above for a C1 machine: page $FE.

```
FE0C  A2 D4      screen size is smaller
FEEB  D0 93
FEF0  BA FF       jump table read into page $02 from
      69 FF         support ROM program
      9B FF
      8B FF
      96 FF
```

(Changes on page $FF for C1 and Superboard II machines, continued from last page.)

```
FFE0      $67
  E1      $17
  E2      $00
  E6      $9F
  EA      $9F
FFEB      $6C 18 02
          $6C 1A 02
          $6C 1C 02
          $6C 1E 02
          $6C 20 02
```

```
FF00-  D8         CLD              SUPPORT ROM: clear decimal mode
FF01-  A2 28      LDX   #$28        initialize stack to $28
FF03-  9A         TXS                 continue
FF04-  20 22 BF   JSR   $BF22       initialize 6850 ACIA
FF07-  A0 00      LDY   #$00        initialize some page $02 flags, etc.
FF09-  8C 12 02   STY   $0212         "
FF0C-  8C 03 02   STY   $0203         "
FF0F-  8C 05 02   STY   $0205         "
FF12-  8C 06 02   STY   $0206         "
FF15-  AD E0 FF   LDA   $FFE0       initialize cursor position
FF18-  8D 00 02   STA   $0200         "
FF1B-  A9 20      LDA   #$20        $20 is "space"
FF1D-  99 00 D7   STA   $D700,Y     clear screen
FF20-  99 00 D6   STA   $D600,Y       "
FF23-  99 00 D5   STA   $D500,Y       "
FF26-  99 00 D4   STA   $D400,Y       "
FF29-  99 00 D3   STA   $D300,Y       "
FF2C-  99 00 D2   STA   $D200,Y       "
FF2F-  99 00 D1   STA   $D100,Y       "
FF32-  99 00 D0   STA   $D000,Y       "
FF35-  C8         INY                 "
FF36-  D0 E5      BNE   $FF1D         "
FF38-  B9 5F FF   LDA   $FF5F,Y     write "C/W/M ?" on screen
FF3B-  F0 06      BEQ   $FF43       branch if reached null at message end
FF3D-  20 2D BF   JSR   $BF2D       JSR to CRT routine in BASIC
FF40-  C8         INY                 next letter of message
FF41-  D0 F5      BNE   $FF38         continue
FF43-  20 B8 FF   JSR   $FFB8       JSR INPUT: fetch char. from tape or keyboard
FF46-  C9 4D      CMP   #$4D        is it (M)?
FF48-  D0 03      BNE   $FF4D       if no, branch
FF4A-  4C 00 FE   JMP   $FE00       if yes, JMP to MONITOR
FF4D-  C9 57      CMP   #$57        is it (W)?
FF4F-  D0 03      BNE   $FF54       if no, branch
FF51-  4C 00 00   JMP   $0000       if yes, JMP to BASIC warm start
FF54-  C9 43      CMP   #$43        is it (C)?
FF56-  D0 A8      BNE   $FF00       if no, branch and seek new key stroke
FF58-  A9 00      LDA   #$00        if yes, set registers to zero and
FF5A-  AA         TAX                 "
FF5B-  A8         TAY                 "
FF5C-  4C 11 BD   JMP   $BD11       JMP to BASIC cold start

FF5F   43 2F 57 2F 4D 20 3F 00
       C  ,  W  ,  M     ?

FF67-  20 2D BF   JSR   $BF2D      OUTPUT: char. to tape and TV screen
FF6A-  48         PHA               save char.
FF6B-  AD 05 02   LDA   $0205       test for SAVE flag
FF6E-  F0 22      BEQ   $FF92       if not save, branch, PLA and return
FF70-  68         PLA               pull char. from stack
FF71-  20 15 BF   JSR   $BF15       go write char. on tape
FF74-  C9 0D      CMP   #$0D        was char. a CR?
FF76-  D0 1B      BNE   $FF93       if no, branch and return
FF78-  48         PHA               if yes, push char on stack
FF79-  8A         TXA               save X on stack too
FF7A-  48         PHA                 "
FF7B-  A2 0A      LDX   #$0A        $0A=10
```

```
FF7D-   A9 00      LDA  #$00    write 10 nulls on tape: load A with 10
FF7F-   20 15 BF   JSR  $BF15   go write a null on tape
FF82-   CA         DEX            repeat 10 times
FF83-   D0 FA      BNE  $FF7F   done?
FF85-   68         PLA          yes, recover A, X
FF86-   AA         TAX            "
FF87-   68         PLA            "
FF88-   60         RTS          return
FF89-   48         PHA         LOAD flag: set LOAD flag, reset SAVE flag
FF8A-   CE 03 02   DEC  $0203   set LOAD flag: load enabled
FF8D-   A9 00      LDA  #$00    null in A to reset SAVE flag, disable SAVE
FF8F-   8D 05 02   STA  $0205        SAVE flag
FF92-   68         PLA          recover A from stack
FF93-   60         RTS          return
FF94-   48         PHA         SAVE: sets SAVE flag
FF95-   A9 01      LDA  #$01    $01 for set SAVE mode
FF97-   D0 F6      BNE  $FF8F   branch always
FF99-   AD 12 02   LDA  $0212  (CTRL/C) routine: checks for (CTRL/C) break
FF9C-   D0 19      BNE  $FFB7   if (CTRL/C) flag in $0212 is set, return
FF9E-   A9 01      LDA  #$01    strobe row 1 of keyboard
FFA0-   8D 00 DF   STA  $DF00     "
FFA3-   2C 00 DF   BIT  $DF00   check for CTRL key depressed
FFA6-   50 0F      BVC  $FFB7   if not, branch and return
FFA8-   A9 04      LDA  #$04    strobe row 4 of keyboard
FFAA-   8D 00 DF   STA  $DF00     "
FFAD-   2C 00 DF   BIT  $DF00   check if key (C) is depressed
FFB0-   50 05      BVC  $FFB7   if not, branch and return
FFB2-   A9 03      LDA  #$03    if so, load A with 3 and jump to BASIC
FFB4-   4C 36 A6   JMP  $A636
FFB7-   60         RTS          return
FFB8-   2C 03 02   BIT  $0203  INPUT: read tape and/or keyboard
FFBB-   10 19      BPL  $FFD6   branch if LOAD is disabled: JMP to keyboard
FFBD-   A9 02      LDA  #$02    poll row 2 of keyboard
FFBF-   8D 00 DF   STA  $DF00     "
FFC2-   A9 10      LDA  #$10    check col. 5 of keyboard
FFC4-   2C 00 DF   BIT  $DF00   was it "space bar"
FFC7-   D0 0A      BNE  $FFD3   if yes, branch to disable LOAD and go to kybd
FFC9-   AD 00 FC   LDA  $FC00   if no, check status of 6850 ACIA
FFCC-   4A         LSR            "
FFCD-   90 EE      BCC  $FFBD   branch if data is not yet ready
FFCF-   AD 01 FC   LDA  $FC01   else load char. from ACIA to A
FFD2-   60         RTS          return
FFD3-   EE 03 02   INC  $0203   disable LOAD flag
FFD6-   4C ED FE   JMP  $FEED   JMP to keyboard, get char.
FFD9-   00         BRK
FFDA-   00         BRK
FFDB-   00         BRK
FFDC-   00         BRK
FFDD-   00         BRK
FFDE-   00         BRK
FFDF-   00         BRK
FFE0-   40                      cursor home
FFE1-   3F                      line size
FFE2-   01                      machine type: C1 is zero, C2    one
```

```
FFE3-   00
FFE4-   03
FFE5-   FF
FFE6-   3F
FFE7-   00
FFE8-   03
FFE9-   FF
FFEA-   3F
FFEB-   4C B8 FF    JMP   $FFB8    INPUT
FFEE-   4C 67 FF    JMP   $FF67    OUTPUT
FFF1-   4C 99 FF    JMP   $FF99    (CTRL/C)
FFF4-   4C 89 FF    JMP   $FF89    LOAD flag set
FFF7-   4C 94 FF    JMP   $FF94    SAVE flag set
FFFA-   30 01       BMI   $FFFD    NMI address, non-maskable interrupt
FFFC-   00                         restart address
FFFD-   FF                           "
FFFE-   C0 01                      address for maskable interrupt

BF07-   AD 00 FC    LDA   $FC00  TAPE PORT, INPUT: 6850 ACIA
BF0A-   4A          LSR            move receive data flag to C
BF0B-   90 FA       BCC   $BF07    branch if data not ready
BF0D-   AD 01 FC    LDA   $FC01    else load data into A
BF10-   F0 F5       BEQ   $BF07    branch for more data if data was a null
BF12-   29 7F       AND   #$7F     else AND off the bit 7
BF14-   60          RTS            return
BF15-   48          PHA          TAPE PORT, OUTPUT: 6850 ACIA
BF16-   AD 00 FC    LDA   $FC00    after saving data in A, loadstatus register
BF19-   4A          LSR            shift twice to put Xmit data flag in  C
BF1A-   4A          LSR
BF1B-   90 F9       BCC   $BF16    branch if ACIA not ready
BF1D-   68          PLA            else pull data into A
BF1E-   8D 01 FC    STA   $FC01    send to ACIA
BF21-   60          RTS            return
BF22-   A9 03       LDA   #$03   ACIA   initialization
BF24-   8D 00 FC    STA   $FC00    perform master RESET of ACIA
BF27-   A9 B1       LDA   #$B1     load ACIA control register for
BF29-   8D 00 FC    STA   $FC00      8 bits, no parity,  2 stop bits
BF2C-   60          RTS              enable receive interrupt logic:return
```

Page $FF in C1 and Superboard II machines is like that in the C2-4P except where noted below.

FF04 - 0D load jump tables from FE0F to page $02

FF0F   initialize ACIA using routine at FCA6

FF12 - 34 initialize page $02 and clear screen

FF35 - 5E similar to FF38 onward of C2-4P

FF55 - 68 table "C,W,M,D ? null"

FF69 - 8A like OUTPUT of C2-4P at FF67 - 88 except write on tape at FCB1, not BF15

FF8B - 99 LOAD and SAVE

FF9B - B9 (CTRL/C) routine like C2-4P at FF99 - B7

FFBA - DA INPUT, C1 keyboard is inverted from that of C2-4P. ACIA is at F000