OSI

BASIC IN ROM

ALL ABOUT OSI BASIC IN ROM

SECOND EDITION

Second Printing

CONTENTS

INTRODUCTION

This book is intended for users of OSI Microsoft BASIC-IN-ROM, Version 1.0, Rev. 3.2, which is used on all OSI BASIC-IN-ROM machines. The material is presented on 2 levels. The first is pure BASIC. The complete set of commands, statements, functions and operators is 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.

The second level takes a systems viewpoint. It examines the functional parts of the BASIC system, including many details of the machine language implementation of BASIC, which allow exotic programs to be written. Using it, your programming will improve in speed, clarity, economy of storage and ease of human interface to screen, keyboard and mass storage. Sample utilities are included, such as line renumber.

OVERVIEW

$ = HEX.

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 5 areas of memory containing code. They are the BASIC <u>interpreter</u> stored in ROM from $A000 to BFFF, the <u>line buffer</u> stored in zero page from $13 to $59, the <u>source</u> program you write, stored from $0300 up, the <u>variable tables</u> stored immediately after the source code, and the <u>string storage</u> at the end of RAM memory.

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, the line is stored in the proper spot in the source program and we continue in the immediate mode. If the line did not start with a line number, the interpreter executes it from the line buffer exactly as if it were a one line program. This one line program may consist of several statements separated by colons, and may create, refer to, or alter the variable tables.

SOME DEFINITIONS

Constant — Constants are of two types, numerical and string.
Examples: numerical 3.62 — string "Computer"

Variable — Variables are of two types, numerical and string. They are distinguished from each other by appending a $ sign on the end of string names. Examples: numerical AY — string AY$

Command — A command causes the computer to execute some definite procedure. Examples: PRINT, LIST, RUN, X=6. Some commands need expressions to be complete. Example: ON ... GOTO ... as used in: ON J*(J+1) GOTO 30, 40, 50

Operator — The usual algebraic operators + - * / plus some others such as < , > , >= , <> .

Function — A function has arguments (numerical or string) and returns one value (numerical or string). Examples: SIN(X), LEFT$(A$,2)

Expression — An expression is a set of constants, variables, operators and functions (which themselves may have expressions as arguments) which has a definite numerical or string value. Examples: A, 3.3, 2*X+Y, 3.1+A*SIN(PI/2), "AT THE END", A$, N$+CHR$(I+J)+"HELP"

Statement — Each statement consists of a single command. Example: CLEAR. The command may require constants and/or operators. Example: PRINT A(X,2)

Line — A line consists of one or more statements separated by a colon ":" and possibly starting with a line number in the range 0 to 63999. Examples:

```
22 PRINT
50 A=3:GOTO 71
LIST
```

STRING CONSTANTS

The most common form for a string constant is a set of ASCII characters set between quotes. Example: "YOUR TURN" But other (non-printing) ASCII characters, or indeed, any hex number can be included in a string. Examples:

```
100 A$="YOUR TURN"+CHR$(13):REM 13 is the CR code
200 HO$=CHR$(14):REM 14 is the graphics character of a house
300 PRINT "THIS IS A HOUSE"+HO$
```

NUMERICAL CONSTANTS

Numbers are represented in source code as integers, decimals, fractions or in scientific notation. Examples: 7, 0.03, -2.E-5, 3/4

Numbers cannot, unfortunately, be represented in source code in binary or hexadecimal form. When numbers are read from source code for use, they are converted into a floating point binary number with a one byte exponent and a 3 byte mantissa. The magnitude of the floating point number varies from about 10^{-38} to 10^{+38}. The largest 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 the display shows considerably less accuracy than what is in memory. 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 $00. If the variable is a <u>string</u>, then $80 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	$41 00	A
A$	41 80	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 the $ for a string). The effect of truncation of the source name is demonstrated in this program:

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

COMMANDS

We will divide commands into 3 groups. <u>Editor commands</u> are used only in the immediate mode. <u>Immediate mode commands</u> can also be used in the run mode, but may perform in a defective manner there. The largest group comprises the <u>run mode commands</u> and all these also work satisfactorily in the immediate mode.

We depart from the usual nomenclature because it is arbitrary and confusing. For example, NEW is often 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). My nomenclature follows the rules set up in the section "SOME DEFINITIONS".

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). Multiple, simultaneous key strokes will be separated with a /.

(SHIFT/O)	Types a _ and erases the last character typed from the line buffer. It doesn't erase it from the screen. This method of "erasing" is left over from the teletype days. Several software houses have "line editor" programs that give true backspace erasing as well as other editor functions.
(SHIFT/P)	Types an @ and erases the line from the line buffer. You still see the line 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, and the machine returns to the immediate mode, ready for more text input.

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

123 (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 beginning at the first line (stored at $0301). Discards the old variable table and constructs a new one as it interprets.

RUN 31 Starts at line 31 of the source code. Gives US ERROR if there is no line 31. Otherwise runs and discards the old variable table and makes a new one.

GOTO 31 Starts running at line 31. Keeps the old variable table.

GOSUB 31 Jumps to line 31 and runs. Keeps old variable table. Expects to find a RETURN statement.

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.

LIST -31 Lists from start of program through line 31.

(CTRL/C) Interrupts execution of the source program or LISTing and returns to the immediate mode. (CTRL/C) may be disabled by POKE 530,1 and enabled by POKE 530,0.

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

LOAD Sets the LOAD flag. This enables the tape port and disables the keyboard (except the (SPACE BAR) key is polled). Then any input from tape is put into the line buffer and treated as usual, depending on whether it starts with a number or not. To exit from LOAD, hit the (SPACE BAR). To further understand LOAD, look at the code in the support ROM at $FF89 and FFB8.

NEW Deletes the present program. It does not erase it from memory however. One thing it does is to load $00 into addresses $0301 and 0302. 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, use the MONITOR, and starting at $0300, step along, looking for the address of the second line of the program. Put the address back into $0301 and 0302 in the format described under the heading SOURCE CODE AND VAR. TABLES. 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 Used to write to tape. The procedure for saving BASIC programs is SAVE, (RETURN), LIST, (but don't type (RETURN) yet), start tape and wait a few seconds to give a leader, then hit the (RETURN) key. To save part of a program, use the appropriate LIST, eg. LIST 100-300. Exit from the SAVE mode by doing LOAD, (RETURN), (SPACE BAR).

It works like this. SAVE calls the short routine at $FF94 to set a flag in $0205. Then whenever BASIC calls OUTPUT (at $FF67) to write to the TV screen, (using the routine at $BF2D), it also transmits each character to the tape port. The time required for transmission by the 6850 ACIA slows down the whole cycle, which is why you see the rate of writing to the screen slowed down.

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 0 to 8. However, the number of nulls requested is poked into $0D, so you can request up to 255 nulls by POKEing into address 13.

IMMEDIATE MODE COMMANDS USED IN RUN MODE

RUN 31 Same as "CLEAR:GOTO 31".

LIST Or LIST 31, etc. Does the indicated LISTing, then <u>goes to immediate mode</u>. A very unhandy characteristic!

CONT Program hangs until you enter (CTRL/C) from the keyboard.

LOAD Sets LOAD flag, with usual results. To get back to normal, next statement should read INPUT A$, and then hit (SPACE BAR).

NEW Poison! If NEW is encountered in your program, it "erases" your program and goes to immediate mode!

SAVE Works normally. Sets the flag in $0205.

NULL Works normally.

RUN MODE COMMANDS

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

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

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

```
10 REM *** PROGRAM ITCH ***
20 REM
30 A=2:REM A IS THE NUMBER OF BITES
```

Statements after a REM cannot be reached by the interpreter.

```
30 A=2:REM CASE TWO:B=4
```

The statement "B=4" cannot be reached. If the REMark follows a GOTO ... , the word REM can be omitted because the interpreter will never reach far enough into the line to detect the syntax error. Example:

```
10 GOTO 33:GO MOVE PIECE          is as good as
10 GOTO 33:REM GO MOVE PIECE
```

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 123456789ABC
2 REM          XX
```

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

GOTO ... Example:

```
GOTO 9900
```

Not allowed:

```
GOTO N
```

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

GOSUB ... Subroutine calling command. Example:

The statements are executed in the order 5,7,13,15,8,10.

```
5 A=2
7 GOSUB 13
8 B=3
10 END
13 A=A+1
15 RETURN
```

```
ON...GOTO...        Example:     5 ON M GOTO 10,20,30

                    The flow is:    if M=0 go to next statement after 5
                                       M=1 go to statement 10
                                       M=2 go to statement 20
                                       M=3 go to statement 30
                                       M=4 or larger, statement after 5
```

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

```
ON...GOSUB...    Example:     5 ON Z GOSUB 10,12,15,3
```

If Z=0 or Z is greater than 4, go to the next statement. If Z= 1,2,3,4 then GOSUB 10,12,15,3 respectively. Upon RETURN, goto the next statement after 5.

```
IF...GOTO...     Example:     10 IF A=2 GOTO 100
```

If A=2 then the next statement executed is line 100. If A≠2 then the next line after the IF...GOTO... is executed. In place of "A=2" there can be any expression that has a numerical value, or otherwise can be interpreted as Boolean "false", i.e. false has the numerical value zero. If the expression is not zero, then it is assumed to be true. This is a more extended interpretation of "true", which should actually be the numerical value -1. Examples:

```
IF A$="DA" GOTO 338
IF (INT(X) AND 12)=8 GOTO 4
IF 3*X > PEEK(Q) GOTO 65
IF Y GOTO 21:GOES TO 21 UNLESS Y IS NOT EQUAL TO ZERO
```

(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:

```
10 IF X < 7.2 THEN X=7.2:GOSUB 10:GOTO 30
```

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

```
20 FOR I=1 TO 3
30 PRINT I
40 NEXT I
50 PRINT "I IS NOW";I
99 END

OK
RUN
 1
 2
 3
I IS NOW 4
```

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

```
20 FOR I=1 TO 3
30 IF I=2 THEN 60
40 NEXT I
50 PRINT "I IS NOW";I
55 END
60 PRINT I:END
99 END

OK
RUN
2
```

The stack still records that you have entered the loop but not exited through NEXT. See the discussion under STACK. 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 NF ERROR break.

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

```
10 FOR X=2.1 TO 3.7 STEP 0.35
10 FOR X=100 TO -33 STEP -10
10 FOR X=1   TO 10  STEP 0.1*X
```

(nesting) Loops can be nested. (Up to 12 deep).

```
10 FOR I=0 TO 1:FOR J=5 TO 6
30 PRINT I;TAB(5) J
40 NEXT J
45 PRINT "BETWEEN LOOPS"
50 NEXT I
99 END

OK
RUN
0   5
0   6
BETWEEN LOOPS
 1   5
 1   6
BETWEEN LOOPS
```

The index can be left off any or all NEXT statements in the program, and when encountered, a NEXT will be assumed to apply to the last FOR... encountered by the interpreter. But this is somewhat dangerous. The variables are put on the NEXT statements to serve as a check that the logic of the actual program is the logic that the programmer intended.

```
10 FOR I=0 TO 1:FOR J=5 TO 6
30 PRINT I;TAB(5) J
40 NEXT:NEXT
```

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 FOR I=1 TO 0
30 PRINT 1
40 NEXT I
50 PRINT "I IS NOW";I
99 END

OK
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 above). 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. Study this example carefully:

```
 10 A=0.6
 20 FOR I=2*A TO 3*I
 30 PRINT I
 40 NEXT I
 50 PRINT "I IS NOW";I
 99 END
OK
RUN
 1.2
 2.2
 3.2
I IS NOW 4.2
```

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

```
 20 FOR I=1 TO 3
 30 I=2
 40 NEXT I
 50 PRINT "I IS NOW";I
 99 END
OK
 LOOPS FOREVER
```

When the interpreter encounters a NEXT I, it clears the stack of any loop calls nested inside the FOR I=... NEXT loop. In the example below, looping over J is never done, and when NEXT J is finally encountered, the stack has no current record of a FOR J ..., so a NEXT without FOR error break occurs.

```
10 FOR I=1 TO 3:FOR J=1 TO 4
30 PRINT I;TAB(5) J
40 NEXT I
50 NEXT J
99 END

OK
RUN
 1   1
 2   1
 3   1

?NF ERROR IN  50
OK
```

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

```
10 FOR I=1 TO 3:FOR J=1 TO 4
30 PRINT I;TAB(5) J
40 NEXT J,I
99 END
```

DATA...   For storing initial data in a program. Example:

```
10 DATA 6,7,8,X,"Y",CHR$(13)
15 FOR I=1 TO 3:READ A:NEXT
17 PRINT A
20 READ A$:READ B$:READ C$
30 PRINT A$,B$,C$
99 END

OK
RUN
8
X             Y                  CHR$(13)
```

Here X and "Y" are alternate ways to store string data. The CHR$(13) is also treated as a string, not a function. Data statements are reasonably economical of storage space. The overhead is 6 bytes plus 1 byte each for commas, spaces, or quotes. Line 10 above uses 27 bytes to store 13 bytes of data. Only the order of the data as it is stored in the program is important, not the number of data statements used or their placement in the program. Example:

```
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.
```

DATA statements cannot contain variables, or be modified. In the example below, the interpreter treats the A as a string of data, while X is a numerical variable.

```
10 A=3
20 DATA A
30 READ X:PRINT X:END

OK
RUN

?SN ERROR IN  20
```

READ... As the above examples show, entries in DATA statements must be transfered to other statements for use. 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 90
30 READ B,C
40 PRINTA;B;C:END
90 READ A:RETURN
92 DATA 3,4

OK
RUN
 1  2  3
```

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

CLEAR This statement discards the variable table (by resetting pointers) 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:

```
10 PRINT "A";"Z"
RUN
AZ
```

But numbers have a space attached to each side so:

```
10 PRINT 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.

FUNCTIONS FOR PRINT   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 0 and 255. If it is not an integer value, it is truncated to an integer value. The value 0 is interpreted as 256. Large values will cause the printing to continue on the next line, or even later.

```
1 PRINT "123456789"
2 PRINT SPC(3)"A"
RUN
123456789
   A
```

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

```
 1 PRINT "123456789012345"
 2 PRINT TAB(2) "A" TAB(10) "B"

OK
RUN
123456789012345
  A       B
```

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

```
10 INPUT "LENGTH, HEIGHT";L,H
20 PRINT "LENGTH ";L,"HEIGHT ";H,

OK
RUN
LENGTH, HEIGHT? 3,5,66.0
LENGTH  3.5   HEIGHT  66
```

Strings can also be entered.

```
10 INPUT "NAME";NA$
20 PRINT NA$

OK
RUN
NAME? EDWARD H. CARLSON
EDWARD H. CARLSON
```

If you input more numbers or strings than were asked for an ?EXTRA IGNORED message appears, and the interpreter continues with the program.

```
10 INPUT A,B,C

OK
RUN
? 2,3,4,5
?EXTRA IGNORED
```

If there is a type mismatch or other confusion to the machine, it may issue a ?REDO FROM START instruction. Then type all the data in from the start of the INPUT instruction.

```
 10 INPUT A,B,C
 20 PRINT A,B,C

OK
RUN
? 1,2,A
?REDO FROM START
? 4,5,6
 4              5              6
```

If INPUT requests more items than you supply, it will request more with a double ??

```
 10 INPUT "NAME"; NA$,A
 20 PRINT LEFT$(NA$,1)
 30 PRINT A

OK
RUN
NAME? ED
?? 2
E
 2
```

If you answer (RETURN) to the INPUT (without giving a numeral or string answer) the interpreter returns to the immediate mode. You can do the usual poking and tweaking and then return to the program with a CONT. The interpreter will again query you with the INPUT statement.

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 command stores an integer N in a location X of memory. Example: (Stores 5 in 57088)

```
10 X=2:KB=57088:POKE KB,2*X+1
```

An error is reported if the number to be stored is out of range. Programs that unintentionally POKE values into pages $00, 01, or 02 can cause very peculiar errors as the run continues, eventually BASIC may become so scrambled that a cold start must be done. However, the most common error can be fixed more easily. Since variables that haven't been defined are treated as having value zero, it quite often happens that address $0000 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 $0000.

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 of the contents of address X. Of course, the value lies in the range 0 to 255. Example:

```
 10 I=3:PRINT PEEK(I*256)

OK
RUN
 0
```

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

```
 10 FOR I=1 TO 10:PRINT I;
 20 IF I=3 THEN STOP
 30 NEXT

OK
RUN
 1  2  3
BREAK IN  20
OK
```

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, the program ends and the machine exits to the immediate mode. The END statement is necessary if the program is to end in the middle of the source code. You may have any number of END statements.

```
 10 A=2
 20 IF A=10 THEN END
 30 A=A+1:PRINT A;:GOTO 20

OK
RUN
 3  4  5  6  7  8  9  10
OK
```

STRING OPERATOR

There is only one string operator, concatenation, using a + sign.

```
10 A$="1":B$="A"
20 C$=A$+B$
30 PRINT A$,B$,C$

OK
RUN
1             A             1A

OK
```

All strings that are not contained in BASIC source code statements are stored in "string memory" at the end of RAM memory. In the example above, A$ and B$ are stored in line 10 of the program as you see, but C$ is stored in the top 2 bytes of RAM memory.

NUMERICAL OPERATORS

-	Negation	-5, -N1
^	(SHIFT/N) Exponentiation	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 SN ERROR
?2+++3    get 5
?2^-1.5   get 0.353553
?2-^1.5   get SN ERROR
```

BOOLEAN OPERATORS

These operators return values of -1 for TRUE and 0 for FALSE. Why these particular numerical values? Well, zero for FALSE seems reasonable enough, and then TRUE should be NOT 0. But in two's complement form, NOT %0000 0000 0000 0000 is %1111 1111 1111 1111=-1. The % tells us that the number is in binary form, and you may want to consult the sections on TWO'S COMPLEMENT NUMBERS and BIT MANIPULATION OPERATORS.

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

Examples:

```
 10 X=2:PRINT2=X:X=2;X=3;X>3;X<3

OK
RUN
-1 -1  0  0 -1
```

```
10 X=2:Y=X>2
20 PRINTY

OK
RUN
0
```

Two strings can be "compared" by using these operators. By this is meant only that the first character of each string is treated as an ASCII (or other) number. Then these 2 numbers are compared.

```
  10 A$="ABC":B$=CHR$(80)
  20 PRINT A$,B$
  30 PRINTASC(A$),ASC(B$)
  40 PRINTA$>B$

OK
RUN
ABC           P
 65            80
 0
```

BIT MANIPULATION OPERATORS

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

```
20 PRINT 1 OR 2;1 OR 3000
 30 PRINT 1 AND 2
 40 PRINT NOT 2E6

OK
RUN
 2 -20001
 3  3001
 0

?FC ERROR IN  40
OK
```

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

```
%0101 1111 1100 0000 AND
 1100 1010 0000 1111 = 0100 1010 0000 0000
```

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

```
0101 1111 1100 0000 OR
1100 1010 0000 1111 = 1101 1111 1100 1111
```

NOT Each bit of the number is reversed, 1 for 0 and 0 for 1.

NOT 0101 1111 1100 0000 = 1010 0000 0011 1111

USER DEFINED FUNCTIONS

Functions can be defined any time before use by a DEF FN... statement. Functions can be redefined any number of times. The definition may involve other user defined functions but may not be recursive (i.e. the definition of a function cannot involve itself). 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. Study this example carefully:

```
10 DEF FNA(X)=X
15 X=2:PRINT FNA(X)
20 DEF FNA(Y)=2*Y
25 Y=3:PRINT FNA(Y)

OK
RUN
 2
 6
```

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$) Returns the ASCII value (decimal integer) of the <u>first</u> character in the string A$.

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 $0 \leqslant A \leqslant 255$. This program prints all the graphics characters (except for I=0, because the CRT routine at $BF38 ignores nulls). When line 10, line feed, is printed, a line feed occurs. When 13, CR is printed, a carriage return occurs. (I.e. the cursor moves far left on the TV screen.)

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

LEFT$(A$,I) Gives the left most I characters of A$. If I=0 there is an FC ERROR reported.

RIGHT$(A$,I) Gives the right most I characters of A$. If I=0 an FC 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 PRINT N$
 40 PRINT LEN(STR$(N))

OK
RUN
AVOGADRO'S NUMBER IS  6.023E+23
 10
```

Note: You see only 8 characters for N in line 10, but a blank is attached to each end in making STR$(N), for a total of 10 characters.

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

```
 10 A$="-0.05E-21"
 20 B$="A"
 30 PRINT VAL(A$),VAL(B$)

OK
RUN
-5E-23         0
```

FRE(A$) The same as FRE(8), so why bother?

NUMERICAL FUNCTIONS

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

```
? EXP(NOT 1.1)   get 0.135335
```

ABS(X)	Yields the absolute of X. For X=2, 0, -2 it returns 2, 0, 2 respectively.
INT(I)	Truncates decimal number to an integer. For I=1.1, 0, -1.2 it gives 1, 0, -2 respectively.
SGN(X)	Gives the sign of X. For X=0, there is no sign. For X= 2, 0, -2 it gives 1, 0, -1 respectively.
RND(X)	This is a pseudorandom number generator. If the argument is zero it gives the same number as the previous call gave. If the argument is negative, it alters the generator in a way that makes the numbers unpredictible, but not evenly spaced between zero and one. In ordinary use, the argument is a positive number (it doesn't matter which one) and a pseudo-random number between 0 and 1 is returned. 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. A generator with a longer period is presented after the section on NEWSLETTERS.
SQR(X)	Square root, for positive arguments only. Example: `PRINT SQR(1000090)   get 1000.05`
EXP(X)	Exponential e^X Where e=2.71828
LOG(X)	Natural log. You can obtain the log to base 10 by using LOG(X)/LOG(10). The argument X must be positive.
SIN(X)	Sine of X where X is in radians. The conversion that 180^o is pi radians is needed to work problems where the angles are expressed in degrees. 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 coldstart 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:

```
                                                   OK
10 PRINT FRE(8)                                    RUN
20 A$="A":PRINT FRE(8)                              212
30 A$=A$+A$:PRINT FRE(8)                            206
                                                    204
```

The value of the argument doesn't matter for this function. I use 8 because it is near the () keys. In the above example, the first FRE printing gives the bytes free after the source program is stored. The second shows that a variable has been entered in the variable table, taking 6 bytes. The third allows for the string "AA", 2 bytes long, stored at $03FD and 03FE. 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. This may give a problem if string arrays are present. BUGS AND FIXES discusses this problem.

TAB(X) Discussed at the PRINT command.

SPC(X) Likewise

POS(X) Intended for use with terminals. It gives the current location of the cursor on the TV screen. In this example the cursor starts at 0. The string " 0 " is printed. The cursor is then at 8. The string " 8 " is then printed in positions 8, 9, 10.

```
10 PRINT "0123456789"
20 PRINT POS(X) SPC(5) POS(X)

OK
RUN
0123456789
 0       8
```

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 ANDed with J. This is repeated until a non-zero result is obtained, upon which the execution of the next statement is begun. While WAITing, the machine is immune to being stopped with the (CTRL/C) command. Examples of use are given under TAPES.

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

```
10 DIM U1(16)
```

but it can be used with variable array sizes:

```
10 INPUT N,I
20 DIM ER(2*N+1,I),L(I)
```

USR(X) FUNCTION
MACHINE LANGUAGE SUBROUTINES IN BASIC

You may need a machine language subroutine which can be entered from BASIC, do its stuff, and then return control to the BASIC program. This is done with the USR function. If desired, the argument X of USR(X) can take a two's complement 16 bit number to the subroutine. Also, 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 R=USR(S)
50 STOP
```

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

address	code	
$000B	$22	
000C	02	
0222	60	$60 is op code for RTS

The address $0222 contained in the two bytes at $0B,0C is the starting address of our program. It is stored "backwards", $22$02, as is usual for 6502 machine language addresses. Actually, our program is extremely short, consisting of only one instruction, RTS, which means "return from subroutine". Now do a (BREAK),W for a warm start of BASIC, and RUN. If all is well you will hit the STOP in line 50 and see BREAK IN 50 on the screen.

It is awkward to have to put the addresses in $0B,0C so we add:

```
2 POKE 11,34:POKE 12,2
```

to the BASIC program. Of course, one must make the hex to decimal conversion $22=34 and $02=2 in order to be able to write this line. It is also commonly done to poke the machine language program in from DATA statements. See the BASIC TRACE for an example of this.

The next more complicated situation is to pass a value S to the machine language program. Add to the BASIC program:

```
5 INPUT "S";S
40 PRINT TAB(15) "R="R,"S="S
99 GOTO 5
```

(BREAK),M to the monitor and enter code starting at $0222:

```
$0222  20 40 02  JSR
       A5 AE     LDA FACHI
       8D 20 D2  STA left byte on the screen
       A5 AF     LDA FACLO
       8D 22 D2  STA right byte
       60
           . . .
 0240  6C 06 00  JMP indirect
```

The address $06 in page zero is called a pointer. That means the contents of $06,07 is a two byte address, in this case $AE05. This address is the entry point to the subroutine INVAR which takes S and converts it to a 16 bit two's complement number and puts it in $AE,AF, high byte first.

Our subroutine must pick it up from there for use. In this case we poke it onto the screen as two graphics symbols, one for each byte. To see all this action, (BREAK),W for a warm start and RUN. Notice that the value of S in BASIC is unchanged by all this, and R has some peculiar value. The business with the JMP indirect was to allow use of the pointer but not force a premature return to the BASIC program.

The last step in learning to use USR is to write a machine language subroutine that will return 2 bytes to BASIC. It must put them into the Y register and the accumulator, Y being the low byte of the 16 bit number. Then a routine called OUTVAR entered at $AFC1 pointed to by $08 takes these bytes and sends them on to the BASIC program. Add to the previous BASIC program:

```
5 INPUT "A,Y,S";A,Y,S
8 Q=3*256
9 POKE Q-2,A:POKE Q-1,Y
4Ø PRINT TAB(15) "A,Y>R="R,"S"S
```

(BREAK),M to the monitor and add to our previous program:

```
$022F  AC FF 02  LDY Y
       AD FE 02  LDA A
       6C 08 00  JMP indirect
```

(BREAK),W and RUN. The variable R is now formed from the 16 bit two's complement number. R is of course a floating point number. Play around with the program. When the value of A is made higher than 127, the value of R will be negative. Of course, both A and Y must be in the range 0 to 255.

```
1 REM   *** USR(X) DEMONSTRATOR ***
2 POKE 11,34:POKE12,2
5 INPUT "A,Y,S";A,Y,S
8 Q=3*256
9 POKE Q-2,A:POKE Q-1,Y
20 R=USR(S)
40 PRINT TAB(15) "A,Y>R="R,"S="S
42 REM A=HI,Y=LO BYTE OF R AS A 16 BIT TWO'S COMPLEMENT NUMBER
50 PRINT
99 GOTO 5
```

ARRAYS

Numerical arrays and string 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 the string length (given in 1 byte) and the address of its first byte (given in 2 bytes). The fourth byte is always zero. If the string was given as a constant in the source code, then that is its storage place. Otherwise, it is stored in string memory at the end of RAM.

Arrays can have from 1 to 11 indices. While only integer indices make sense, the interpreter will accept non-integers, by truncating them. A(I,J,K) has 3 indices, and XZ(R)has one. The indices take on values zero 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 10 (so the index is allowed to take on the 11 values 0 through 10). 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, and all elements are set to zero. 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 maximum size)
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. Arrays are stored in the order they are first encountered (in a DIM statement or by use) in the program, regardless whether they may be string or numerical arrays.

BUGS AND FIXES

There are 2 bugs. The first may occur on a warm start. Because the stack is not initialized on a warm start, an OM ERROR may occur. To avoid this I have made a habit of hitting some key, usually P, and (RETURN), after every warm start, and accepting the error, to clear the decks.

The other bug is more serious, but only occurs in programs that have string arrays. It is called the "garbage collector" bug. The garbage collector is a routine at $B147 that is called under 2 conditions. It is always called by FRE(8). It is also called when memory fills up because the variable table growing upward in memory and the string storage growing downward from high memory have collided. Usually string memory contains a lot of abandoned strings, "garbage", so by discarding the now unused strings, some memory will be freed and the program can continue. An example of how string garbage forms is given by this program:

```
10 A$="D"
20 FOR I=1 TO 100:B$=B$+A$:NEXT
30 B$="X":GOTO 20
```

Each time B$ is redefined in line 20, the new B$ is stored in high string memory, without erasing the previously defined B$!

The bug has a simple origin. In the garbage collector routine, there is a "3" which should be a "4". Remember that the "value" of a string array is stored in 4 bytes, but only 3 are actually used. MICROSOFT must have changed its mind part way through development of the interpreter, and forgot to change the garbage collector. They have, of course, long since corrected the error and notified their customers, but OSI had already masked its ROM's and it was too late.

There are two fixes that can be tried, both published in PEEK(65) V.1, no.3. The easiest fix comes from Mark Minasi. Simply pick the dimension of each string array to be 3*(any integer)+2 This often works and is usually no hardship because there will be such a number near any desired array size. The other fix is complete, and was given by Stan Murphy. It consists of changing the 3 to a 4, but requires moving the whole garbage collector routine to RAM. The following program does this. It takes up 261 bytes of RAM. (You need not reserve this at cold start time. The pointer to the end of BASIC memory is automatically adjusted.) The garbage collector is called by the statement X=USR(X), and must be called often enough to prevent the "real" flawed garbage routine from being automatically called into action.

```
1 REM      *** GARBAGE COLLECTOR ***
2 REM
100 REM    *** DRIVER ***
101 REM
107 PRINT FRE(8)
108 GOSUB 9800
109 GOSUB 9850
115 PRINT FRE(8):REM L$ HASN'T BEEN DEFINED YET
116 GOSUB 9860
120 GOSUB 500
125 GOSUB 9860
126 PRINT FRE(8):REM HANGS BECAUSE OF L$ FROM LINE 520
130 END
500 REM    *** GARBAGE MAKER ***
502 REM
504 REM         By Stan Murphy
506 REM
510 INPUT Q,K:REM TRY 20,26
520 DIM L$(Q)
530 FOR I=1 TO Q
540 FOR J=1 TO K: L$(I)=L$(I)+CHR$(64+J)
550 NEXT J
555 X=USR(X)
570 PRINT L$(I),I:NEXT I
599 RETURN
1000 REM
9800 REM    *** GARBAGE COLLECTOR ***
9801 REM
9802 REM         By Robert Badger, PEEK(65) V.1, no.8, p.20
9803 REM         after Stan Murphy,PEEK(65) V.1, no.3, p.4
9805 REM
9806 REM Note: Uses up 261 bytes EACH time it is called!
9808 REM
9810 L=PEEK(134)*256+PEEK(133)-262:GH=INT(L/256):GL=L-256*GH
9815 POKE 11,GL:POKE 133,GL:POKE 12,GH:POKE 134,GH
9820 FOR I=0TO261:M=PEEK(I+45383):POKE I+L,M:NEXT I
9825 POKE L+67,4:POKE L+216,2:POKE L+217,24:FOR I=1TO5:READ AD,M:M=M+L
9830 AD=AD+L:POKE AD,INT(M/256):POKE AD-1,M-INT(M/256)*256:NEXT I
9835 DATA 59,140,34,146,84,209,137,146,261,4
9840 PRINT "GARBAGE COLLECTOR LOCATED AT"L"GH"GH"GL"GL
9845 RETURN
9850 DEF FNF(I)=PEEK(129)-PEEK(127)+(PEEK(130)-PEEK(128))*256
9851 RETURN:REM THIS INITIALIZES THE "FRE" FUNCTION
9860 PRINT FNF(I)"BYTES FREE":RETURN:REM "FRE" FUNCTION
```

SPEED, SPACE, AND CLARITY

As your programming skills grow and you tackle more demanding tasks, you begin to encounter failures of three types: the program runs too slowly, takes up too much memory or becomes so complex and unwieldly that you lose comprehension of what you have done. Here is a unified scheme to tackle all these problems at once, making an optimum compromise between the conflicting requirements of clarity on one hand and space on the other.

First speed, since it is the key to the whole scheme. The central results of the timing tests I published in kilobaud MICROCOMPUTING (November 1980, p. 128) are clear. The two procedures most responsible for the slow running of unsophisticated BASIC programs are:

1) Conversion of decimal constants to floating point binary numbers.
2) Searching for the target lines of GOTO's, GOSUB's, etc.

Either of these procedures can be very costly if repeatedly performed in loops, especially in the intermost loops of a nested set of loops.

Converting decimal constants to floating point binary numbers takes about 1.1 ms per digit. Note the difference in the running times of these two (crude) screen clear programs:

```
10 FOR I=0 to 2047            5 Q=53248:B=65
20 POKE 53248+I,65            10 FORA=QTOQ+2047:POKEA,B:NEXT
30 NEXT
25 seconds                    8 seconds running time
```

(Actually, they fill the screen with the letter "A".)

The cure is to assign variable names to all long constants during the initialization phase of the program. E.g. KYBD=57088. In extreme cases, even one digit constants should be declared as variables, e.g. N0=0, N1=1, ... N9=9.

The target line numbers in GOTO and GOSUB statements must be converted to 16 bit integers at each encounter, so it takes a little longer (0.2 ms /digit) to process GOTO 25000 than GOTO 5. This is one reason to put "popular" subroutines at low program line numbers. The other reason is more important. A search for a line starts at the beginning of source code and requires 0.85 ms per line inspected. Lines numbered 2 to 9 would be best, if the routines are short enough.

It then follows that initializing procedures (done once at the beginning of a program run) should be located in statements at high line numbers, since they are executed only once. This leaves middle memory for the "main loop" of the program, the one where the main logic is blocked out and which makes frequent calls to the "popular" subroutines at low line numbers and infrequent calls to subroutines at high line numbers.

So much for speed, now clarity. The initialization code should contain many REM's, should explain variable names, and should give an outline of the operation of the program. It also helps clarify things if all the programs you write have a similar format. Start all new logical sections on "even hundreds" line numbers and always start the main loop at 100 and the initialization at 1000. These numbers may sound a little low to those of you used to renumbering each program with an interval of 10 between lines, no matter how large the numbers may get. But remember the conversion time required to process target line numbers! Small line numbers are best and so I space my lines 2 to 5 numbers apart.

All this suggests a standard format, given below. The format adds to clarity and ease of writing by including (at standard line numbers and with standard variable names) those utilities that are used again and again, such as rapid screen clear, keyboard POKE and screen corner addresses, score writing subroutines, etc. I put utilities in lines 9000to 9999, and tape the whole format. Then when starting to write a new program, I just read in the format, and begin to add code (and drop unwanted lines of the format).

DEBUGGING AND UTILITIES

Effort spent in learning to use the available facilities and in developing some utilities will enable you to perform your debugging chores efficiently. The resources are divided into three classes.

Editor: While RUNning your program, it may stop because you hit (CTRL/C), or the program reached a STOP, END, or ERROR IN... . Then you are back in the immediate mode, wondering what happened. Take your time and think it through. To clarify things, you can print out variable values singly, or with one line programs (no line number!) to display arrays. You can alter variable values with these one liners, and do any variety of LISTings. You can poke around and think as much as is necessary, just so long as you do not add, delete or change any numbered lines (which would destroy the variable table.) When all is set, you can use CONT to continue the program from where it stopped, or use GOTO ... or GOSUB ... to start elsewhere, and still preserve the variable table created by the running of the program up to the present moment. However, if you alter, add or delete any lines, your only choice is to start again from the beginning.

Insertions:While building a program, you may insert STOP or PRINT... statements to help pinpoint program malfunctioning. You may also want to insert some FOR I=1 TO 5ØØØ:NEXT delay loops to slow down the program for better observation of its functioning. After the trouble is fixed, you remove these diagnostic tools.

Utilities: A package of short BASIC programs can be put into high line numbers and used during program construction and debugging. They need not be included in the tape of the final product. Some useful ones are:

Hex to decimal
Decimal to hex
Line renumber
Tape view
Screen dump (if you have a printer)

Branch locator
Variable cross reference table generator

The most useful renumber program will allow you to renumber one or a few lines without changing the rest of the program. Tape view is useful to display another BASIC program on the screen so you can see what you did, without overwriting your current program in memory. Branch locator is useful to pinpoint those lines targeted by GOTO's and GOSUB's. Also it helps unravel the structure of foreign programs that swim into your possession. Likewise, a Variable Cross Reference table pinpoints variable usage and variable mispelling and is necessary if you are going to condense code by reusing variable names in a long program.

PROGRAM FORMAT AND UTILITY PROGRAMS

```
1 GOTO 1000:REM *** PROGRAM NAME ***
2 REM Remove all free standing REM's in lines
3 REM 2 TO 999.
4 REM
5 REM "Popular" subroutines in lines 2-99.
6 REM
100 REM MAIN LOOP IN LINES 100 TO 999
999 STOP
1000 REM
1001 REM *** PROGRAM NAME ***
1002 REM
1003 REM     Edward H. Carlson
1004 REM     3872 Raleigh Dr.
1005 REM     Okemos MI 48864
1006 REM     (517) 349-1219
1007 REM
1100 KB=57088:REM KEYBOARD
1105 SC=53248:REM SCREEN CORNER
8999 GOTO 100
9000 REM
9001 REM    *** MENU ***
9002 REM
9007 PRINT:PRINT:PRINT:PRINT
9009 PRINT "9000  MENU"
9010 PRINT "9100  RAPID SCREEN CLEAR"
9012 PRINT "9200  PRINT AT"
9015 PRINT "9310  DECIMAL TO HEX"
9020 PRINT "9410  HEX TO DECIMAL"
9030 PRINT "9500  ERROR CODE FIX"
9035 PRINT "9600  SCREEN DUMP"
9037 PRINT "9700  BELL"
9038 PRINT "9800  RANDOM NUMBER GENERATOR
9040 PRINT "9900  LINE RENUMBER"
9060 PRINT "61000 CROSS REFERENCE GENERATOR"
9062 PRINT "62000 BRANCH LOCATOR"
9099 PRINT:PRINT:PRINT:STOP
```

```
9100 REM
9101 REM   *** RAPID SCREEN CLEAR ***
9102 REM
9103 REM       From kilobaud MICROCOMPUTING somewhere
9104 REM
9110 AA=PEEK(129):BB=PEEK(130):POKE 129,255:POKE 130,215
9112 D$="                                              "
9114 FOR I=1 TO 35:D$=D$+"":NEXT:POKE 129,AA:POKE 130,BB:RETURN
9119 PRINT "9000  MENU"
9200 REM
9201 REM   *** PRINT AT ***
9202 REM
9203 REM       Roger Olsen, Aardvark Catalog
9204 REM
9210 FORY=1TOLEN(D$):POKED+Y,ASC(MID$(D$,Y,1)):NEXTY:RETURN
9300 REM
9301 REM   *** DECIMAL TO HEX ***
9302 REM
9304 INPUT"DECIMAL NUMBER";N:GOSUB 9310:PRINT D$:GOTO 9300
9310 G$="0123456789ABCDEF":IF N>65535 THEN PRINT"ERROR"


9312 D$="":F=4096:FOR I=1 TO 4:N1=INT(N/F)
9314 N=N-N1*F:D$=D$+MID$(G$,N1+1,1):F=F/16:NEXT I:RETURN
9400 REM
9401 REM   *** HEX TO DECIMAL ***
9402 REM
9405 INPUT"HEX 4 DIGIT NUMBER";D$:GOSUB 9410:PRINT D$:GOTO 9400
9410 N=0:L=4096:FORI=1TO4
9415 M=ASC(MID$(D$,I,1))-48
9420 IFM>9THEN M=M-7
9425 N=N+M*L:L=L/16:NEXT:D$=STR$(N)
9430 RETURN
9500 REM
9501 REM *** ERROR MESSAGE CORRECTOR ***
9502 REM
9504 REM     E.D. Morris Jr. and Tim Finkbeiner
9505 REM     MICRO Nov. 1980, p. 30:37
9506 REM
9520 DATA 72,173
9530 DATA 64,215:REM SUPERBOARD 101,211
9540 DATA 201,63,208,8,173
9550 DATA 66,215:REM SUPERBOARD 103,211
9560 DATA 41,127,141
9570 DATA 66,215:REM SUPERBOARD 103,211
9580 DATA 104,76,195,168,0,0
9590 FORX=576 TO 597
9592 READ Q:POKE X,Q:NEXT
9594 POKE 4,64:POKE 5,2:END
9600 REM
9601 REM   *** SCREEN DUMP ***
9602 REM
9603 REM USEFUL IF YOU HAVE A PRINTER, BUT WILL DEPEND
9604 REM ON YOUR PARTICULAR MACHINE.
9605 REM
```

```
9700 REM
9701 REM   *** BELL ***
9702 REM
9703 REM I have added a speaker to my C2-4P.
9705 FOR I=0 TO 200:POKE AC,0:POKE AC,255:NEXT I:RETURN
9706 POKE AC,0:POKE AC,255
9708 NEXT I:END
9900 REM
9901 REM   *** LINE RENUMBER ***
9902 REM
9910 INPUT"FROM ... TO";NF,NT
9915 NH=INT(NT/256):NL=NT-NH*256
9920 FOR I=768TO40000:B=PEEK(I):IF B<>0 THEN NEXT I
9925 N=PEEK(I+3)+PEEK(I+4)*256
9926 PRINTCHR$(13)N;
9930 IF N=NF THEN POKE I+3,NL:POKE I+4,NH:END
9940 IF N>9999 THEN END
9945 I=I+4:NEXT I
```

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 the immediate mode and then a SAVE, LIST to put the program on tape. The code on tape is ASCII (no tokens) which we here represent in decimal numbers.

```
13  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 10  0  0  32 50 32 66 66 66 66 66
13  0  0  0  0  0  0  0  0  0  0
```

where 10	is line feed
32	space
13	return (or carriage return CR)
49	1
50	2
65	A
66	B

The two nulls after the 10 (line feed) are the work of the NULL command. Default is zero nulls. Each line begins with a CR and ten nulls (see support ROM at $FF7B) followed by a line feed and the text. An empty line is sent before the BASIC program code starts.

The OSI system differs from some others in that you can add a program to one already in the machine by reading it in from tape. Of course no line numbers can be the same in the two programs, or more exactly, all the line numbers of one must be above all the line numbers of the other, so that the flow of execution cannot get mixed between them.

The tape port address of a C2 or C4P is at \$FCØØ=64512, and for a C1 or superboard II is at \$FØØØ=61440. You might want to read your BASIC tapes with a program like this:

```
1 Q=64512:R=Q+1
4 WAIT Q,1
5 PRINT PEEK(R):GOTO 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 a little when making the tape, more later.

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

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

When you get an error break because you tried to fill D(201), you can enter this line in immediate mode to see the output.

```
FOR I=1 TO 2ØØ:PRINT D(I);:NEXT
```

The problem here is that the first part of D may be filled with noise characters from the "blank" tape. You may have trouble deciding where the taped program starts.

If you want to store some data generated by a program onto 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.

```
10 REM    *** PROGRAM TO GENERATE DATA AND SAVE IT ***
15 REM
20 DIM Y(20):FOR I=1 TO 20:Y(I)=I:NEXT
30 SAVE:FORI=1 TO 5:PRINT 0:NEXT:PRINT 255:REM LEADER
40 FOR I=1 TO 20:PRINT Y(I):NEXT
60 LOAD:REM TO EXIT FROM SAVE
65 PRINT "HIT (SPACE BAR) TO UNLOCK KEYBOARD"
70 END
```

```
1000 REM    *** PROGRAM TO READ TAPE ***
1001 REM
1005 DIM Y(20):LOAD
1010 INPUT X:IF X<>0 THEN 1010
1020 INPUT X:IF X=0 THEN 1020
1030 FOR I=1 TO 20:INPUT Y(I):NEXT
1040 PRINT "HIT SPACE BAR TO CONTINUE"
1050 FOR I=1 TO 20:PRINT Y(I);:NEXT
9999 END
```

And here is a program to read the data generated. Both programs can be in the machine at once. To write to tape do RUN. To read from tape do RUN 1ØØØ. Line 30 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 60 is to get 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=1 TO 2ØØ: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. Without line 6 it runs at full speed and can be read by the second program 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 fromOSI use the autoload format. Each byte to be sent is broken down into the two ASCII characters that represent it in hexadecimal notation. For example if %1111ØØ11 is the form stored, it is sent as 2 bytes F and 3, in ASCII as $46 and $33. Thus 1 byte in memory is recorded as 3 bytes on tape. This 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 auto-start 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 or some other program, or the warm start of BASIC (ØØØØ) or the monitor (FEØØ). 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, MSB (most significant byte) byte first. The right most HL HL is the address at which the monitor will start execution, if G is found on the tape (or entered from the keyboard). This format is exactly the same that you would use from the keyboard to enter and run a program.

The monitor in the OSI machines can read tape in the above format, but cannot write tapes. To write such tapes, use a program like the one below, which assumes your machine language code is in memory from $0222 to 02FF.

```
1 REM WRITE MACHINE LANGUAGE TAPES IN OSI FORMAT
2 REM        E. H. CARLSON
3 REM        3872 RALEIGH DR.
4 REM        OKEMOS MI 48864
5 REM        COMPUTE Issue 3, March/April 1980, p.115
6 N=221:M=3*N+15
7 Q=64512:R=Q+1
8 REM ACIA AT 64512=$FCOO IN 500 BOARD MACHINES
9 REM USE 61440=$F000 FOR 600 BOARD MACHINES
10 INPUT "START TAPE AND WAIT FOR LEADER, THEN INPUT G ";A$
100 DATA 46,48,50,50,50,47:REM .0222/
105 DATA 46,70,69,48,48,71:REM .FE00G
110 FOR I=1 TO 6:READ C:WAIT Q,2:POKE R,C:PRINT CHR$(C);:NEXT
116 S=546:E=S+N
119 REM FOR I=$0222 TO $02FF
120 FOR I=S TO E
125 C=PEEK(I):H=C AND 240:L=C AND 15
130 H=H/16+48:IF H>57 THEN H=H+7
135 L=L+48:IF L>57 THEN L=L+7
136 WAIT Q,2:POKE R,H
137 WAIT Q,2:POKE R,L
138 WAIT Q,2:POKE R,13
145 PRINT CHR$(H);CHR$(L);" ";
150 NEXT I
155 FOR I=1 TO 6:READ C:WAIT Q,2:POKE R,C:PRINT CHR$(C);:NEXT
160 REM FORMAT FOR TAPES IS:
165 REM .HLHL/HLRHLR...HLR.HLHLG
170 REM WHERE THE HLHL AT THE START IS THE STARTING ADDRESS,
175 REM HI BYTE FIRST, THE HLHL AT THE END IS THE EXECUTE
180 REM ADDRESS AND THE HLR'S IN THE MIDDLE ARE THE TEXT
185 REM BYTES, THE R BEING A CARRIAGE RETURN
190 REM THE . / G ARE THE SAME AS THE COMMANDS IN THE MONITOR
200 REM THE H AND THE L ARE ASCII CODE FOR THE HEX DIGITS
205 REM 0 THROUGH F.
```

```
1 GOTO 62000:REM *** BRANCH LOCATOR ***
100 REM
102 REM   *** TEST PROGRAM ***
103 REM
110 GOTO 500
120 GOSUB 510
122 ON A GOTO 52`,530
124 ON A GOSUB 540,550
126 IF A THEN 560
128 IF A GOTO 570
130 IF A THEN GOSUB 580
132 IF A THEN B=1
133 REM LOCATOR FINDS "THEN" BUT PRINTS NO ADDRESS
134 IF A THEN GOTO 590
136 REM GOTO 0
138 REM GOSUB 0
140 REM IF A THEN GOTO 0
142 IF A THEN GOSUB 600:GOSUB 610:GOTO 620
999 STOP
9700 RETURN:MY MACHINE HAS A BELL PROGRAM HERE
62000 REM
62001 REM   *** BRANCH LOCATOR ***
62002 REM
62010 PRINT:PRINT:PRINT "BRANCHES:":PRINT:PRINT
62020 A=772:L=0:FOR I=1 TO 9999:REM START HERE FOR NEW LINE
62035 L=PEEK(A-1)+PEEK(A)*256:PRINT CHR$(13) L;
62036 IF L>9999 THEN GOSUB 9700:END
62040 FOR J=1 TO 9999:A=A+1:B=PEEK(A):REM NEW STATEMENT
62050 IF B=136 OR B=138 OR B=140 OR B=144 THEN 62100
62055 FOR K=1 TO 255:REM LOOK FOR STATEMENT OR LINE END
62060 A=A+1:B=PEEK(A):IF B=0 THEN A=A+4:PL=0:NEXT I
62065 IF B=58 THEN PL=1:NEXT J
62070 NEXT K:STOP
62100 FOR K=1 TO 73:B=PEEK(A)
62110 IF B=136 THEN D$="GOTO     ":GOTO 62143
62120 IF B=140 THEN D$="GOSUB    ":GOTO 62143
62130 IF B=160 THEN D$="THEN     ":TH=-1:GOTO 62143
62141 A=A+1:NEXT K:STOP
62143 IF PL=1 THEN PL=0:PRINT CHR$(13) L;
62144 PRINT TAB(7);D$;
62145 A=A+1:B=PEEK(A):IF B=32 THEN PRINT "";:GOTO 62145
62147 IF TH THEN 62200:REM LOOK FOR COMPLICATED "THEN" LINES
62150 IF B=44 OR (B>46 AND B<58) THEN PRINT CHR$(B);:GOTO 62145
62152 PRINT ""
62155 IF B=0 THEN A=A+4:PL=0:NEXT I
62160 IF B=58 THEN PL=1:NEXT J
62165 GOTO 62055
62200 TH=0:IF B=136 OR B=138 OR B=140 THEN 62110
62210 GOTO 62150
OK
```

```
1 A=1:REM *** TEST PROGRAM ***
2 REM "RUN 62000" TO COMPACT THE TEST PROGRAM
3 : ::C=3:D=4:REM AAAAA
4 END:DON'T SEE THIS AFTER COMPACTION
5 RETURN:NOR THIS
6 GOTO 11111:NOR THIS
7 A$="SEE THIS":REM NOT THIS
999 STOP
62000 REM
62001 REM   *** COMPACTOR ***
62002 REM
62010 PRINT:PRINT:PRINT "COMPACTING":PRINT:PRINT
62015 DIM L(80):AP=769:AD=3*256-3
62020 A=768:L=0:FOR I=1 TO 9999:A=A+4
62025 IF L<>0 THEN GOSUB 62600
62035 L=PEEK(A-1)+PEEK(A)*256:AN=0
62036 IF L>9999 THEN POKE AP,0:POKE AP+1,0:END
62040 A=A+1:B=PEEK(A):IF (B=32)OR(B=58) THEN 62040
62050 A=A-1:FOR K=1 TO 255:A=A+1:B=PEEK(A)
62060 IF B=0 THEN NEXT I
62065 IF B=142 THEN GOTO 62100
62068 IF (B=128)OR(B=143)OR(B=141) THEN L(AN)=B:AN=AN+1:GOTO 62100
62070 IF B=58 THEN GOTO 62400
62073 IF B<>32 THEN L(AN)=B:AN=AN+1
62075 IF B=136 THEN GOTO 62200
62080 IF B=34 THEN GOTO 62300
62090 NEXT K:STOP
62100 FOR K=1 TO 255:A=A+1:B=PEEK(A):REM LOOKING FOR LINE END
62110 IF B=0 THEN NEXT I
62120 NEXT K
62200 FOR K=1 TO 255:A=A+1:B=PEEK(A):REM FOUND "GOTO"
62210 IF B=0 THEN NEXT I
62215 IF B=32 THEN A=A+1:B=PEEK(A):GOTO62210
62220 IF B=58 THEN GOTO 62100
62225 L(AN)=B:AN=AN+1:NEXT K
62300 FOR K=1 TO 255:A=A+1:B=PEEK(A):REM FOUND " CHAR.
62320 IF B=34 THEN L(AN)=B:AN=AN+1:GOTO 62090
62325 IF B=0 THEN NEXT I
62327 IF B=58 THEN 62400
62330 L(AN)=B:AN=AN+1:NEXT K
62400 A=A+1:B=PEEK(A):IF (B=32)OR(B=58) THEN 62400:REM FOUND :
62410 IF B=0 THEN NEXT I
62420 IF B=142 THEN GOTO 62100
62430 L(AN)=58:L(AN+1)=B:AN=AN+2:GOTO62120
62600 PRINT L;:REM POKE MEMORY WITH COMPACTED LINE
62601 AH=INT((A-3)/256):AL=(A-3)-256*AH
62602 POKE AP,AL:POKE AP+1,AH:PRINT TAB(8) AL;AH;
62604 IF AN=0 THEN PRINT:RETURN
62605 AH=INT(AP/256):AL=AP-256*AH
62607 PRINT TAB(16) AL;AH;
62608 POKE AD,AL:POKE AD+1,AH:AD=AP:AP=AP+2
62610 AH=INT(L/256):AL=L-256*AH
62611 POKE AP,AL:AP=AP+1:POKE AP,AH:AP=AP+1
62616 FOR I=0 TO AN-1:POKE AP,L(I):PRINT CHR$(L(I));:AP=AP+1:NEXT I
62620 POKE AP,0:AP=AP+1:PRINT:RETURN
```

HOOKS INTO BASIC and BASIC TRACE

After you have been using your machine for a while, a case of "whatifcitis" sets in. To overcome some of the minor annoyances or to make some major extensions to BASIC, you must seek out the spots where BASIC protrudes from its fortress in the ROM's. There are several such places.

Of course, USR(X) is designed to be an exit from BASIC. But there are others that lead even deeper into the fortress. BASIC passes through the JMP in $0000 on its way to warm start at $A274. Change the address in $01,02 and you can make "warm start" into anything you wish! For example, write your own screen editor with true backspace and middle-of-the-line editing. Or buy one in firmware or software offered by the software houses. Other jump pointers in zero page are the message printer at $04, INVAR at $06, and OUTVAR at $08. Superboards and C1 machines have a very useful set of hooks in page $02 for INPUT, OUTPUT, (CTRL/C), and LOAD FLAG.

There is one gigantic crack that extends to the very center of fortress BASIC. The routine stored in page zero from $BC to $D3 gets characters from the BASIC source code lines and sends them on to be processed by the rest of the interpreter. Every character of every line of BASIC source code goes through this routine! I wrote an article "PUT YOUR HOOKS INTO OSI BASIC" about it (MICRO, June 1980, page 15). Dale Mayers has written a BASIC TRACE program by modifying the page $00 routine and adding code in page $02. A version of this program is given below.

```
10 REM    *** BASIC TRACE ***
12 REM
15 REM        by Dale Mayers
16 REM        2301 S. Washington
17 REM        Lansing MI
20 REM
100 FORX=546TO642:READD:POKEX,D:NEXT
105 FORX= 218 TO 238 :READD:POKEX,D:NEXT
106 REM CODE STARTING AT $0222
107 DATA132,247,134,248,162,0,181,172
110 DATA149,240,232,224,5,208,247,165,136,166,135,133
120 DATA173,134,174,134,239,162,144,56,32,232,183,32
130 DATA110,185,162,0,189,0,1,201,0,208,2,240
                 163
140 DATA27,157,128,215,232,224,6,208,239,32,0,253
150 DATA162,0,181,240,149,172,232,224,5,208,247,164
                                   208
160 DATA247,166,248,96,169,32,157,128,215,232,224,6
170 DATA208,248,240,225,162,176,134,206,162,10,134,207
180 DATA96,32,2,2,2
200 REM CODE STARTING AT $00DA
210 DATA133,238,165,135,197,239,240,3,32,34,2,165
220 DATA238,56,233,48,56,233,208,96,32,32,85,21
225 REM LOADS CODE $B0, $A0 INTO ADDRESSES $00CE AND $00CF
226 REM USING A SUBROUTINE THAT STARTS AT $0276.
230 POKE11,118:POKE12,2:X=USR(X)
300 REM RUN THIS PROGRAM.  THEN "NEW" AND LOAD YOUR CODE.
310 REM THE CURRENT LINE NUMBER WILL APPEAR AT THE
315 REM BOTTOM OF THE SCREEN AS YOUR PROGRAM RUNS.
320 REM YOUR PROGRAM WILL RUN WHILE THE SPACE BAR IS
325 REM HELD DOWN, STOP WHEN THE SPACE BAR IS RELEASED.
```

For C1 and Superboard II
line 140 change 128 to 163
line 160 change 215 to 208
This changes the address
on the screen.

KEYBOARD AND SCREEN TRICKS

Good programs have optimum human-machine interfacing. Whether you run a word processing, game, or business program, you quickly become fatigued and annoyed if the keyboard requires unnecessary pounding or the TV screen displays inappropriate stuff.

The PRINT and INPUT commands of BASIC, while easy to use, promote idiotic repetitive and mechanical conversation. Humans feel most at home if the computer mimics human conversation patterns.. For example, instructions at the start, menus, HELP if needed, complete prompts for early use, and minimal prompts when familiarity with the software system has been reached. All this takes some extra effort by the programmer. Rather than pontificate on the principles of good human-machine interfacing, I will just point out some keyboard and screen techniques that are useful. With them, you can obtain clean input and output if you give some thought to the process and turn your annoyance detectors up high as you try out your programs during their development.

Scroll free displays. The most primitive displays use a succession of PRINT statements so that old material is scrolled upward. Information entered does not stay where you put it, requiring you to search upward on a cluttered screen to find the nuggets you need. Perhaps the worst cases of "scrollitis" occur in those board games where the whole board is rePRINTed after every move. The resulting scrolling is visually equivalent to the nerve jarring racket of a stick rattling along a picket fence. The best way is to create the board and subsequently update it with POKEs. Scores and other text can be POKEd in with the "print at" subroutine given in the section on FORMAT and UTILITIES.

We have this scroll free gem from the Aardvark Journal:

```
120 PRINT CHR$(13)"message";
```

The CHR$(13) and the semicolon at the end are the essential elements of the trick. The message is printed at the usual entry spot at the bottom of the screen. But the semicolon insures that no scroll follows the message, and the CHR$(13) sends a CR before the message so it starts at the left of the screen rather than at the end of the previous screen output.

Invisible tagging of spaces. In programs, the screen display itself can be data, deposited in screen memory by POKEs and retrieved by PEEKs. There are 2 distinct characters, $20 and $90, that are displayed as a "blank" on the screen. This fact allows some unusual effects to be programmed. For example, in a "fox and rabbit" game, the field may consist of type $20 blanks (plus trees, houses, fences, rabbit,fox, etc.) and as the rabbit moves, he may lay down a trail of type $90 blanks, invisible on the screen but followed by the fox, sniffing with PEEKs and using IF... to recognize the $90 scent.

<u>Keyboard</u> <u>input</u>. There are three ways to get input from the keyboard, or rather, one hardware way that can be used directly or accessed through 1 or 2 levels of software.

The hardware method uses the keyboard port at 57088. This method differs depending on whether you have a C1 or a C2 (C4P) machine. At any rate, it is described in the OSI literature. The only point I will make here is that the AND, OR, NOT functions are very useful to detect if one key is depressed when others may or may not also be depressed.

```
1 REM FOR A C2-4P
100 KB=57088
110 POKE 530,1:REM DISABLE (CTRL/C)
115 S$=""
120 POKE KB,4:REM 4=%00000100, activates R2
130 P=PEEK(KB)
140 IF(P AND 16)=16 THEN S$="B"
150 REM 16=%00010000, C4
160 PRINT P;S$
199 GOTO 115
200 REM Detects a B key depression, even if
        other row R2 keys (XCVBNM,) are
        depressed also.
```

On a C2 machine, the row and columns are designated by bits being equal to 1. On a C1 machine, by bits being zero. For example:

```
C2:  R2=%00000100
C1:  R2=%11111011
```

So the NOT operator can be used to translate from variables suitable for a C2 machine to those for a C1. Software can be written that works on either machine. Write the program for one (say a C2-4P) and have the program look at the byte in $FFE2 to see if the machine being used is a C1. If so, do a NOT on the keyboard variables.

The next level of use of the keyboard from BASIC has USR(X) call the keyboard routine at $FD00 directly. This routine goes into a loop waiting for a key closure. Upon getting one, it stores the character at address 531 and returns to BASIC.

```
From the            1ØØ POKE 11,Ø;POKE 12,253
Aardvark Journal:   11Ø X=USR(X)
                    12Ø P=PEEK(531)
                    13Ø PRINT CHR$(P);
                    14Ø REM Or use P to make a string, etc.
                    199 GOTO 110
```

There are advantages to this input over using the INPUT command in the handling of commas and quotation marks.

INPUT: some problems and partial solutions. If you are entering a string, the computer usually interprets commas as marking the end of the string. This is unacceptable in many applications— for example, in word processing programs. Example:

```
10 INPUT S$
20 PRINT S$
RUN
? HERE, WE HAVE A COMMA.
?EXTRA IGNORED
HERE
```

A fix is to start the inputed string with a quotation mark. Same program:

```
RUN
? "HERE, WE HAVE A COMMA.
HERE, WE HAVE A COMMA.
```

However, there is a price. The program now will take a second quote as sufficient cause to be confused. Note the same program with two more input sentences:

```
RUN
? THIS IS A " MARK.
THIS IS A " MARK.

RUN
? "WE WANT BOTH A, AND A " IN THE SAME LINE.
REDO FROM START
```

but

```
RUN
? THIS IS A " AND THIS IS ANOTHER ".
THIS IS A " AND THIS IS ANOTHER ".
```

but

```
RUN
?  THIS IS A " AND, THIS IS ANOTHER ".
EXTRA IGONRED
 THIS IS A " AND
```

All this makes strings a very poor way to do word processing. More accurately, a poor way to input text. Once a string is properly given a quotation mark, it treats it right from then on. Example:

```
10 Q$=CHR$(34)
20 S$="YOU CAN HAVE , AND " +Q$+"MARKS IN THE"
30 PRINT S$
RUN
YOU CAN HAVE , AND "MARKS IN THE
```

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 two's complement form. Example:

4	%0000000000000100
3	0000000000000011
2	0000000000000010
1	0000000000000001
0	0000000000000000
-1	1111111111111111
-2	1111111111111110
-3	1111111111111101
-4	1111111111111100

To get the negative of any number (+ or -) when in the two's complement integer form, first invert each digit (every 1 goes to 0 and 0 to 1). Then add 1 (with binary carry).

```
Example:        3   %0000000000000011
               -3    1111111111111100+1=
                     1111111111111101

               -4    1111111111111100
                4    0000000000000011+1=
                     0000000000000100
```

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 (fraction) and sign. For example the number 3 is represented as

3 = %0011   in one binary nybble.

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

3 = %0000 0011   in one byte

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

$3 = \%0.11 \times 2^{+2}$ in analogy with

$3 = 0.3 \times 10^{+1}$

So the internal representation of 3 <u>could</u> look like this;

$$\underbrace{3 = \$02}_{\text{exponent}}\ \underbrace{\%1100\ 0000\ \$00\ 00}_{\text{3 byte mantissa}}\ \text{but } \underline{\text{doesn't}}\text{, quite.}$$

We have neglected two details. We want to be able to express both positive and negative exponents, so the byte representing the exponent is biased by adding \$80 to it. The exponent +2 is represented by \$82, zero by \$80 and -2 by \$7E.

Also, we want to represent the sign of the number, +3 and not -3. We make use of the fact that the mantissa is chosen such that its left most digit is always 1. So this digit is redundant and we remove it and replace it with a sign digit, 0 for + and 1 for -. The final result is:

```
                          3 = %11
is stored as              3 = $82 %0100 0000 $00 00 = $82 40 00 00
while                    -3 = $82 %1100 0000 $00 00 = $82 C0 00 00
and                     1/3 = $7F %0010 1010 1010 1010 1010 1011
and                    -1/3 = $7F %1010 1010 1010 1010 1010 1011

finally:                  0 = $00 00 00 00 as a convention
```

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

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

$$= \%1111\ 1111\ 1111\ 1111\ 1111\ 1111$$

```
       stored as               $98 7F FF FF in the table.

       shown as                1.6772E+07    on the screen.
```

Finally, what happens if you try to store an undefined value? The 2 line program

```
1 A=B
2 PRINT A;B
RUN
 Ø  Ø
```

run ok. The variable B, of course, is undefined in this program and has no entry in the variable table. A is represented by

A = \$00 00 A5 7D in the table.

This number is treated as being zero by the BASIC interpreter. In fact, any floating point number whose exponent is \$00 is treated as zero. If the sign bit in the mantissa is set, the number is treated an -Ø.

TOKENS

80	128	END	A3	163	+
81	129	FOR	A4	164	-
82	130	NEXT	A5	165	*
83	131	DATA	A6	166	/
84	132	INPUT	A7	167	(power) ^
85	133	DIM	A8	168	AND
86	134	READ	A9	169	OR
87	135	LET	AA	170	>
88	136	GOTO	AB	171	=
89	137	RUN	AC	172	<
8A	138	IF	AD	173	SGN
8B	139	RESTORE	AE	174	INT
8C	140	GOSUB	AF	175	ABS
8D	141	RETURN	B0	176	USR
8E	142	REM	B1	177	FRE
8F	143	STOP	B2	178	POS
90	144	ON	B3	179	SQR
91	145	NULL	B4	180	RND
92	146	WAIT	B5	181	LOG
93	147	LOAD	B6	182	EXP
94	148	SAVE	B7	183	COS
95	149	DEF	B8	184	SIN
96	150	POKE	B9	185	TAN
97	151	PRINT	BA	186	ATN
98	152	CONT	BB	187	PEEK
99	153	LIST	BC	188	LEN
9A	154	CLEAR	BD	189	STR$
9B	155	NEW	BE	190	VAL
9C	156	TAB(	BF	191	ASC
9D	157	TO	C0	192	CHR$
9E	158	FN	C1	193	LEFT$
9F	159	SPC(	C2	194	RIGHT$
A0	160	THEN	C3	195	MID$
A1	161	NOT			
A2	162	STEP			

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 }
       Ø3 } address of next line
       Ø1 }
       ØØ } line number
       41   A
       AB   token for =
       33   3 in ASCII
       ØØ   line end symbol
       ØØ }
       ØØ } when address of next line is zero, source ends.
       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
       80   $
       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:

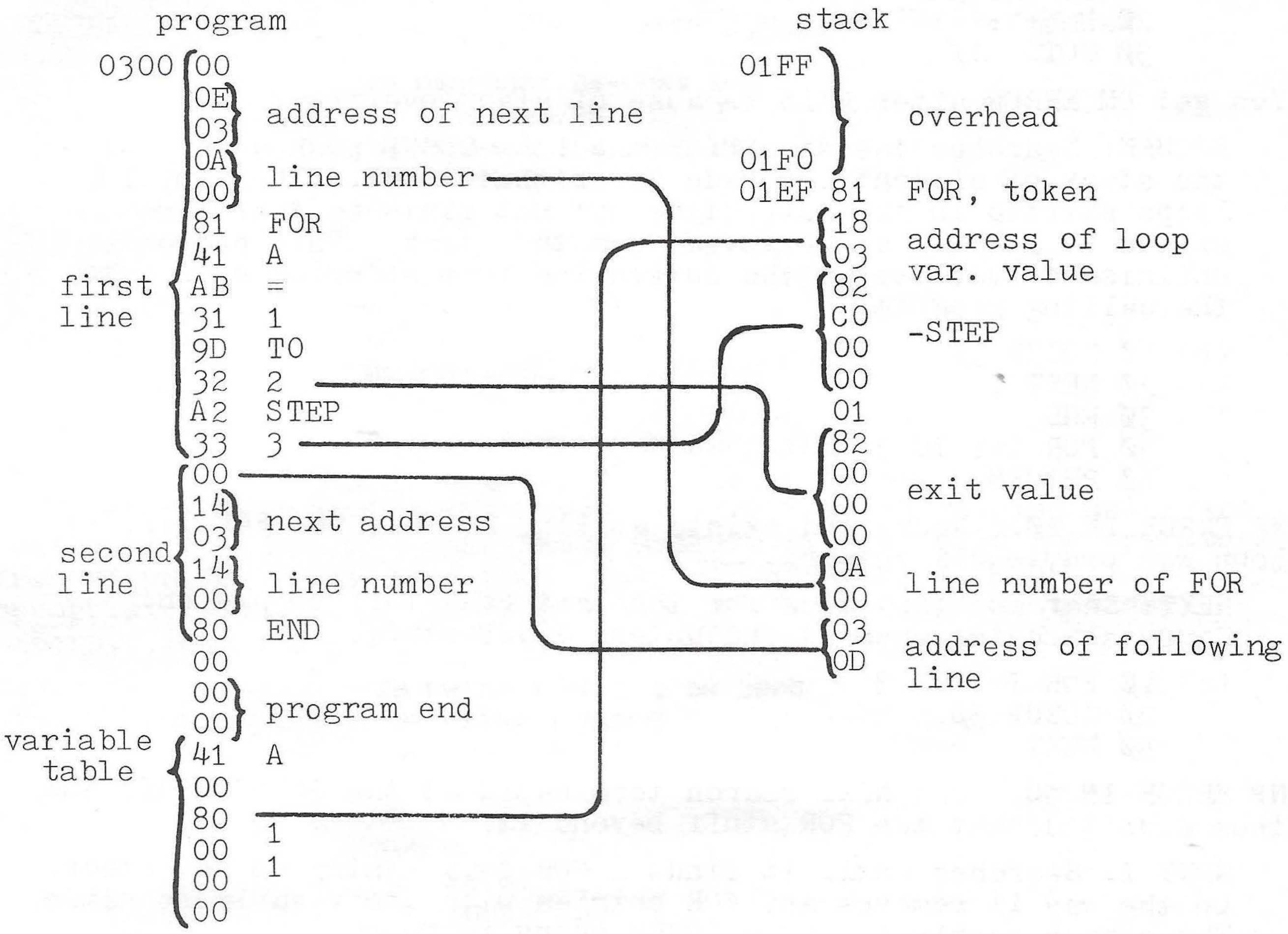
```
$Ø348 49 I                      $Ø35D  ØØ =0
      ØØ                               ØØ A(0,0)
      82 2                             ØØ
      ØØ                               ØØ
      ØØ
      ØØ                               84 =10
      4A J                             2Ø A(1,0)
      ØØ                               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                               82 =2
      Ø3 J has 3                       ØØ A(0,2)
           values                      ØØ
      ØØ                               ØØ
      Ø2 I has 2
                                       84 =12
                                       4Ø A(1,2)
                                       ØØ
                                       ØØ
                                       empty ...
```

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

THE STACK

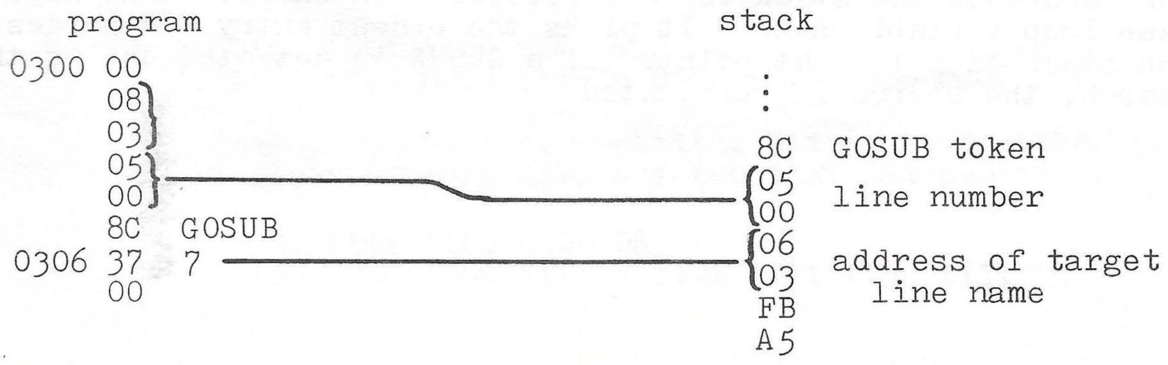
Each time the interpreter encounters a FOR... statement, it pushes some stuff on the stack. The depth of all kinds of nesting combined, (...) sets, FOR...NEXT loops, or subroutines, is limited by the stack length available. Consider this short program:

```
10 FORA=1TO2STEP3
20 END
```



The entry on the stack for subroutines is demonstrated by this little program:

```
5 GOSUB7
6 REM
7 END
```



We see that FOR pushes 16 bytes on the stack and GOSUB pushes 7 bytes. Within a line "(" pushes 5 bytes and expressions within the parentheses may push additional bytes on the stack.

Now we consider the two commands (FOR, GOSUB) that push stuff on the stack, and the three (NEXT, RETURN, FOR) that search the stack.

GOSUB: Pushes 7 bytes on the stack, does no search of the stack.

```
(a) 1Ø GOSUB 3Ø
    2Ø N=N+1:PRINT N
    3Ø GOTO  1Ø
```

You get OM ERROR after N=26 because of stack overflow.

RETURN: Searches the stack for the last GOSUB pushed on. Clears the stack of all entries made after that GOSUB. Thus any FOR loops started in the subroutine but not finished there (not exited by a NEXT) are removed from the stack. This prevents unfinished business in the subroutine from slopping over into the calling program.

```
(b) 1Ø GOSUB 5Ø
    2Ø NEXT I
    3Ø END
    5Ø FOR I=1 TO 3
    6Ø RETURN
```

NF ERROR IN 2Ø. No record exists at line 20 that the FOR I=... loop was previously entered.

NEXT: Searches the stack for the last FOR stuff pushed on. Stops searching when it encounters GOSUB stuff.

```
(c) 1Ø FOR I=1 TO 3
    2Ø GOSUB 5Ø
    5Ø NEXT
```

NF ERROR IN 5Ø. The NEXT search terminated at the GOSUB stuff and thus didn't detect the FOR stuff beyond it.

NEXT I: Searches until it finds a FOR I=... entry on the stack. On the way it removes any FOR entries with other variable names. The search terminates if a GOSUB entry is found.

```
(d) 1Ø FOR I=1 TO 3
    2Ø FOR A=1 TO 3
    3Ø NEXT I
    4Ø NEXT A
```

NF ERROR IN 4Ø. The information about the FOR A=... has been wiped from the stack by the time line 40 is reached.

FOR: Searches the stack for all previous FOR entries that have the same loop variable name. It picks the oldest entry and purges the stack back to that point. If a GOSUB is detected during the search, the search is terminated.

(e)
```
10 FOR I=1 TO 3
20 FOR A=1 TO 3
30 FOR I=1 TO 3
40 NEXT I
50 NEXT A
60 NEXT I
```

(f)
```
10 FOR I=1 TO 3
20 GO TO 10
```
loops forever

NF ERROR IN 50. Line 30 purged the stack back to the FOR stuff put on in line 10. This purging of extra entries with the same variable name permits jumping out of a loop and then re-entering it without a stack overflow. See program (f)

Again, the search terminates at a GOSUB to isolate the main program from the shenanigans in the subroutine. But this isolation cannot be complete because the stack is not the only thing altered by the FOR statement. The loop variable entry in the variable table is also initialized. The new value persists even after return from the subroutine.

(g)
```
10 FOR I=1 TO 3
20 GOSUB 50
30 PRINT I:NEXT I
40 END
50 FOR I=7 TO 9
60 RETURN
```

Runs to a normal END at line 40. But it only "loops" once, printing the number "7". The moral? Either use different loop variable names in the subroutine, or make a normal exit through NEXT in the subroutine's loop.

Some other instructive programs:

(h)
```
10 FOR I=1 TO 3
20 FOR A=1 TO 3
30 N=N+1:PRINT N;I;A
40 GOTO 10
```
loops forever

(i)
```
10 FOR I=1 TO 3
20 N=N+1:PRINT N;I
30 GOSUB 10
```
OM ERROR after N=8

OSI NEWSLETERS

PEEK(65) P.O.Box 347 Owings Mills MD 21117	$12.00 for 12 issues
OSIO Newsletter David Morganstein 13329 Woodruff Pl. Germantown MD 20767	Membership $15.00/year Program exchange, discounts on OSI and other equipment.
The Aardvark Journal 1690 Bolton Walled Lake MI 48088	$9.00 for 6 issues
O.S.I. Users Independent Newsletter Charles Curley 6061 Lime Ave. #2 Long Beach CA 90805	$10.00 for 6 issues
OSI's Small Systems Journal Defunct. Complete set from PEEK(65) for $10.00. Contained many programs that (modified) are still useful.	Later, OSI published a section by that name in kilobaud MICROCOMPUTING. Presently is a section in MICRO.

6502 PUBLICATIONS

MICRO 34 Chelmsford Street Chelmsford, MA 01820	$2.00/issue "Best of MICRO" also available
COMPUTE! Circulation Dept. P.O.Box 5406 Greensboro, NC 27403	$2.00/issue, $16.00/year

```
1 REM   *** RANDOM NUMBER GENERATOR ***
2 REM
100 REM *** DRIVER ***
101 REM
105 GOSUB 9850:REM INITIALIZE
110 FOR I=1 TO 100
120 GOSUB 9800:REM USE
130 PRINT R7:REM R7 IS THE RANDOM NUMBER
140 NEXT
150 REM THE PERIOD OF THIS GENERATOR IS ABOUT 14000
999 STOP
9800 REM   *** RANDOM NUMBER GENERATOR ***
9801 REM
9810 F7=F7*15-233*INT(F7*15/233)
9815 G7=G7*15-251*INT(G7*15/251)
9820 R7=(F7*251+G7)/(233*251):RETURN
9850 REM ENTER HERE TO INITIALIZE
9855 F7=113:G7=71:RETURN
```

OSI SOFTWARE HOUSES

Aardvark Technical Services 1690 Bolton Walled Lake, MI 48808	Games, utilities, data sheets, firmware, hardware mods.
Aurora Software Associates P.O. Box 99553 Cleveland OH 44199	Games, utilities, business
Progressive Computing 3336 Avondale Court Windsor, Ont. CANADA N9E 1X6 or 3281 Countryside Circle Pontiac TWP, MI 48057	Games, utilities, data sheets, firmware, hardware mods.
Mittendorf Engineering 905 Villa Nueva Dr. Litchfield Park, AZ 85340	Software, data sheets, hardware
DBIS One Mayfair Road Eastchester NY 10707	Business
DBMS, Inc. P.O. Box 347 Owings Mills MD 21117	Manuals, business software
Bill's Micro Services 210 S. Kenilworth Oak Park IL 60302	OSI 1P programs
Software Federation Inc. 44 University Drive Arlington Heights IL 60004	Business
Orion Software Associates 147 Main Street Ossining NY 10562	Games
Prism Software Box 928 College Park MD 20740	Disk copy utility
Retelle 2005 Whittaker Rd. Ypsilanti MI 48197	Games
Mile High Software Co. 318 Linden Ave. Boulder CO 80302	Games
Earthship Box 489 Sussex NJ	Games

BAP$ Software 6221 Richmond Ave., Suite 220 Houston TX 77057	Personal financial
Perceptions Unlimited Box 3-186 ECB Anchorage AK 99501	Games
Dwo Quong Fok Lok Sow	Word processor
and	
Structured Program Designers 371 Broome St. NYC NY 10013	Business
Digital Technology, Inc. P.O.Box 178590 San Diego CA 92117	Business
The 6502 Program Exchange 2920 W. Moana Reno NV 89509	General 6502 programs, can deliver in KC tape format
Technical Products Co. P.O.Box 12983 Gainsville FL 32604	Disk FORTH, etc.
Systek, Inc. P.O. Drawer JJ Miss. State, MS 39762	Engineering programs
Honders Inc. 57 North Street Middletown NY 10940	Business
Aristo/Polks 314 5th Ave. NYC NY 10001	Games
Software Consultants 7053 Rode Trail Memphis Tenn. 38134	OS-65D V3.2 Manual
D $ N Micro Products, Inc. 3932 Oakhurst Dr. Fort Wayne IN 46815	OSI compatible hardware, boards

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	0D 96 0D 0D	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

See also Jim Butterfields list in COMPUTE., issue 2, January/February 1980, page 41.

BASIC MEMORY ROM

Thanks are due to many people who wrote me with entries, and especially to Bruce Hoyt and to Jim Butterfield. See also Jim's article in compute II., issue 2, June/July 1980 and the article in PEEK(65), Vol. 1, No. 12, December 1980.

A000 - A037	Initial Work Jump Table
A038 - A065	routine entry addresses
A084 - A186	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
A24E	message output
A274	warm start
A295	tokenize and store in BASIC
A2A2	delete a line from program
A32E	rebuild chaining of BASIC lines
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
A47A	CLEAR
A477	initialize
A491	clear stack, reset addresses
A4A7	initialize program scan pointer to beginning of program.
A4B5	LIST
A556	FOR
A5FF	execution routine
A61A	RESTORE
A629	CNTL/C
A638	STOP
A63A	END
A661	CONT
A67B	NULL
A691	RUN

A69C	GOSUB
A6B9	GOTO
A6E6	RETURN
A70C	DATA
A71A	scan for next BASIC statement
A71F	scan for next BASIC line
A73C	IF
A74F	REM
A75F	ON
A77F	decimal to binary, put answer in $11,12
A7B9	LET
A829	PRINT
A866	end of input line routine, puts out CR and LF & nulls
A8C3	string output routine, address in A,Y (lo, hi) end string with a null
A8E0	output single character
A8E5	output routine, calls $FFEE
A904	handle bad input data
A923	INPUT
A946	prompt and receive input
A94F	READ
AA1C	Message table "EXTRA IGNORED, REDO FROM START"
AA40	NEXT
AA9B	check data, print "TYPE MISMATCH"
AAC1	expression handler
ABAC	non-numeric expressions
ABD8	NOT
ABF5	check for "("
AC00	check for ")"
AC03	check for ","
AC0C	print "SN"
AC66	OR
AC69	AND
AC96	comparison
AD01	DIM
AD0B	search for variable location in memory
AD81	is character alphabetic?
AD8B	create new variables
ADE6	array pointer subroutine

ADF7	evaluate integer expression
AE05	= command
AE17	create new arrays
AF7C	compute array subscript size
AFAD	FRE
AFC1	fixed to floating
AFCE	POS
AFD4	check if "ILLEGAL DIRECT"
AFDE	DEF
B00B	check FNx syntex
B021	evaluate FNx
B08C	STR$
B0AE	scan and set up string
B115	build string vector
B147	garbage collector
B1D4	find string for collection
B218	collect string
B24D	string concatenation
B28A	put string in memory
B2B3	discard unwanted string
B2EB	clear descriptor stack
B2FC	CHR$
B310	LEFT$
B33C	RIGHT$
B347	MID$
B36F	pull string function paramerers from stack
B38C	LEN
B392	go from string mode to numerical mode
B39B	ASC
B3AB	input byte parameter
B3BD	VAL
B3FC	get 2 parameters for POKE and WAIT
B408	floating number in accumulator converted to fixed and put in $11,12
B41E	PEEK
B429	POKE
B432	WAIT
B44E	add 0.5 to acc. #1
B455	- command

B46C	+ command
B537	complement acc. #1
B564	print "OV"
B569	multiply a byte
B59C	function constant table
B5BD	LOG
B5FE	* command
B622	multiply a bit
B64D	load acc. #2 from memory
B673	test and adjust acc. #1 and #2
B690	over and underflow
B69E	multiply by 10
B6B5	10 in floating point binary
B6C2	divide by
B6CD	divide into, /
B74B	unpack memory into acc. #1
B76B	store acc. #1 in memory
B79B	acc. #2 to #1
B7F8	compare acc. #1 to memory
B7AB	transfer acc. #1 yo #2
B7BA	round off acc. #1
B7CA	sign of acc. #1
B7F5	ABS
B831	floating to fixed
B862	INT
B887	string to floating point
B912	get next ASCII digit
B947	table of constants to build string of a number
B953	output line number
B95E	hex in A,X converted to deximal and printed
B962	output decimal value of number (binary) in $AC,AF
B96E	build ASCII number in $100-10C from number in $AC-AF
BA96	table of constants for numeric conversions
BAAC	SQR
BAB6	^raise to a power
BAEF	negation
BAFA	table of constants for string evaluations
BB1B	EXP

BB6E	series evaluation
BBB8	table of constants for RND
BBC0	RND
BBFC	COS
BC03	SIN
BC4C	TAN
BC78	table of constants for trig. functions
BC99	ATN
BCEE	get character routine, moved to $BC
BD11	cold start
BE39	cold start messages
BF2D	output character to TV screen, do scroll, etc.

This list may contain some errors, or at least some omissions. The listed addresses are (sometimes approximately) where the code for that function begins. In many cases it is not the entry point. Often the code is not in the form of a complete subroutine, rather it is entered and left by jumps and branches, and thus cannot be used as a self standing unit outside of BASIC. This list of addresses should be very helpful if you wish to play around in the innards of BASIC, but you will also need a disassembly of the machine language code in the region of interest, and lots of patience.

```
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      $65
 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
```

ERROR CODES

CODE		MEANING
BS	B	Bad Subscript: Array index out of DIM range.
CN	C	CoNtinue error: Incorrect CONTinue from a BREAK.
DD	D	Double Dimension: Array DIMensioned twice, or DIM after using the array set the DIM to 10 by default.
FC	F	Function Call error: Either a BASIC function such as SIN, or an internal function such as AND, has been given an inappropriate variable.
ID	I	Illegal Direct: INPUT or DEF FN commands cannot be used in the immediate (direct) mode.
LS	L	Long String: String longer than 255 characters.
NF	N	NEXT without FOR.
OD	O	Out of Data: Have done a READ past the end of the last DATA statement.
OM	O	Out of Memory: Either the program and variable table used up memory, or the stack has overflowed from GOSUB's etc. This error may occur on the first command after a warm start. Just repeat the command.
OV	O	Overflow: Floating point number too large.
OS	O	Out of String memory.
RG	R	RETURN without GOSUB.
SN	S	SyNtax error: Incorrect spelling of commands, etc. (Have you a command hidden in a variable name, such as "TO" in "PAGETOP"?)
ST	S	String Temporaries: String expression too complex.
TM	T	Type Mismatch: String variable where a numerical variable was expected, etc.
UF	U	Undefined Function.
US	U	Undefined Statement: GOTO or GOSUB to a non-existent line.
/Ø	/	Division by zero.