RCA 1802 BASIC level 3 ver. 1.1
User Manual

Last revised: 9 Feb 2022. Comments? Corrections? Questions? Contact
Lee Hart <leeahart@earthlink.net> or Chuck Yakym <The--Eagle@sbcglobal.net>

BASIC3 was written for the RCA 1802 by Ron Cenker in 1981. It was RCA's most powerful BASIC interpreter for the COSMAC 1802 family of microcomputers. It is a fast, full-featured floating-point BASIC, much like the Microsoft BASICs of the time. It was documented in RCA's “BASIC3 High-Level Language Interpreter User Manual”, MPM-841A. There were versions to run from RAM or ROM; for disk-based or cassette-based storage systems; versions with the complete interpreter, or that simply ran a compiled BASIC program from ROM for stand-alone applications. It was originally intended for RCA's 1802 products; their various development systems, the VIP, and RCA's line of Microboards.

Ron graciously provided the source code, so it could be modified for other 1802 computers. These include the Netronics Elf-II, Quest Super Elf, COMX, Spare Time Gizmos Elf2k, and now the 1802 Membership Card <http://www.sunrise-ev.com/1802.htm>. Thanks to the combined efforts of Lee Hart, Herb Johnson, Ed Keefe, and Chuck Yakym, we are proud to bring BASIC3 back to life on modern 1802 hardware! Please let us know how you like it, or if you find any bugs or have ideas for improvements.

REQUIREMENTS

The BASIC3 interpreter needs 12k of memory, plus room for the I/O drivers. It also needs at least 1k of RAM, but will use all it finds up to 32k. We have two versions (with, and without monitor). Each version is available with different starting addresses, and normal or inverted Q outputs. These should be sufficient to run on an 1802 Membership Card, or just about any other Elf computer.

MCBASIC3 version (without monitor): This comes in a 16k EPROM (27128 or 27C128), which contains the BASIC3 interpreter and “bit-banger” serial I/O routines using the 1802 Q output and EF3 input. Address the EPROM at 0h, and RAM to start at 8000h. BASIC3 starts immediately upon power-up or reset. Just connect the power and serial terminal, and switch to RUN.

MCSMP20 version (with monitor): This version comes in a 32k EPROM (27256 or 27C256). It contains Chuck Yakym's Serial Monitor Program version 2.0, Ron Cenker's BASIC3, and the same “bit-banger” serial I/O routines using the 1802's Q and EF3 pins. Versions are available to address the EPROM at 0h (with RAM at 8000h), or at 8000h (with RAM at 0h). If the EPROM is at 8000h, you must manually enter an LBR 8000 instruction (C0 80 00) with your front panel to tell the 1802 to jump to the Monitor at 8000h. Enter this 3-byte program, switch to CLEAR, then switch to RUN.

SERIAL FORMAT

Connect your 1802 to a serial terminal (or PC running a terminal program). Chuck likes Realterm, but it also works with Hyperterm, Teraterm, Minicom, Procomm (and probably others). Set the serial format as follows:

9600 baud for 1802 at 1.79 MHz; or try 300-4800 baud for other clock 1802 speeds
1 start bit
8 data bits
no parity bit
1 stop bit
full duplex
10 msec “pacing” delay per character (pacing may not work with USB-serial adapters)
1 second “pacing” delay per line (or up to 6 sec/line to load long ASCII programs)

Press the <enter> or <return> key. The serial I/O driver in ROM uses this to learn your terminal's baud rate. At this point, you should be connected! Any key you type will get displayed on the screen, and the 1802's front panel LEDs will show the ASCII code of that character. For example, the <enter> key displays 0Dh.

Note: All BASIC keywords must be UPPER CASE! Push your CAPS LOCK key to make typing easier. Mistakes made while typing can be corrected with the <backspace> key.

The boxes show what you will see on your computer screen. **BOLD TEXT LIKE THIS** is what the computer types. `Text like this` is what you type. End each line that you type with the <enter> or <return> key.

The ORG 8000h ROM starts in Chuck's monitor. Its sign-on message is

```
Membership Card's Serial Monitor Program Ver. 2.0
Enter "H" for Help.
>N
```

Press “H” for a menu of its functions. To cut to the chase and run BASIC3, press “N”.

The ORG 0h ROM immediately goes into BASIC3. You will see the message

```
WELCOME TO THE 1802 BASIC3 V1.1
(C)1981 RCA
C/W?
C

READY
:
```

Type “C” for a Cold start; this clears memory, so use it the first time you start BASIC. Or, after you've run BASIC once, you can also type “W” for a Warm start; this leaves whatever program is already in memory alone.

BASIC is now up and running! It will display READY and a colon (:), to show it is ready for your commands.

```
READY
:
```

If you've used other BASICs before, this one will be very familiar. If a line does not begin with a number, BASIC tries to execute it immediately as a command. If it can't, an error message will be displayed.

If a line begins with a number, it is treated as a line number in a BASIC program, and is inserted in numerical order into the program. If a line with that number already exists, the new line replaces the old one. A number typed with nothing after it deletes that line from the program.

Numbers are floating-point unless defined as integers with a DEFIN command. You can type extra spaces between words for readability. You can put multiple statements on the same line by separating them with colon (:).

COMMAND AND FUNCTION SUMMARY

Commands:	BYE	DISINT	FORMAT	RENUMBER	
	CLD	EDIT	LIST	RUN	
	CLS	ENINT	NEW	TRACE	
Definitions:	DEFINT	DEG	FIXED	RAD	
	DEFUS	DIM	LET	REM	
Controls:	END	GOSUB	NEXT		
	EXIT	GOTO	RETURN		
	FOR... TO... NEXT	IF... THEN	WAIT		
Data:	DATA	READ	RESTORE		
I/O:	DMAPT	INPUT	POKE	STQ	
	EF	OUT	PRINT	TIN	
	INP	PEEK	QST	TOUT	
File:	* CLOSE	* DOUT	PLOAD	* RFLN	
	* DIN	* DSAVE	PSAVE	* WFLN	
	* DLOAD	(* = not implemented, as there is no form of mass storage)			
Machine Language:	CALL	USR			
Functions: Math:	ABS	EXP	MEM	RND	
	ATN	FNUM	MOD	SGN	
	COS	INT	QST	SIN	
	EOD	INUM	PI	SQR	
	EOP	LOG			
String:	ASC	FVAL	MID$		
	CHR$	LEN	TAB		

FUNCTIONS

Here is a more detailed list of the functions available, with examples of what they do.

ABS

Format: var = ABS (expr)

This function returns the absolute value of the expression. For example

```
:PR ABS (-10*2)
20
```

ASC

Format: var = ASC (string expr)

Returns the decimal equivalent of the ASCII value of the first character in the referenced string. The value returns in floating point as a default. If preceded by any integer function, it will be converted to integer. For example

```
:A$ = "ARITHMETIC": B=ASC(A$): PR B
65
```

The first character of A$ is “A”, and the ASCII character “A” has a decimal value of 65. As another example

```
5 A$="TEST"
10 FOR A = 1 TO LEN(A$)
20 PR ASC (MID$(A$,A))
30 NEXT A
RUN
84                              (decimal value of “T”)
69                              (decimal value of “E”)
83                              (decimal value of “S”)
84                              (decimal value of “T”)
```

ATN

Format: var = ATN (expr)

ATN (ARCTAN) returns the angle (normally expressed in radians) whose tangent is equal to the value of the expression. This is a floating-point function. For example

```
:PR ATN (1)
.785399                         (since TAN (0.785399) = TAN PI/4 = TAN 45°)
```

BYE

Format: BYE

Exits BASIC3. In the ORG 8000h version, BYE exits to the Monitor. In the ORG 0 version, BYE just restarts BASIC3 since there is no monitor. You will be prompted to hit the <CR> key to continue.

CALL

Format: CALL (expr1)
CALL (expr1, expr2)
CALL (expr1, expr2, expr3)

This statement CALLs an 1802 machine language program or subroutine at the address specified by expr1. If expr2 is present, its value is passed to the machine language program in R8. If expr3 is present, its value is passed to the machine language program in RA. Write the machine language routine as follows:

1. R3 is the program counter upon entry into the subroutine.
2. Transfer back to BASIC with a SEP R5 D5h) instruction.
3. The machine language routine has free use of R8, RA, RC, RD, and RE. If any other registers are used, they should be saved first on the stack, and restored before returning to BASIC.
4. RCA's Standard Call and Return Technique (SCRT) is **not** used because it destroys the upper part of register F (RF.1).
5. Call and return routines are initialized by BASIC so no further initialization is required by the machine language program.
6. The stack (R2) is available for use so long as it returns as it was left.

Any of the expressions (expr1, expr2, expr3) may be expressed in either integer or floating point. BASIC will automatically convert them to integer. RD is initialized to timing constant for utility program.

CHR$

Format: string var = CHR$ (expr, expr.....)

This function returns a character corresponding to each bracketed expression. Thus, CHR$ is the opposite of the ASC function. CHR$ evaluates each expression and outputs a character having an ASCII decimal code equal to the value of the expression. This is a useful way to send control codes like TAB=9, LF=10, CR=13, and ESC=27. For example, the ASCII decimal code for A is 65. So

```
:PR CHR$ (65)
A
```

Multiple values can be provided, separated by commas. Values can be expressed as variables, or numbers in decimal or hexadecimal (preceded by #). For example

```
:A=65: PR CHR$(A, 9, 66, 9, #43)
A       B       C
```

CLD

Format: CLD

CLD (Clear Data) erases all strings and arrays. It does so by resetting all string and array pointers to their initial states.

CLS

Format: CLS

CLS (Clear Screen) clears the screen, and positions the cursor in the top left corner.

COS

Format: var = COS (expr)

COS returns the Cosine of the angle determined by the value of the expression (normally expressed in radians). This function is a floating-point function. For example

```
:PR COS (PI/3)
.500001
```

DATA

Format: DATA data list

The DATA statement contains data to be used by a READ statement. Each element in the DATA statement must be separated by a comma. Any string of characters must be enclosed in quotes. The DATA statement may appear anywhere in a BASIC program. The following are acceptable DATA statements.

```
1 DATA 2*A, A$(1), "HELLO", B
1000 DATA 1,2,3,4
5000 DATA SIN (45), SIN (60), SIN (90)
```

DEFINT

Format: DEFINT
DEFINT var

This statement (DEFfine INTeger variables) without any variable name sets all variables (A-Z) and all arrays (A (expr) - Z (expr)) as floating point. If a variable name is included in the DEFINT statement, all variables

and arrays from A up to and including the variable name will be set up as integers. For example,

```
DEFINT D
```

defines variables and array names A, B, C and D as integer. A NEW statement resets all variables to floating point. For this reason, if integer variables are used in a program, a DEFINT statement should appear early in the program.

Note: if a DEFINT is used with no variable name, it must end with an <enter> or <return> key. A colon (:) may not be used to put further commands on the same line.

DEFUS

Format: DEFUS expr

This statement (DEFine start of User Space) moves the start of BASIC's program space further up in memory, as shown in figure 1. It allows the programmer to create a "hole" in memory to store machine language routines or data. Expr defines where the program space will begin. In the ORG 0 ROM, the user's BASIC program begins at @8300 (hexadecimal). The expression must evaluate to a number greater than @8300. BASIC will round down the expression to the beginning of the page. If an attempt is made to define the user space at an address lower than @8300, BASIC will "self-destruct", since the program would overwrite the system area (see figure 1) and the BASIC interpreter will no longer work properly. Once a DEFUS statement is executed, the only way to get the program space back to @8300 is by another DEFUS to @8300. The statement destroys the user program currently in memory.

An interesting feature of moving the start of user program space lies in the PSAVE statement. If a program has been generated at a moved location and it has associated with it some machine language routines, a PSAVE statement will save everything starting at @8300 to the end of program space. Thus it will save the machine language routines as well as the associated BASIC program. A subsequent PLOAD will load in all of the above, including the machine language routines, and will also redefine the start of user space to where it was when the file was created. No book-keeping is necessary.

```
:DEFUS @8700
```

Or

```
:DEFUS @87CC
```

moves the start of user space to @8700 to create a 1k byte hole (@8300-@86FF). To move it back to its original location, use

```
:DEFUS @8300
```

Figure 1: Action by DEFUS to move the beginning of program space up.

@FFFF	Top of Memory	
@8700		New start of BASIC program
@8300-@86FF		Hole created by DEFUS @8700
@8200-@82FF	BASIC Variables	
@8000-@81FF	System Variables	
@0000-@3FFF	BASIC (in ROM)	

DEG

Format: DEG

This statement sets the unit of measure for angular functions to degrees. The default state is radian measure. The complement of this statement is RAD. (Note: Quest Super BASIC does not support the DEG and/or RAD command and only uses unit of measure in degrees.)

DIM

Format: DIM var list

This statement (DIMension arrays) reservs memory for arrays of numbers. These arrays may be one dimensional or two dimensional. The expression defines how large the array is in one or both dimensions. The mode of the array (floating point or integer) depends on whether its associated variable name has been defined as either integer or floating point. For instance, if all variables A-Z are floating point, then all arrays will be floating point (independent of the mode of the expression defining its limits). If variables A, B, and C are defined as integer, then arrays named A, B and C (if used) will be defined as integer. Multiple arrays may be dimensioned in the same DIM statement so long as each array is separated by a comma.

If memory is exceeded when an array is being dimensioned, an error message will result. Trying to use any array element that has not been previously dimensioned will also result in an error message. The programmer can use a CLD statement to clear all unwanted data space in order to reclaim space.

Here is an example to dimension space for 100 numbers (a 2-dimensional 10 x 10 array) with names ranging from A(1, 1) to A(10, 10); and a 1-dimension array of 20 numbers ranging from B(1) to B(20).

```
:DIM A(10, 10), B(20)
```

The dimension statement can be used in the direct mode of execution (as shown), or in a program. Any arrays generated by a program remain intact after program execution is completed. One can interrogate the contents of any array in the direct mode by means of the PRINT statement. The array remains intact until a CLD is executed or any editing to the program is done. Note that if "RUN" is executed, the data space is not cleared and the array has not been deleted. In this case, re-dimensioning will result in an error message. For that reason, the programmer should begin execution at a line-number after the DIM statement, if he wants to use previously generated data. The array data as well as the string data, is stored **after** the user space. Any editing of the original program will alter the user space and thus destroy the data. It should be noted that an array can be re-dimensioned without destroying other arrays or strings.

DISINT

Format: DISINT

DISable INTerrupt command, to disable the 1802 interrupt flag. See ENINT command to turn interrups on.

DMAPT

Format: var = DMAPT (expr)

Load the 1802's DMA Pointer register R0 with the value specified by expr. Example

```
10 DMAPT (@D445)
```

Will result in R0 being loaded with the hexadecimal value of @D445.

Note that this command does not work in the EMMA 1802 emulator. This is caused by some code introduced to handle output to a Pixie (CDP1861) screen. This code will always reset R0 with @2000, which is used as the Pixie display buffer. If no Pixie output is used this feature can be turned off by changing

address @21E7 and @21EA to #C4, from BASIC via:

```
POKE (@21E7,#C4)
POKE (@21EA,#C4)
```

After these commands the DMAPT statement will work as intended.

EDIT

Format: EDIT X

This statement allows the user to modify line number (X) on a character-by-character basis. To correct a short line, it's easier to simply retype the line. For a long line, it's possible to add new errors while attempting to correct the old ones. In this case, the use of EDIT is preferred.

In response to EDIT X, line X is displayed. BASIC3 is now in the "edit mode" (i.e. under the control of the EDITOR). The user can now move the cursor (by pressing the <space> key) to a position one character before (i.e. to the left of) the character to be modified. At this point, three EDIT options are available: I (for Insert), D (for Delete) or C (for Change). After the option is selected; the option character (I, D or C) is displayed and the cursor is now directly under the character to be modified.

- Insert allows characters to be inserted immediately after the I.
- Delete removes a character after the D for each press of the <space> key.
- Change sequentially replaces each character with the new one that is typed after C.

After making a change, type <ctrl-S> (control-S, i.e. hold the CTRL key down and then press S) to SHOW the modified line. Note that in one pass, only one option (I, D or C) can be used. But you can use another option (I, D, or C) after the <ctrl-S> to make further changes as many times as is needed until the retyped line is correct. When finished, press <ctrl-S> again to "exit" from the EDIT mode and put the edited line in the program. This process is illustrated as follows, where "." denotes the <space> key. Type:

```
:10 PRINT "HELLO"          (type this line to enter it in your program)
:EDIT 10                   (now type this to EDIT line 10)

 10 PRINT "HELLO"          (BASIC prints the line as it currently is)
```

To insert SAY before HELLO in line 10, first move the cursor under the first " by typing spaces. Then type the letter ”I” to INSERT. Now type the text to insert (SAY), and then a space.

```
 10 PRINT "HELLO"
..........ISAY.            (space under the “, type I to Insert, then SAY, and another space)
 10 PRINT “SAY HELLO”      (type ctrl-S to show the new line as edited)
 READY                     (it looks right; so type another ctrl-S to finish and exit EDIT mode)
:
```

Now let's delete SAY, and change HELLO to JERRY.

```
:EDIT 10                   (EDIT line 10)

 10 PRINT "SAY HELLO"      (BASIC prints the line as it currently is)
..........D....            (space under the “, type D to Delete, then four spaces)
 10 PRINT “HELLO”          (type ctrl-S to show the new line as edited)
..........CJERRY           (now space under the “, type C to Change, then JERRY)
 10 PRINT “JERRY”          (type ctrl-S to show the edited line)
 READY                     (it looks right; so type another ctrl-S to finish and exit EDIT mode)
```

```
:
```

This time, we used the D (for DELETE) option. Each space you type after D will Delete one character; so we typed 4 <spaces>. When done, press <ctrl-S> to SHOW the corrected line.

Then we used the C (for CHANGE) to replace HELLO with JERRY. Another ctrl-S shows the result. It looked OK, so we typed a final ctrl-S to exit the editor. BASIC responded with READY to show that control has returned to the BASIC interpreter.

If you make a mistake while editing, press <ctrl-A> while in the EDIT mode to ABORT editing, and return to BASIC with no changes in the line.

EF

Format: var = EF1
var = EF2
var = EF3
var = EF4

This function returns the 1 bit value (i.e. 0 or 1) of EF flag 1 to 4.

END

Format: END

This statement ENDs program. It terminates program execution and returns BASIC to the direct execution mode. An END statement may be placed anywhere and any number of times in a BASIC program. It may also be left out, if it would have been the last statement in a program.

ENINT

Format: ENINT line number

ENable INTerrupts. After this command, a BASIC program can be interrupted by a low on the 1802's /INT pin. When this happens the program will jump to the specified line number. Example:

```
10 ENINT 1000
20 LET A = 0
30 PRINT "HELLO"
40 LET A = A + 1: PR A
50 GOTO 30
100 END

1000 PRINT "INTERRUPT"
```

When an interrupt occurs, the BASIC program will be stopped by a jump to line number 1000. See also the DISINT, the DISable INTerrupt command.

Note that the interrupt here is the physical CDP1802 Interrupt pin, which in the default Membership Card hardware is not connected to anything. External hardware could however make use of this, allowing a BASIC program to execute a specific routine when triggered by the hardware.

EOD

Format: EOD

EOD (End Of Data) prints the hexadecimal address of the end of the data space (arrays and strings).
For example

```
:EOD
@441A
READY
:
```

which says the end of the user program and data space is currently at 441A (hex). EOD is valid only after data has been generated by running the program (and after DIM has been executed).

EOP

Format: EOP

EOP (end of Program) is similar to EOD; it prints the hexadecimal address of the end of the current user program space. The user can type it at any time during program development to see where the end of program is located in memory. For example

```
:EOP
@6411
READY
:
```

EXIT

Format: EXIT expr

This statement is an unconditional branch to a line number defined by expr. It is intended for a premature exit from a FOR-NEXT loop or a subroutine. Note: If subroutines or FOR-NEXT loops are nested, the EXIT statement transfers control to a line number within the next level down of nesting. For example, if your FOR/NEXT loops are nested four deep and you need to jump out of the fourth FOR/NEXT loop, the EXIT must branch somewhere in the third FOR/NEXT loop. A second EXIT may then be taken to some line within the second FOR/NEXT loop, and so on.

Use EXIT instead of GOTO to exit FOR-NEXT loops so the stack pointers will be corrected. For example

```
10 FOR I=1 TO 10
20 FOR J=1 TO 5
30 A=B+C
40 IF A=15 THEN EXIT 60
50 NEXT J
60 PR A
70 NEXT I
```

EXP

Format: var = EXP (expr)

EXP returns a floating-point number of 2.71828 (e) raised to the power of the value of the expression.

```
:PRINT EXP (6/3)
7.38906                    (since e^6/3 = e^2 = 2.718282^2 = 7.38906)
```

FIXED

Format: FIXED expr

FIXED formats the printing of all numbers, floating point and integer. The value of the expression defines how many digits to the right of the decimal point will be printed. Trailing zeroes will be filled in to complete the number. If necessary, the number will be rounded. The expression must evaluate to a number between

0 and 6. If a number greater than 6 is entered, BASIC reverts to its normal mode of outputting numbers. Here are some examples

```
:FIXED 2: PRINT 123
123.00

:FIXED 2: PRINT 123.567
123.57

:FIXED 2: PRINT 6E7
.60E+08

:FIXED 7: PRINT 123.50
123.5                          (7 is too large, so BASIC reverted to normal number printing)
```

FNUM

Format: var = FNUM (expr)

This function rounds the expression to the nearest whole number and converts it to the floating-point mode. It is an exact counterpart to INUM. For example

```
:PR PI, FNUM (PI)
3.14159 3
```

FOR ... TO ... STEP

Format: FOR var = expr1 TO expr2 STEP expr3

This statement assigns the value of expr1 to a variable, then continues execution. When a NEXT statement is found, the value of the variable is incremented by the amount defined by expr3 (which may be either positive or negative). (Expr2 – var) is then evaluated, and compared to the sign of the step. It is assumed that zero has a positive sign. If a match in sign does not occur, execution goes back to the first statement after the last FOR statement, and repeats the loop again. At any time during the loop, the user can modify the value of the variable name (such as LET...). The mode of the variable name sets the mode in which expr1, expr2, and expr3 are evaluated. If the STEP expr3 is deleted, a step size of +1 is used.

FORMAT

Format: FORMAT N

FORMAT specifies the field size (i.e. the number of places of printed numeric data). Any PRINT statement coming after the FORMAT statement, whether printing the value of a variable, or printing a number, will use the field size specified.

If the field size of a positive number is greater than N, N asterisks will be printed. If the number is negative, a negative sign followed by (N-1) asterisks will be printed. For example

```
:10 A = 123456
:20 FORMAT 6
:30 PR A
:RUN
123456                         (the number is printed, because it fits in a 6-character field)

:20 FORMAT 5
:RUN
```

```
*****                          (now the number is too big to fit, so asterisks are printed instead)

:10 A = -12345
:RUN
-****                          (the number fits, but is negative which needs one more space)
```

The range of n is from 1 to 15. FORMAT 0 turns formatting off. For example

```
:10 FORMAT 5
:20 PR 123456
:30 FORMAT 0
:40 PR 123456
:RUN
*****
123456
```

FVAL

Format: var = FVAL (string expr)

This function evaluates the string expression as an arithmetic expression and returns its value. In this manner the user can create mathematical functions. Here are three examples

```
:10 A$="8+4"
:20 PR FVAL (A$)
:30 PR FVAL (A$+"/2")
:40 PR FVAL ("SQR(3)")
:RUN
12
10
1.73205
```

Note: The string length is limited to about 48 characters. Beyond that, valuable data area may be clobbered!

GOSUB

Format: GOSUB expr

GOSUB calls a subroutine. It is the same as GOTO, except that the program remembers where the GOSUB occurred. When BASIC encounters a RETURN statement, it returns to the statement following the last GOSUB statement executed. In this way, subroutines may be nested as deep as memory will allow (the stack grows as the number of nested GOSUB's grows). For example

```
10 A=2000
20 GOSUB 1000: GOSUB 1000
30 GOSUB A
40 END

1000 PR "WORKING... ";
1010 RETURN

2000 PR "FINISHED" : GOSUB 3000: GOTO 1010

3000 PR "COMPLETE!": RETURN
```

```
RUN
WORKING... WORKING... FINISHED
COMPLETE!
```

GOTO

Format: GOTO expr

GOTO is an unconditional branch. It transfers execution immediately to the start of the line number specified by the expr. If the line number does not exist, an error message is generated. Examples

```
10 GOTO 50
10 GOTO A+B
10 GOTO 100*(A-B)
```

IF ... THEN

Format: IF stringref <> string expr THEN statement
IF stringref = string expr THEN statement
IF expr (reloperator) expr THEN statement

IF is a conditional branch. It tests for a condition, and if the condition is met, a statement or group of statements (separated by colons) is executed. If the condition is not met, then execution continues at the next line number. When strings are compared, <> (not equal to) or = (equal to) are the only acceptable relations. When arithmetic expressions are compared, the acceptable relational operators are:

```
 =  equal to
<>  not equal to
 >  greater than
 <  less than
>=  greater than or equal to
<=  less than or equal to
```

Both expressions being compared must be the same type (i.e. both strings, or both numeric). For example

```
10 IF A=B THEN PR A: WAIT (200): GOTO 200
20 PR B
```

If the value of A is equal to the value of B, then print A, wait a second, and then go to line 200.
If A is not equal to B, then continue on line 20 (print B, etc.)

The keyword THEN is optional and need not be used. If THEN is used, anything to the right of THEN must be an executable statement. A common mistake is

```
IF A>N THEN 200              (won't work because “200” is not an executable statement)
IF A>B THEN GOTO 200         (this works)
IF A>B GOTO 200              (so does this)
```

More examples of IF statements are given below. Note the multiple conditions in the last example. If any of the conditions are not met, execution continues at the next line number. If **all** of the conditions are met, execution will finally get transferred to line 500 by the GOTO statement. It follows that the GOTO statement could have been any executable statement.

```
10 IF A(2)*B(1) >= C*SIN (A) PR A: GOTO 50
10 IF A$(2) = "LIST" LET B=3: GOTO 100
```

```
10 IF B$(A+B) = MID$ (A$, 2, 4) PR "OK"
10 IF MID$(A$(5), 2, 4) = "EBCD" PR "MATCH": GOTO 500
10 IF A$=B$+C$ THEN GOSUB 200: CLS: PR "DONE"
10 IF A=INT(S/5) IF B>10 IF C<5 GOTO 500
```

INP

Format: var = INP (Z, X)

Read Input port. If Z = 0, the value of Input port X is returned. If Z is not equal to 0, first Output the 8-bit value of Z to Output port 1. After this, the value of input port X is returned.

Table 1: INP instruction selected by X value

X	1802 assembler mnemonic	1802 opcode
1	INP1	69
2	INP2	6A
3	INP3	6B
4	INP4 (Membership Card input port)	6C
5	INP5	6D
6	INP6	6E
7	INP7	6F

Examples

```
PR INP (0, 4)          (prints the value of input port 4, i.e the Front Panel switches)
PR INP (10, 3)         (output 10 to port 1, then print the value of input port 3)
```

INPUT

Format: INPUT var list
INPUT "string" var list

When BASIC encounters an input statement, it prints a question mark "?" as a prompt. It then waits for a user response. A list of variable names may appear after the INPUT statement. Each of the variable names must be separated by a comma. In response to the prompt "?", the user may answer with any expression or group of expressions separated by commas.

If the INPUT statement has three variable names after it and the user responds with only two expressions, BASIC will issue another prompt (?) and wait for the third expression. Conversely, if the INPUT statement has only one variable name and the user responds with two expressions, BASIC will use the first and continue execution. At the next INPUT statement, it will use the previous unused expression without issuing another prompt. When INPUT statements have used up the unused expressions, another prompt "?" will be issued.

The data that is input is assigned to the associated variable name as it is inputted. The mode of the variable name defines the mode in which the associated input expression will be evaluated.

Note that an INPUT statement may not contain both string variable names and normal variable names at the same time. Furthermore, each INPUT statement containing a string variable name can contain one name. Because of this limit, in response to an INPUT A$ statement, the user can answer with a string of characters with no quotation marks. The carriage return marks the end of the string. The following are acceptable INPUT statements.

```
10 INPUT A, B, C (1)
10 INPUT A (1), A (B)
10 INPUT A$
```

```
10 INPUT B$ (B)
```

The following are incorrect INPUT statements.

```
10 INPUT AB                (no delimiter between variables)
10 INPUT A$, B$            (multiple string names forbidden)
```

When the optional 'string' parameter is used the string test is printed prior to the question mark prompt. No delimiter or separator is used before the first quote or after the last quote.

INT

Format: var = INT (expr)

This function returns the INTeger part of a floating-point expression without the fractional part. The result is still a floating-point number. This is different from INUM described next. Example

```
A=7.9: B=INT(A): PRINT A,B
7.9     7
```

INUM

Format: var = INUM (expr)

This function converts a floating-point expression to the integer mode, and rounds it to the nearest whole number. It provides a means for forcing a particular function to the integer mode. For example

```
PRINT SQR(62.41), INUM(SQR(62.41)
7.9       8
```

LEN

Format: var = LEN (string expr)

This function returns the LENgth (number of characters) in the specified string. The length is a floating point number. For example

```
:A$="ARITHMETIC"
:PR LEN(A$)
10
```

LET

Format: LET

The assignment symbol is the equal sign "**=**". When this symbol is used for the purpose of assignment, it takes on the meaning "takes the value of" rather than "equals". The keyword LET is optional. Examples:

```
:LET A=5                   (sets the variable A to 5)
:A=5                       (same thing; LET is optional)
:LET B=B+A*2               (sets the new value of B to the old value of B plus A times two)
```

LIST

Format: LIST
LIST expr
LIST expr, expr

This statement LISTs the program. With no expression, it lists the entire program in user space. If one

expression is given, it lists only that one line number. If two expressions are given and separated by a comma, the listing will start and end at the respective lines numbers. If an expression evaluates to a line number that does not exist, then the line with the next higher number will be used.

LIST can be used to save a BASIC program in ASCII format, so it can be viewed, saved, and edited. Type the LIST command with line number, but not the final <enter> key. Set your terminal program to "capture" or "receive" plain ASCII text. Then hit the <enter> key to start the transfer.

LOG

Format: var = LOG (expr)

This function returns the natural logarithm of the value of the expression. It is the opposite of EXP. Example:

```
:PR LOG (10)
2.30259
:PR LOG(EXP(2))
2
```

MEM

Format: PR MEM

This function returns the decimal number of bytes of memory left between the end of data (string and arrays) and the end of the normal stack. The actual number returned is reduced by 256 to allow for stack growth during program execution. To print the remaining amount of memory left, simply type

```
PR MEM
31702                    (this is with a 32k RAM and no program in memory)
```

MID$

Format: string var = MID$ (string expr, expr1)
string var = MID$ (string expr, expr1, expr2)

This function extracts a portion of the specified string. The first term, expr1, defines which character from the left is to start the substring. Expr2 defines the number of characters to be used after that point in the substring. If expr2 is not used, all of the remaining characters are taken. Example

```
:A$ = "KNOW"
:PR MID$ (A$, 2, 2)          (starting at the 2nd character, print 2 character in A$)
NO
:PR MID$ (A$, 2)             (starting at the 2nd character, print the rest of A$)
NOW
```

This program loops until the user inputs a word beginning with the letter "Y". When found, it prints the word.

```
10 INPUT A$(1)
20 IF MID$(A$(1), 1, 1)="Y" GOTO 100
30 GOTO 10
100 PR A$
```

Functions like LEFT$ and RIGHT$ in other BASICs can be implemented with MID$ like this

```
MID$ (A$, 1, N)                    (is the same as LEFT$(A$, N)  )
MID$(A$, (LEN(A$)-N+1), LEN(A$))   (is the same as RIGHT$(A$,N)  )
```

MOD

Format: var = MOD (expr1, expr2)

The MODulo function returns the remainder of a division. Each expression is converted to an integer, and the calculation expr1 – (expr1/expr2)*expr2 is performed. For example

```
:PR MOD (10,3)
1                                   (because 10 divided by 3 is 3 with a remainder of 1)
```

NEW

Format: NEW

This statement initializes BASIC, clears the user space, and all data space. That is, it erases the user-generated BASIC program and initializes all string and array pointers.

NEXT

Format: NEXT
NEXT simple var

This statement closes the FOR/NEXT loop (see the FOR statement). If the variable name is omitted, the NEXT statement returns to the last FOR statement and continues. If the variable name is included, BASIC will check to see if it matches the variable name used in the last FOR statement. If it does not match, an error message is issued.

OUT

Format: OUT (Z, X, Y)

If Z = 0, this function outputs the (8-bit) value of Y to port X. If Z is not equal to 0, this function outputs the (8-bit) value of Z to port 1, and then outputs the (8-bit) value of Y to port X. This supports RCA's 2-level I/O scheme for getting more than 7 I/O ports. For example

```
:OUT (0, 4, 255)          (outputs the value 255 (hex FF) to output port 4 (the Front
                           Panel LEDs)
:OUT (10, 4, 1)           (first outputs 10 to port 1; then 1 to port 4
```

Table 1: OUT instruction selected by X value

X	**1802 assembler mnemonic**	**1802 opcode**
1	OUT1	61
2	OUT2	62
3	OUT3	63
4	OUT4 (Membership Card output port)	64
5	OUT5	65
6	OUT6	66
7	OUT7	67

The default configuration displays the last serial character received on the Front Panel LEDs (OUT 4). To use the OUT4 port and LEDs for other purposes, use a POKE statement as follows:

ORG 0 ROM	**ORG 8000 ROM**	**Usage**
POKE (@8008, 0)	POKE (8, 0)	OUT4 displays last serial character received
POKE (@8008, 1)	POKE (8, 1)	OUT4 only displays what the OUT statement writes to it

PEEK

Format: var = PEEK (expr)

This function returns the decimal equivalent of the contents of memory at an address determined by the expression. The decimal result defaults to floating point.

PI

Format: var = PI

This function returns as its value 3.14159.

PLOAD

Format: PLOAD

Clears the program space (NEW), and then loads a BASIC program (previously saved with PSAVE). For example

```
:PLOAD

READY TO LOAD                          (BASIC is now waiting for the program file)
```

Now use your terminal program to send a BASIC program file (saved in PSAVE format) to your serial port. The Membership Card's Front Panel LEDs will flash as the data is received. After the entire file has been received, BASIC will respond with

```
PRESS ANY KEY TO CONTINUE              (loading is done; hit <enter> to return to the BASIC prompt)
:
```

You can Abort PLOAD (**after** the file transfer has started) by pressing the IN button on the Membership Card Front Panel (to pull the 1802 EF4 pin low). If you abort PLOAD, you must also stop the file transfer from your terminal (so it won't keep sending garbage). Then return to the terminal mode, and press the <ENTER> or <CR> key to continue.

BASIC will respond to the break with

```
ERR CODE  21
READY
:
```

NOTE: The PLOAD and PSAVE data format use only the ASCII characters @ABCDEFGHIJKLMNO which can be loaded and saved using simple ASCII text files. The lower 4 bits of each character pair are a binary representation of the BASIC program as it appears in memory. Character pacing may be needed for this command to work correctly (I'm successfully using 10 msec/character at 9600 baud with a 1.79 Mhz clock).

POKE

Format: POKE (expr1, expr2)

POKE writes to any memory location (expr1) with any (8-bit) data (expr2). For example, POKE (5000,255) will write the decimal number 255 (hexadecimal FF) into decimal memory address 5000 (hexadecimal 1F23). A safer and more readable format is POKE (@1F23, #FF). Use POKE with extreme caution, as it can write into memory locations that will crash BASIC!

PRINT or PR

Format: PRINT
PR
PRINT print list
PR print list

The PRINT statement outputs the values of a list of expressions or string functions to the serial I/O routine.

Multiple items can be printed by separating them with a comma (,) or semicolon (;). When a semicolon is used, no spaces are added, so the next item is printed directly after the previous one. When a comma is used, spaces are added to reach the next multiple of 8 columns, and then the next item is printed. A comma or semicolon at the end of a print list inhibits the carriage return at the end of the line. The next PRINT statement will therefore continue printing on the same line where the last PRINT statement ended. If a PRINT statement is used all by itself, then a carriage return/ line feed is outputted.

PR can be used as an abbreviation for PRINT. In a listing, PR's are replaced with PRINTs. There are two functions usable only in a PRINT statement: TAB (expr) and CHR$ (expr). When more than one arithmetic expression appears in a PRINT statement, each expression may be evaluated in a different mode.

Examples of valid PR statements are

```
:PRINT 5
:PRINT A
:PR (A+B)/C
:PR A, B, 1234, C(5)
:PR "THE VALUE OF A= "; A
:PR A$(1) + A$(2)
:PR A$(2), A, B;
:PRINT MID$ (A$, 1, 2)
:PRINT TAB(A+B); 10; TAB(30); E
:PR A
:PR
```

PSAVE

Format: PSAVE

Saves the current BASIC program, by sending it out the serial port. For example

```
:PSAVE

READY TO SAVE          (BASIC now pauses 15 seconds while you set your terminal program to receive)
```

You have approximately 15 seconds to configure your terminal program to receive (capture) data from the serial port to a file. BASIC will send the entire program in an encoded format using only the ASCII characters @ABCDEFGHIJKLMNO. The Membership Card's Front Panel LEDs will flicker as the data is sent. When they stop flickering, sending is complete. Tell your terminal program to **close** the capture file to save the data. Then return it to “terminal” mode, and press the <ENTER> or <CR> key to continue.

NOTE: You can Abort PSAVE (**before** the start of the file transfer) by pressing the IN button on the Membership Card’s Front Panel. If you abort PSAVE, BASIC will respond with

```
ERR CODE 0
READY
:
```

You then have to tell your terminal program to abort the transfer (close and discard the captured data or file received so far).

QST

Format: var = QST

This function returns the 1 bit value (i.e. 0 or 1) of the 1802's Q flag and pin.

RAD

Format: RAD

RAD sets the unit of measure for angular functions to RADIANS. This is the default state of BASIC. The complement of this statement is DEG.

READ

Format: READ var list

The READ statement reads data from DATA statements, and assigns it to the specified variables. Each time a READ statement requests data, an internal BASIC pointer moves to the next item in a DATA statement. When that DATA statement is out of data, BASIC searches ahead for the next DATA statement. If none is found, an error message is returned for lack of data. It is important to keep track of the variable or string names in the READ statement and the corresponding data in the DATA statement. They must match in form; strings go with strings, and so on. Here is an example of acceptable DATA/READ statement pairs.

```
10 DIM A(4)                    (define A as a 4-element array)
20 DATA 10, 20, 30, 40         (here is the data we want to put in it)
30 FOR X = 1 TO 4              (For elements 1 to 4...)
40 READ A(X)                   (...read each DATA item in line 20, and store it in array A)
50 NEXT                        (...so now A(1)=10, A(2)=20, A(3)=30, and A(4)=40)
60 READ A$(1)                  (another READ; line 20 DATA is used up, so A$=line 80 DATA)
70 PR A$(1); A(2)*A(3)         (print A$, then multiply A(2) x A(3) = 600)
80 DATA "20*30 IS EQUAL TO "
RUN
20*30 IS EQUAL TO 600          (and here is the result)
```

REM

Format: REM can be followed by anything

REMark is used for comments. Any text after the REM is listed in the program, but ignored during program execution. It allows the programmer to include comments in the listing.

RENUMBER

Format: RENUMBER
RENUMBER N

This command will RENUMBER the lines in a program with a given increment N. If no N is given, the increment defaults to 10, so the line numbers will be 10, 20, 30... When N is given, it is the first line number as well as the increment. N numbers above 256 will be modulo 256 (i.e. only the low 8 bits of N are used).

Use RENUMBER with care! Save a copy of your program before renumbering it. If you have computed branches, BASIC cannot correct them automatically. A **computed branch** is a statement (like GOTO A or GOSUB 10*B) that requires the computer to branch or jump to another statement whose line number isn't known until you run the program.You will have to correct such statements manually.

RENUMBER prints a message indicating the number of computed branches it found. If it is “0 COMP BR”, renumbering was successful. For example

```
:10 FOR A =1 TO 5
:15 N(A) = 0
:20 NEXT
:25 GOTO 10
```

```
:RENUMBER 20
0 COMP BR

:LIST
:20 FOR A=1 TO 5
:40 N=0
:60 NEXT
:80 GOTO 20
```

The program has been successfully renumbered. The statement "25 GOTO 10" has become "80 GOTO 20" since the old line 10 has been renumbered 20. The “0 COMP BR” message indicates that "GOTO 10" is not a "computed branch" because BASIC could determine the new line to go to.

Now consider the same program but with "GOTO 10" replaced by "GOTO 5*2" (or "GOTO B")

```
:10 FOR A=1 TO 5
:15 N(A)=0
:20 NEXT
:25 GOTO 5*2
:RENUMBER 20

1 COM BR                    (uh-oh... a computed branch that BASIC can't renumber!)
:LIST
:20 FOR A=1 TO 5
:40 N(A)=0
:60 NEXT
:80 GOTO 5*2                (...and here it is!)
```

The message warns that there is one computed branch. One line (in this case, line 80) will cause an error because of the renumbering. "GOTO 5*2" will try to go to line 10, but there **is** no line 10! You have to correct line 80 yourself. This can be very difficult if you didn't keep a copy of the original program!

RESTORE

Format: RESTORE

This statement resets the BASIC DATA pointer back to the start of the first DATA statement in the program, thus allowing you to reuse the DATA again in one program.

RETURN

Format: RETURN

This statement RETURNs from a GOSUB subroutine call. Use it at the end of a routine called by GOSUB. Execution returns to the statement after the last GOSUB executed. See the example for the GOSUB command.

RND

Format: var = RND
var = RND (expr)

RND (expr) returns a random integer number greater than or equal to zero and less than the value of the expression. However, if the random integer number generated is assigned to a variable, the variable must be defined as an integer first, (e.g. DEFINT A).

RUN

Format: RUN

This statement tells BASIC to execute the program in memory. RUN (with no expression) clears all array and string data space to make room for new data, and begin execution at the lowest line number, and executes each subsequent line in numerical order.

RUN expr begins execution at the line number specified by the expression. It does **not** clear the data space, so RUN expr can be used to resume an interrupted program with the data still intact. If the line number specified by expr does not exist, an error code is generated.

RUN+

Format: RUN+

This statement searches the program and replaces "interpretive branches" with "absolute address branches" and then starts execution. This greatly increases execution speed. After the initial RUN+ cycle, subsequent RUN commands will also execute in this faster mode.
If the program is edited or reloaded, BASIC automatically goes back to the slower mode.
A program converted by RUN+ should not be saved in this form because of the absolute address assignments. Edit the program first to return it to the normal slower mode before saving it.

RUN+ replaces the line number in GOTO and GOSUB statements with an absolute memory address where the line is stored, so BASIC can jump there directly without interpreting it. A statement such as "GOTO B" is not an "interpretive branch" and will not be replaced, since RUN+ doesn't know the value of the variable B.

SGN

Format: var = SGN (expr)

The value of this function is either +1, 0, or -1 depending on the sign of expr. If expr is positive, SGN returns +1. If expr is zero, SGN returns 0. If expr is negative, SGN returns -1.

SIN

Format: var = SIN (expr)

This function returns the sine of the angle determined by the value of the expression (normally expressed in radians). This function is a floating-point function. For example

```
:PR SIN (PI/6)
0.5
```

SQR

Format: var = SQR (expr)

This function returns the square root of the value of the expression. This function is floating point. Example

```
:PR SQR (2)
1.41422
```

STQ

Format: STQ expr

This function sets the 1802 Q flag and output pin to the value of the expression. Only the least significant bit of the expression is used.

```
:STQ 0          (reset Q=0)
:STQ 1          (set Q=1)
:STQ 2          (reset Q=0, since the least significant bit of 2 is 0)
```

TAB

Format: TAB (expr)

The TAB function does not return a value of any kind. It is used and recognized only in PRINT statements. TAB moves the cursor to the horizontal position determined by the value of the expression. The new cursor position is referenced to column 0 and not to the previous cursor position. Printing continues from that point.

Here's an example:

```
:10 FOR X=1 TO 60
:20 WAIT(20): PR TAB(X); " /\";          (first shape of worm)
:30 WAIT(20): PR TAB(X); " ___";         (second shape of worm)
:40 NEXT
:RUN                                     (displays a little worm crawling across the screen)
```

TRACE

Format: TRACE (X)
TRACE X

If X evaluates to anything other than zero, "trace" action is turned on. Each line executed by BASIC will send the following to the screen (in addition to whatever PRINT and INPUT statements send):

```
TR line number
```

The numbers on the screen are the line numbers of each statement as it is executed. This enables the user to follow or "trace" the flow of the program, and is especially useful in the debugging stage to ensure that the program is doing what is intended.

If a TRACE statement is executed with X evaluating to zero, “trace” is turned off. TRACE statements may be placed anywhere in a BASIC program. A very useful place for a TRACE statement is in an interrupt routine.

Try the following with a BASIC program.

```
:TRACE (1): RUN
```

USR

Format: var = USR (exprl)
var = USR (exprl, expr2)
var = USR (exprl, expr2, expr3)

This function acts like a CALL statement, but USR is a function that can be used as part of an expression. When USR is encountered, a subroutine call is made to the machine language routine stored at expr1. Data may be passed to the subroutine in exactly the same way as the CALL statement. See CALL for details on the rules for writing the machine language routine.

WAIT

Format: WAIT (expr)

This statement inserts a delay into the execution of a program. The length of the delay is directly proportional to the value of the expression. 1 is approximately 10 milliseconds with a 1.8MHz 1802 clock speed. Example

```
:10 FOR X=1 TO 10
:20 STQ X                                (toggles Q alternately on/off, since only the least significant bit is used)
```

```
:30 WAIT (100)                  (wait about 1 second)
:40 NEXT                        (repeat 10 times, so Q blinks 5 times)
:RUN
```

OPERATORS

Arithmetic operators

-	unary minus	highest priority
^	exponent	first priority
*****	multiply	second priority
/	divide	second priority
+	add	third priority
-	subtract	third priority
( )	parenthesis	alter the priority
=	equal	assignment symbol

Relational operators

=	Equality
<>	Inequality (not equal to)
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Logical operators

AND
XOR
OR
NOT

Miscellaneous operators

:	Separates multiple statements on the same line.
;	(in PRINT statement) no extra space to next PRINT item. No <cr><lf> if **;** is at end of line.
,	(in PRINT statement) Tab to next column for next PRINT item. No <cr><lf> if **,** is at end of line.
"..."	String delimiter.
(...)	Group items or expressions.
#	Convert 8-bit hexadecimal number into decimal (number must be TWO digits).
@	Convert 16-bit hexadecimal number into decimal (number must be FOUR digits).

SYSTEM MEMORY USAGE

BASIC3 uses a number of RAM locations for storing pointers, system variables, BASIC programs, data, and variables. Usage for some of these addresses is explained below. All addresses are in HEXADECIMAL.

MCBASIC3 (org 0 ROM)	**MCSMP20** (org 8000 ROM)	**Usage**
0000-3FFF	8000-FFFF	The BASIC3 EPROM itself
4FFF		BASIC Autorun flag; =FF if no autorun program in ROM, or =00 if you put a BASIC program in ROM from 5000-7FFF
8000-FFFF	0000-7FFF	RAM (2k minimum, 32k maximum)
8000-80FF	0000-00FF	Protected RAM (used by BASIC and its I/O routines)
8000-8001	0001-0002	Restart vector that BYE jumps to
8002-8003	0003-0004	Serial I/O control parameters (do not change)
8005	0005	Break detect flag: 0=check serial input for Break, 1=don't check
8006	0006	Full/half duplex flag: 0=full duplex, 1=half duplex
8007	0007	Reserved: used by PLOAD command
8008	0008	Front Panel LEDs (OUT4 port): 0=show serial inputs, 1=don't show
8100-815F	0100-015F	Line buffer
8181-8182	0181-0182	DEFUS (Define Start of User Space) address
8183-8184	0183-0184	EOP (End Of Program) address
818A-818D	018A-018D	CALL to Input routine
818E-8191	018E-0191	CALL to Output routine
8192-8193	0192-0193	End of arrays / start of string values
8194-8195	0194-0195	Start of arrays
8199-819A	0199-019A	EOD (End Of Data) and end string value
819B	019B	X value for TAB
819D	019D	Flags
819E	019E	Start page address of User RAM
819F	019F	Call to BASIC functions
81A3-81A6	01A3-01A6	Call to BASIC functions
81B1-81B2	01B1-01B2	DATA pointer
81BB	01BB	End page address of RAM
81D0-8268	01D0-0268	Converted line buffer (note: this may overwrite A-Z variable storage!)
8200-8268	0200-0268	Storage for variables A, B, C ... Z (4 bytes per variable)
8300	0300	Start of user BASIC program

ERROR MESSAGES

The following Error Messages are most likely valid in some form for all BASIC versions.

00 Program HALTED by USER
01 Syntax error in ASC or LEN function
02 ARRAY out of RANGE or NOT DIMENSIONED
03 DIMENSION error
04 DEFINT has illegal ending
05 PARENTHESES missing in ARGUMENT
06 ARGUMENT out of range
07 MIXED MODE calculation encountered
08 DIVIDE by ZERO error, or LOG of NEGATIVE NUMBER

09 NON-EXECUTABLE function encountered
10 EXIT command must be used with FOR/NEXT or GOSUB/ RETURN
11 FOR/NEXT stack overflow, or FOR/NEXT executed directly.
12 Syntax error in FOR
13 GOSUB stack overflow
14 UNACCEPTABLE character in HEXADECIMAL number
15 Floating point number too large to be converted to integer or integer multiply overflow
16 UNACCEPTABLE OPERATOR in CONDITIONAL statement
17 INPUT or FVAL cannot be directly executed
18 Must have VARIABLE or STRING name in READ
19 Syntax error in READ or INPUT
20 Syntax error in LEN function
21 Syntax error in ASSIGNMENT statement
22 Missing QUOTE
23 Syntax error in LIST
24 No such WORD found in LIBRARY / illegal command
25 Syntax error in MID$ function
26 UNACCEPTABLE variable name found in NEXT statement
27 Either a NUMBER or a LETTER is EXPECTED
28 Missing arithmetic PARENTHESES
29 Wrong number of ARGUMENTS in POKE statement
30 UNACCEPTABLE last character in PRINT statement
31 Syntax error in DATA statement
32 No more DATA found
33 No such STRING found in INPUT statement
34 Missing EQUAL sign in ASSIGNMENT statement
35 Missing PARENTHESES in STRING ARRAY
36 Too many ARGUMENTS in USR or CALL
37 Syntax error in CHR$ function
38 UNACCEPTABLE character in BINARY number
39 Line buffer OVERFLOW
40 File not opened for INPUT
41 File not opened for OUTPUT
42 UNACCEPTABLE line end or NON-EXECUTABLE statement
43 STACK OVERFLOW
44 Too many DIGITS in number
45 UNACCEPTABLE character in NUMBER fold
46 No such LINE NUMBER found
47 UNACCEPTABLE operation in IF statement
48 MEMORY OVERFLOW
49 Wrong number of arguments in MOD statement
50 Program TOO LARGE for memory
51 ARGUMENT out of RANGE
54 STRING variable not defined
55 TAPE READ error
56 TAPE WRITE error
57 FILE is not a BASIC program
58 FILE is not BASIC data
63 Not enough MEMORY for RENUMBER to OPERATE
64 RENUMBER located LINE NUMBER error