

BASIC PROGRAMMING GUIDE

COPYRIGHT

Copyright 2009 - Remote Processing Corporation.
All rights reserved.

The software described in this manual is furnished
under license.

The contents of this manual and the specifications
herein may change without notice.

PRODUCT SUPPORT

If you have a question about the Basic in this manual
and cannot find the answer, call us at the number
listed below during normal business hours.

When you call, please have the following at hand:

This programming guide
Your card hardware manual
A description of the problem

Remote Processing Corporation
7975 E. Harvard Avenue
Denver, Co 80231

Phone: 303 690 1588
Fax: 303 690 1875
email: getinfo@rp3.com
Internet: www.rp3.com

Document order # 2475

Revision 1.1

BASIC PROGRAMMING GUIDE

TABLE OF CONTENTS

PREFACE	1	EXP	18
MANUAL CONVENTIONS	1	FOR-TO-STEP-NEXT	19
Symbols and Terminology	1	FREE	21
Basic Interpreters	2	GET	22
ELEMENTS OF A BASIC PROGRAM	2	GOSUB	23
Commands	2	GOTO	24
Functions	2	IF THEN ELSE	25
Line Numbers	2	INPUT	26
Operators	2	INT	27
Tasking Statements	2	LD@	28
Expressions	2	LEN	29
WRITING AND EDITING PROGRAMS	2	LIST	30
Uppercase/Lowercase	3	LOG	31
Variables and Constants	3	MTOP	32
Subroutines	4	NEW	33
Addresses	5	NULL	34
Arrays	5	ONERR	35
Strings	5	ON GOSUB	36
OPERATING MODES	6	ON GOTO	37
Command and Run Modes	6	ONTICK	38
Autorunning Programs	6	PI	39
Stopping Program Execution	6	POP	40
STORING PROGRAMS	6	PH0.	41
OPERATORS	6	PH1.	41
Operator Precedence	6	PRINT	42
ARITHMETIC OPERATORS	6	PRINT #,	42
BASIC-52 COMMANDS & FUNCTIONS	1	P.	42
ABS	1	?	42
ASC	2	PROG	44
ATN	3	FPROG	44
CBY	4	PUSH	45
CHR	5	RAM	46
CLEAR	6	READ	47
CLEAR S	6	REGREAD	48
CLEAR TICK	8	REGWRITE	49
CONT	9	REM	50
COS	10	RESTORE	51
CR	11	RETI	52
DATA	12	RETURN	53
DBY	13	RND	54
DIM	14	ROM	55
DO-UNTIL	15	RROM	56
DO-WHILE	16	SGN	57
END	17	SIN	58
		SPC	59
		STOP	60
		STR	61
		STRING	65
		SQR	66
		ST@	67
		TAB	68
		TAN	69
		TICK	70
		USING	71

BASIC PROGRAMMING GUIDE

U.	71
XBY	72
APPENDIX A- ERROR MESSAGES	1
A-STACK	1
ARITH. UNDERFLOW	1
ARITH. OVERFLOW	2
ARRAY SIZE	2
BAD ARGUMENT	2
BAD SYNTAX	2
C-STACK	2
CAN'T CONTINUE	2
DIVIDE BY ZERO	2
I-STACK	3
MEMORY ALLOCATION	3
NO DATA	3

BASIC PROGRAMMING GUIDE

PREFACE

This programming guide is for Remote Processing CX-10 or similar controllers using a variation of BASIC-52 language. It was derived from Intel MCS-51 BASIC, V1.1. Several command extensions and features have been added to effectively speed up command execution.

- Buffered serial ports. Received characters are buffered to 256 characters. PRINT strings are put into a 256 character buffer, making it much faster.
- Modbus protocol support
- Some control and timing functions such as TICK(0) and ONTICK

Some cards do not have all hardware features so do not support all of the commands. Cards supported or exceptions are listed with each command. In some cases you must refer to your hardware manual for exact ranges.

A few original BASIC-52 commands have been removed. These commands were oriented around specific registers in the 8052 chip or a specific design.

MANUAL CONVENTIONS

Information appearing on your screen is shown in a different type.

Example:

```
RPBASIC-52 V1.0
Copyright Remote Processing (1995)
Bytes free: 27434
```

Symbols and Terminology

<xxx> Paired angle brackets are used to indicate a specific key on your keyboard. For example, <esc> means the escape key.

expr Term meaning a number, simple variable, or mathematical expression involving variables and numbers. The following are valid *expr*:

```
45.3
B
CYCLE
B*45
C*D+54
INT(D)
```

expr can be another function. Complexity of *expr* is limited by available stack memory. Usually this is 7 levels of parentheses.

For clarity, *expr* may be another name such as *position*, *channel*, and so on.

italic Italicized variables require an expression or value. For example:

```
AIN(channel)
KEYPAD(function)
```

Ellipsis (...) follow an instruction which optionally accept more data.

```
DATA data[,data][,data]...
READ variable[,variable]...
```

Optional portions of an instruction are enclosed in brackets []:

```
DISPLAY option[,option][,option]
```

BASIC PROGRAMMING GUIDE

Basic Interpreters

There are several types and levels of interpreters. A slow, very basic type of interpreter figures out what each command is supposed to do during run time. A token-based interpreter, such as this basic, is much faster. This type examines each program line as it is typed in, figures out what it should do, and converts it to a string of Basic tokens mixed with text. A token is a single character that represents a command. For example, an ASCII value of 89H represents the PRINT command.

After a line is processed, it is stored in memory. When you type the RUN command, each program line is scanned. A token causes a branch to an assembly language routine which carries out the required action.

ELEMENTS OF A BASIC PROGRAM

Commands

Commands direct or perform an output action. Examples are PRINT, SAVE, POKE, and LOAD. Commands do not return a value used for computation.

Functions

Functions return a value used for computation. Examples are REGREAD, SIN, and ABS. Functions do not cause a change in an output.

Line Numbers

Program lines begin with a unique line number. Each line number may contain one or more Basic statements separated by a colon. Line numbers are in the range of 1 - 65535.

Operators

Operators act on or convert numeric or string data. These include arithmetic (+, -, *, and /), natural logarithmic (base "e"), trig (SIN, COS), relational (>, <, or <>), logical (.AND., .OR., .XOR.), and string (ASC) functions. Special operators control the hardware-specific features of Basic such as interrupts, timers, counters, and direct read/write of I/O ports.

Tasking Statements

Tasking statements define a condition and execution location when a condition is met. Statements include ONTICK. Programs executed as a result of this statement is treated as subroutines. The only

difference between a tasking routine and one called by a GOSUB is the tasking can be called at any time.

Expressions

An expression is a combination of instructions, operators, data (constants, arrays or strings) and variables which, when evaluated by Basic, is equivalent to a single numerical value. Many Basic commands accept expressions as well as explicit data. Expressions which are used by commands and functions are also called arguments.

WRITING AND EDITING PROGRAMS

Program development takes place on your PC using your word processor or the RPC card. Programs from your PC are downloaded using a serial communication program.

Each program line can contain at most 79 characters. Program lines can be entered in any sequence. Basic properly orders line numbers.

Multiple statements on a single line are allowed when statements are separated by colon (:) and do not exceed a total of 79 characters per program line. Ending a program line with a colon may cause a program to hang.

There are two ways to write Basic programs. The first way is to directly type in the program to the card. All standard Remote Processing cards have a means of storing programs to a flash type EPROM. The second way is use a text editor and download the resulting file to the system. Just be sure to save files in DOS text format.

Downloading programs means transferring them from your PC (or MAC or terminal) to the card. Uploading means transferring them from the card back to the PC.

When uploading or downloading files, select ASCII text format. XMODEM, YMODEM, or other formats are not used. Basic does not know when you are typing in a program or if something else (laptop or mainframe) is sending it characters. The upload and download file does not contain any special codes; they are simply ASCII characters.

Uploading programs is simply a process of receiving an ASCII file. You or your program simply need to send "LIST" to receive the entire program.

BASIC PROGRAMMING GUIDE

Downloading a program requires transmitting an ASCII file. As you type in (or download) a line, Basic tokenizes that line. The time to do this depends upon its complexity and how many lines of code have been entered.

Basic must finish compiling a line before starting the next one. When a line is compiled, a ">" character is sent. This should be your terminal programs pacing character when downloading a program.

If your communications program cannot look for a pacing prompt, set it to delay transmission after each line is sent. A 100 ms delay is usually adequate, but your program may be long and complex and require more time. A result of short transmission time is missing or incomplete program lines.

A technique used to further program documentation and reduce code space is the use of comments (REM) in a downloaded file. For example, you could have the following in a file written on your editor:

```
REM Check position

REM Read output from the pot and
REM calculate the position

2200 a = ain(0) :REM Get position
```

The first 3 comments downloaded to the card are ignored. Similarly, the empty lines between comments are also ignored. Line 2200, with its comment, is a part of the program and could be listed. The major penalty by writing a program this way is increased download time.

Notice that you can write a program in lower case characters. Basic translates them to upper case.

Some programmers put "NEW" as the first line in the file. During debugging, it is common to insert "temporary" lines. This ensures that these lines are gone. Downloading time is increased when the old program is still present.

If you like to write programs in separate modules, you can download them separately. Modules are assigned blocks of line numbers. Start up code might be from 1 to 999. Interrupt handling (keypad, serial ports) might be from lines 1000 to 1499. Display output might be from 1500 to 2500. The

programmer must determine the number of lines required for each section.

Basic automatically formats a line for minimum code space. For example, you could download the following line of code:

```
10 for a=0to5
```

When you listed this line, it would appear as:

```
10 FOR A=0 TO 5
```

Spaces are displayed but not stored. The following line:

```
10 for a = 0 to 5
```

would be compressed and displayed as in the second example above. Spaces are removed. However, spaces as part of a remark or PRINT are not removed.

Basic contains no line renumbering capability.

Basic contains a rudimentary line editor which allows editing a program line until a carriage return is sent. The rubout or backspace key can be used to delete characters working backwards from the current character. After a line is entered, it cannot be edited; you must enter an entire new line. Deleting an undesired line is done by typing the line number followed by a carriage return. Basic automatically deletes all such lines.

Uppercase/Lowercase

Basic is generally not case-sensitive. Program or command lines may be entered in lowercase or uppercase; however they are (with some exceptions) converted to uppercase. The case of text in remarks and strings is preserved.

Variables and Constants

More than 25,000 unique variable or constant names may be defined. Names may be up to eight characters in length and must begin with a letter between A-Z (no numbers or special characters). The rest of the name may contain numbers or letters and include the underline character.

All numeric variables are floating point. Variables cannot be declared as integer or double precision. Basic supports eight digits plus sign and exponent. Extra digits are simply discarded. The range of valid values is $\pm 1E-127$ to $\pm 0.99999999E+127$.

BASIC PROGRAMMING GUIDE

Names are identified by the first and last characters and its length. Identical length names with identical first and last characters are considered the same. PUMP_42 and PRIMER2 are considered the same. The way to correct this is to change the name length or first or last character.

Variable names longer than two characters require more time to process. Once a variable name is declared, it can only be erased by the CLEAR statement or by LOADING in a new program.

It is possible to have variable names longer than 8 characters. A problem is the name length is stored partly as a modulo 256 number. What it boils down to is a variable may or may not be recognized as unique. The Basic considers FEED_BIN_01 and FEED_BIN_11 as the same variable.

The original BASIC-52 had a bug where the variable name 'F' was erased if it was the last letter in a variable followed by a space. Basic corrected this.

Watch out for commands embedded in variable names. FORM_5 contains the command FOR. A BAD SYNTAX error is usually returned in these instances. The statement FORM_5=BOTTOM does not return an error but interprets it as

```
FOR M_5=BOT TO M
```

The key is to look at your statements as they are printed on the screen and make sure they are what you intended.

Valid variables names are:

```
CA5, DA15_679, PUMP_A, VALVE02, A(10),  
SIZE(5), ABC_
```

Invalid variables, which may include embedded commands include:

```
4C, C$0, GOTOE, FORM, #XYZ, _ABC
```

Constants are literal values. These are "known" values as opposed to variables which can be assigned any value, usually by a function. Constants may be numeric or string. To Basic-52, there is no difference between the two.

Constants are expressed as integer, decimal, hexadecimal or exponential floating-point. The range of valid values are:

```
± 1E-127 to ± .99999999e+127
```

Using constants instead of a number speeds up execution by at least 5%. For example, use

```
10 CH = 5  
20 A = REGREAD(CH)
```

instead of

```
20 A = REGREAD(5)
```

Variables and constants are expressed as follows:

A = 5	Integer format
A = 5.3	Decimal format
A = 0ACH	Hexadecimal format
A = 1.4E3	Exponential

Basic supports eight significant digits plus and exponent and truncates any extra digits. Hexadecimal constants with a leading alpha character must be preceded by a leading zero. If you fail to do this, Basic interprets them as variable names.

All hexadecimal constants are followed by a trailing "H" (OFFH for example). A "0" prefix is necessary when the first number is a letter (A-F).

Certain logical operators, such as .NOT., .AND., .XOR., and .OR., assume a 16-bit argument such as OFFFFH. If you supply fewer than 16 bits, it returns a 16-bit value based on the assumption the unsupplied most significant bits are zero.

Subroutines

Use of subroutines tends to make programming more modular and easier to follow. The number of subroutines is limited to the amount of internal stack space. Usually this is about 35 subroutines, but can go down if FOR-NEXT loops are active. This is sufficient to handle all multi-tasking and several levels of subroutines.

Most complex programs tend to have a maximum of 7 nested subroutine levels. Usually the maximum is 4.

Addresses

Addresses are specified as either decimal or hexadecimal numbers. Hexadecimal addresses with a leading alpha character need a preceding zero otherwise they will be interpreted as variable names.

Arrays

Arrays are single dimension and start with element 0. They are dimensioned using the DIM statement.

BASIC PROGRAMMING GUIDE

Each variable may have up to 255 elements (0 to 254). Un-dimensioned arrays default to 11 elements, *variable*(0) through *variable*(10). Naming conventions used for scalar variables apply to arrays.

Strings

Memory is allocated to strings using the STRING command. There is no power up default. Up to 255 strings, identified as \$(0) through \$(254) are available.

To use strings, you must first determine the maximum length of any one string and then the maximum number of strings. Using the formula

$$(\text{bytes/string} + 1) * \text{number of strings} + 1$$

returns the number of bytes to allocate.

The ASC and CHR commands are used to evaluate and manipulate strings. Text assigned to a string is enclosed in double quotation marks:

```
100     STRING 1000,40
110 $(0)=">03"
```

BASIC PROGRAMMING GUIDE

OPERATING MODES

Command and Run Modes

Basic operates in two modes, Command and Run. Command mode is the direct, interactive mode accessed when Basic is not running a program. The Basic console prompt ">" indicates that Basic is ready for Command mode input.

Run mode is when the processor is actively executing a Basic program. Some commands (such as SAVE, LIST, LOAD) can only be executed when the processor is in command mode. Most Basic instructions can be executed in either Command or Run mode.

In Command mode, LOAD selects a Basic program from the flash. The RUN command then causes the selected program to execute. Within a Basic program, the EXECUTE instruction is used to allow the currently running program to call another stored program. A number of programs may be available to run depending upon the card and flash EPROM size installed. Refer to your hardware manual for more information.

Autorunning Programs

Programs may automatically load and run on powerup or reset when FPROG2 is executed.

Stopping Program Execution

<Ctrl-C> halts the execution of a program and forces the processor into Command mode (unless <Ctrl-C> has been disabled). Operation can be resumed by typing the CONT command. The STOP instruction stops a running program; execution resumes with a CONT command.

Sometimes it is desirable to not stop program execution. To disable <Ctrl-C>, execute:

```
DBY(38) = DBY(38) .OR. 1
```

STORING PROGRAMS

Basic programs are stored in non-volatile flash type EPROM on the CPU. The FPROG and PROG commands are used to write programs.

OPERATORS

Operator categories include:

Arithmetic	=, +, *, /, **, SQR
Relational	=, <>, <, >, <=, >=
Logical	.AND., .OR., .XOR., .NOT.
Value	ABS, INT, PI, RND, SGN

Operator Precedence

The precedence of operators determines the order in which mathematical operations are executed. Basic scans an expression from left to right and performs no operations until it encounters an operator of lower or equal precedence. For instance, multiplication takes precedence over addition. Parenthetical expressions have the highest precedence.

The following list is Basic's order of precedence:

1. Operators in parenthesis
2. Exponential operators (**)
3. Negation (-)
4. Multiplication (*) and division (/)
5. Addition (+) and Subtraction (-)
6. Relational expressions (=, <>, <=, <, >=, >)
7. .AND. (logical AND)
8. .OR. (logical OR)
9. .XOR. (logical XOR)

Parenthetical expressions have the highest precedence, so their use is a good way for you to reduce ambiguity and make your programs more readable. However, parenthetical expressions use internal data memory.

ARITHMETIC OPERATORS

Arithmetic operators perform basic arithmetic functions:

+	addition
-	subtraction, not negation
*	multiplication
/	division
**	exponential

BASIC PROGRAMMING GUIDE

ABS

Syntax: ABS(*expr*)

Where: *expr* = any number in Basic's range

Function: Returns the absolute value of an expression

Mode: Command, run

Use: PRINT ABS(C)

DESCRIPTION

The absolute value of a number is always positive or zero.

BASIC PROGRAMMING GUIDE

ASC

Syntax: *ASC(ASCII character)*
 ASC(string,position expr)
Where: *ASCII character* = number from 0 to 255
 string = any valid string variable
 position expr = 1 to length of string

Function: Returns or sets the integer value of an *ASCII character* or the character in *string* at *position expr*.

Mode: Command, run

Use: PRINT ASC(C)
 ASC\$(3,1)=48H
 C = ASC\$(0,P)

DESCRIPTION

The ASC operator either sets or returns the value of an ASCII character. Use ASC to evaluate, change or manipulate individual characters in a string.

The first syntax returns the value of an ASCII character. If *ASCII character* were the letter 'B', a 66 is returned. Basic converts any lower case variable symbols to upper case. Lower case characters must be put into a string to be evaluated.

The second syntax, shown under Use, sets a character in a string to a specific value. This is useful when you want to manipulate individual characters in a string.

The third syntax returns a value in *string* at *position expr*. This form is useful when you want to evaluate individual characters in a string, such as generating a checksum.

RELATED

CHR, STRING

ERROR

SYNTAX Attempt to convert an improper value.

EXAMPLE

The following example prints ASCII values from the string \$(0). The first 3 characters are modified at lines 70 to 90. The result is then printed.

```
10  STRING 200,20
20  $(0)="abc123"
30  FOR N=1 TO 6
40  PRINT ASC$(0),N),
50  NEXT
60  PRINT
70  FOR N=1 TO 3
80  ASC$(0),N)=65+N
90  NEXT
100 PRINT $(0)

READY
>RUN
  97  98  99  49  50  51
BCD123
```

BASIC PROGRAMMING GUIDE

ATN

Syntax: `ATN(expr)`

Where: *expr* = value between 0 and $\text{PI}/2$

Function: Returns a trigonometric arc-tangent of *expr*. Returned result is between $-\text{PI}/2$ and $\text{PI}/2$ radians.

Mode: Command, run

Use: `PRINT 4*ATN(1)`

DESCRIPTION

SIN, COS, and TAN operators use a Taylor series to calculate the function. These operators first reduce the argument to a value that is between 0 and $\text{PI}/2$. The algorithm used to reduce the value will reduce accuracy when *expr* is large. To maintain accuracy, keep the arguments for trig functions as small as possible.

ERRORS

ARITH. UNDERFLOW *expr* or result is less than BASIC-52's smallest floating-point value of $\pm 1\text{E}-127$

ARITH. OVERFLOW *expr* or result is greater than BASIC-52's largest floating-point value of $\pm .9999999\text{E}+127$

DIVIDE BY ZERO Attempt to take $\text{TAN}(X)$ when $\text{COS}(\text{PI}/2) = 0$

EXAMPLES

```
100 PRINT SIN(PI/2),COS(10001*PI),TAN(5*PI/4)
110 PRINT ATN(TAN(PI/4))/PI
```

>run

```
1 -1 1
.24999996
```

BASIC PROGRAMMING GUIDE

CBY

Syntax: CBY(*expr*)
 Where: *expr* = address from 0 to 65535
Function: Reads internal program code
Mode: Command, run
Use: PRINT CBY(1000H)

DESCRIPTION

The CBY instruction reads data from program memory space in the 8052. *expr* must evaluate to a valid integer address of 00H through 0FFFFH (65535). Code memory is read-only.

RELATED

DBY, XBY, PEEK, POKE

ERROR

BAD ARGUMENT *expr* must be a valid integer (0 through 65535).

EXAMPLE

```
10  FOR N=0 TO 10
20  PRINT CBY(N),
30  NEXT
```

```
>RUN
97  203  255  210  22  50  2  39  110  255  255
```

BASIC PROGRAMMING GUIDE

CHR

Syntax: CHR(*expr*)
 CHR(*string,position*)
 Where: *expr* = number from 0 to 255
 string = string variable
 position = 1 to length of string

Function: Converts *expr* to ASCII character or prints *string* at *position*

Mode: Command, run

Use: PRINT CHR(65)
 PRINT CHR\$(0),1

DESCRIPTION

CHR is a dual use operator, similar to ASC. One version converts a numeric expression to an ASCII character, allowing a variety of string manipulation operations. The second version uses CHR to print individual characters in an ASCII string. *expr* is a decimal number and truncates numbers from 0 through 65535. There must be no space between CHR and the left parentheses or an ARRAY SIZE error results. Although *expr* can be any integer, printable ASCII characters range from 20H through 7EH (32 through 127).

RELATED

ASC, STRING

ERRORS

BAD ARGUMENT *expr* can't be truncated to an integer (0 through 65535)
ARRAY SIZE space between CHR and left parentheses

EXAMPLE

```
10   STRING 200,20
20   $(1)="1234567890"
30   FOR N=64 TO 80
40   PRINT CHR(N),
50   NEXT
60   PRINT
70   FOR N=1 TO 9
80   PRINT CHR$(1),N),
90   NEXT
```

```
RUN
@ABCDEFGHIJKLMNPO
1234567890
```

BASIC PROGRAMMING GUIDE

CLEAR

CLEAR S

Syntax: CLEAR
 CLEAR S

Function: Sets variables to zero, clears stacks

Mode: Command, run

Use: CLEAR
 CLEAR S

DESCRIPTION

The CLEAR instruction sets all variables to 0 and resets all Basic stacks. ONERR is cleared. Error trapping must be redeclared after a CLEAR. CLEAR is generally used to clear all variables. CLEAR does not de-allocate memory allocated to strings by the STRING instruction. It does clear the contents of the strings. Data put to the stack by PUSH is cleared. CLEAR also resets any FOR-NEXT loops. A C-STACK error is returned when a NEXT is performed after a CLEAR. CLEAR also resets any GOSUB return addresses.

Use CLEAR to perform a soft reset of a program. Keep in mind that multi-tasking routines are not cleared or reset using this command. However, if CLEAR is used as part of a multi-tasking program (ON COM\$, ON LINE, etc.), a RETURN will cause a C-STACK error.

CLEAR S resets the control stack (C-STACK) only. This stack is used in loops and subroutines to tell it where to return to. Use this command to branch (GOTO) out of FOR-NEXT, GOSUB-RETURN, DO-UNTIL type structures. It can be used in emergency stop situations where nesting of loop structures is not known. Variables are not cleared using CLEAR S.

RELATED none

EXAMPLE

```
10   CLEAR TICK(0)
20   ONTICK 1,1000
25   ONERR 500
30   IF TICK(0)<2.5 THEN 30
40   A=TICK(0)/0
50   IF TICK(0) < 3.3 THEN 50
60   CLEAR
70   PRINT "CLEARED"
80   GOTO 80
500  PRINT "IN ERROR"
510  ONERR 500
520  GOTO 50
1000 PRINT TICK(0),A
1010 A=A+1
1020 RETI
```

```
>RUN
1  0
2  1
IN ERROR
3  2
4  0
5  1
6  2
```

BASIC PROGRAMMING GUIDE

The above example shows that ONTICK continues to run after a CLEAR statement but variables are cleared. If a program error were generated after the clear, the program would stop because ONERR was cleared.

The next example demonstrates how CLEAR S can be used in a FOR-NEXT loop. A C-STACK error is returned if the CLEAR S is not in line 20.

```
10  FOR N=0 TO 10
20  IF N=5 THEN CLEAR S : GOTO 10
30  PRINT N
40  NEXT
```

>RUN

```
1
2
3
4
0
1
```

BASIC PROGRAMMING GUIDE

CLEAR TICK

Syntax: CLEAR TICK(*timer*)
Where: *timer* = 0 to 3

Function: Resets specified tick timer
Mode: Command, Run
Use: CLEAR TICK(1)

DIFFERENCES FROM BASIC-52

The TICK function replaced TIME as a process clock. See TICK function for more information.

DESCRIPTION

There are four independent tick timers that can be cleared independently of each other. This statement resets any one of the four tick timers to 0.

RELATED

TICK

ERRORS

BAD SYNTAX Any parameters left out
BAD ARGUMENT When *timer* > 3 or negative

BASIC PROGRAMMING GUIDE

CONT

Syntax: CONT
Function: Continue program execution after a STOP or Command-C
Mode: Command
Use: CONT

DESCRIPTION

CONT resumes program execution following a <Ctrl-C> or STOP instruction. You can display or modify variables while the program is stopped, but you cannot continue a program that is modified.

RELATED

STOP, GOTO, RUN

ERROR

CAN'T CONTINUE When program was modified.

BASIC PROGRAMMING GUIDE

COS

Syntax: `COS(expr)`

Where: *expr* = numeric value up to $\pm 200,000$

Function: Returns the trigonometric cosine of *expr* which is in radians.

Mode: Command, run

Use: `PRINT COS(PI)`

DESCRIPTION

SIN, COS, and TAN operators use a Taylor series to calculate the function. These operators first reduce the argument to a value that is between 0 and $\text{PI}/2$. the algorithm used to reduce the value will reduce accuracy when *value* is large. To maintain accuracy, keep the arguments for trig functions as small as possible.

ERROR

ARITH. UNDERFLOW *value* or result is less than BASIC-52's smallest floating-point value of $\pm 1\text{E}-127$

ARITH. OVERFLOW *value* or result is greater than BASIC-52's largest floating-point value of $\pm .9999999\text{E}+127$

DIVIDE BY ZERO Attempt to take $\text{TAN}(X)$ when $\text{COS}(\text{PI}/2) = 0$

EXAMPLES

```
10 PRINT SIN(PI/2),COS(10*PI),TAN(8*PI/4)
20 PRINT ATN(PI)
```

>run

```
1 1 0
1.2626272
```

BASIC PROGRAMMING GUIDE

CR

Syntax: PRINT CR,
Function: Used with PRINT. Sends a carriage return without a line feed.
Mode: Command, run
Use: PRINT CR,

DESCRIPTION

Used to update a line on a serial console device. A comma is necessary to prevent the usual line feed from terminating the PRINT instruction.

RELATED

PRINT

EXAMPLE

```
100 PRINT TICK(0),CR,  
110 GOTO 10
```

```
>run  
3.242
```

The number is continuously printed at the same position.

BASIC PROGRAMMING GUIDE

DATA

Syntax: DATA *expr* [,*expr*,...]
Where: *expr* = numeric data.
Function: It is an expression list used by READ.
Mode: Run
Use: DATA 23.4,17,3.2,PI*3

DESCRIPTION

Elements of a DATA statement are sequentially retrieved by the READ instruction. Multiple DATA expressions on a single program line must be separated by commas. There must be no spaces between *expr* and the commas.

See RESTORE for more information and examples.

RELATED

READ, RESTORE

BASIC PROGRAMMING GUIDE

DBY

Syntax: A=DBY(*expr*)
 DBY(*expr*)=*variable*
 Where: *expr* = 0 to 255
 variable = 0 to 255

Function: Read/write internal data memory.

Mode: Command, run

Use: DBY(0F0H) = 45H
 A=DBY(100)

DESCRIPTION

The DBY instruction retrieves or assigns a value to the 8052 internal data memory. *expr* and *variable* must both be between 0 and 255 since there are only 256 internal memory locations and one byte can only be between 0 and 255.

BASIC-52 uses many internal memory locations for its own use. Change internal memory with caution or Basic may malfunction. Locations 1BH through 21H may be used in any way you wish.

RELATED

CBY, XBY

ERROR

BAD ARGUMENT Invalid *expr* value, such as DBY(256) or attempt to assign an invalid value to a DBY(*expr*), such as DBY(18H)=1000.

EXAMPLE

```
100  DBY(1EH) = 234
110  PRINT DBY(1EH)
```

```
>run
```

```
234
```

BASIC PROGRAMMING GUIDE

DIM

Syntax: DIM *name(size)*[,*name(size)*...]
Where: *name* = Any valid variable name
size = 1 to 255 elements

Function: Reserves storage for single-dimension array.

Mode: Command, run

Use: DIM FLOW(200) : REM dimensions a 200 element array called FLOW

DESCRIPTION

The maximum number of array elements is 255, accessed as *name(0)* through *name(254)*. CLEAR, NEW, or RUN commands de-allocate all array storage. The default size of undeclared arrays is 10 (i.e. 11 elements). An array cannot be redimensioned after it has been dimensioned. Memory required for an array is ((integer size + 1) * 6). Array A(99) requires 600 bytes of memory. Available memory typically limits the size and number of dimensioned arrays.

RELATED

STRING, CLEAR

ERROR

ARRAY SIZE When *size* >255

EXAMPLE

```
10 DIM FLOW(200), LEVEL(200)
20 ONTICK 1,1000
30 IF PTR < 200 THEN 30
40 ONTICK 0,1000
50 FOR N=0 TO 199
60 PRINT FLOW(N),LEVEL(N)
70 NEXT
80 END
1000 FLOW(PTR)=REGERAD(7001)
1010 LEVEL(PTR)=REGREAD(3)
1020 PTR=PTR+1
1030 RETI
```

BASIC PROGRAMMING GUIDE

DO-UNTIL

Syntax: DO
 {program statements}
 UNTIL *relational expr*
 Where: *relational expr* is any logical evaluation such as =, <, >, etc.

Function: Executes a number of program statements a relational expression is true.

Mode: Run

Use: 100 A=0 : DO : A=A+1 : PRINT A : UNTIL A=4 : PRINT "Done"

DESCRIPTION

This statement always executes at least once. DO-UNTIL loops may be nested. This loop may be exited without meeting *relational expr* by executing a CLEAR or CLEAR S statement.

This statement always executes to UNTIL once. When *relational expr* is evaluated and if it is false, program flow branches back to DO. If true, program resumes at the next statement after UNTIL.

When there are no {program statements} between DO and UNTIL, and {*relational expr*} is false, the "loop" will repeat forever, or until a <ctrl-c> is typed at the console.

DO-UNTIL and DO-WHILE loops can be nested.

RELATED

DO-WHILE, FOR-TO-NEXT-STEP

ERROR

BAD SYNTAX When *relational expr* is omitted

EXAMPLE

The following program stays in a DO-UNTIL loop until a line has changed.

```
10   ON LINE 0,0,500
20   DO
30   UNTIL C=1
40   PRINT "Line 0 changed.  Is now a",line(0)
50   C=0
60   GOTO 20
500  C=1
510  RETURN
```

```
>run
Line 0 changed.  Is now a 0
Line 0 changed.  Is now a 1
Line 0 changed.  Is now a 0
```

BASIC PROGRAMMING GUIDE

DO-WHILE

Syntax: DO
 {program statements}
 WHILE {relational *expr*}

Function: Executes {program statements} while {relational *expr*} is true.

Mode: Run

Use: 100 CLEAR TICK(0) : DO : PRINT TICK(0) : WHILE TICK(0)<10

DESCRIPTION

The {program statements} between DO and WHILE are executed once, regardless of the {relational *expr*} result. At WHILE the {relational *expr*} is evaluated. If true, all {program statements} are executed again, and the test is repeated. If false, execution continues at the program statement after WHILE. DO-WHILE and DO-UNTIL loops can be nested.

RELATED

DO-UNTIL, FOR-TO-STEP-NEXT

EXAMPLE

The following program stays in a DO-UNTIL loop until a line has changed.

```
10    ON LINE 0,0,500
20    DO
30    WHILE C=0
40    PRINT "Line 0 changed.  Is now a",line(0)
50    C=0
60    GOTO 20
500   C=1
510   RETURN

>run
Line 0 changed.  Is now a 0
Line 0 changed.  Is now a 1
Line 0 changed.  Is now a 0
```

BASIC PROGRAMMING GUIDE

END

Syntax: END
Function: Terminates program execution and returns to command mode.
Mode: Run
Use: 65000 END

DESCRIPTION

The END instruction terminates Basic program execution. If no END instruction is used at the end of a program, the last instruction automatically terminates the program. Use END after the body of your program and prior to any subroutines.

Without an END after the main body of your Basic program and prior to any subroutine program lines, BASIC-52 will attempt to execute any subroutines at the end of your program as if they were a continuation of the main program. This will generate a C-STACK error whenever a RETURN is encountered.

RELATED

CONT, STOP, GOSUB, ON-GOSUB

ERROR

CAN'T CONTINUE The CONT instruction cannot follow an END instruction.

EXAMPLE

```
10 GOSUB 100
20 END
100 PRINT PI
110 RETURN
```

>run

3.1415926

If you remove line 20, a C-Stack error is returned.

BASIC PROGRAMMING GUIDE

EXP

Syntax: EXP(*expr*)
Function: Raises "e" (2.71828) to the power of *expr*
Mode: Command, run
Use: PRINT EXP(COS(1))

DESCRIPTION

This function returns the result of the number e (2.718282) raised to the power given by *expr*. This function is very computation time intensive. Small values of *expr* take about 5 milli-seconds to calculate while larger ones (near 250) require nearly 0.2 seconds. Avoid using this function in tight control or time intensive applications.

ERROR

BAD ARGUMENT When result of *expr* > 256

BASIC PROGRAMMING GUIDE

FOR-TO-STEP-NEXT

Syntax: FOR *variable*=*initial index expr* TO *index limit expr* [STEP *step expr*]
program statements
NEXT [*variable*]

Where: *variable* = any valid variable symbol
initial index expr = starting value assigned to *variable*
index limit expr = ending value of *variable*
step expr = optional increment or decrement to *variable* when repeating a loop

Function: Looping structure useful for executing a sequence of instructions a number of times.

Mode: Run, command

Use: FOR A=0 to 4000 STEP 200 : AOT 0,A : NEXT

DESCRIPTION

The FOR-TO-STEP-NEXT instruction is a loop structure common to many high level languages. It is used to perform *program statements* a number of times.

variable is a loop counter initialized to *initial index expr* at the start of the loop. A number of *program statements* are executed until NEXT is encountered. At this point the value of *step expr* is added to the value of *variable*. The resulting new *variable* value is compared to the value of *index limit expr*. If the new value of *variable* value is less than or equal to the value of *index limit expr*, all *program statements* are executed again, and the test is repeated.

program statements are always executed at least once. If *step expr* is larger than *index limit expr*, the loop executes only once.

STEP is optional. When omitted, it defaults to 1. The value of *step expr* may be positive or negative.

FOR-NEXT loops may be inside other FOR-NEXT loops. *variable* following NEXT is optional.

There are two ways to break out of a for next loop and still maintain the control stack. The first is to execute a CLEAR S command. This command also clears any subroutine return locations and DO-WHILE, DO-UNTIL loops. Another is to set *variable* to a high value within *program statements*. When a program continuously breaks out of a FOR-NEXT loop and re-declares a new loop, a C-Stack error is eventually returned.

RELATED

DO-UNTIL, DO-WHILE

ERROR

C-STACK NEXT without a corresponding FOR. This error can also appear if a number of FOR-NEXT loops were set up but were illegally branched out of or re-declared.

BASIC PROGRAMMING GUIDE

EXAMPLE

The following example gets characters as a result of a modbus query command and prints out the data.

```
rem loop to get all data
rem first data item is ID, second is function type
rem next set is data followed by CRC
210 for n = 1 to x

220 print "# of variants=",regread(4781),"  ",
230 print "Variant data=",
240 ph0. regread(4782),"  ",
250 print "COM 3 status=",regread(4783),"  ",

290 next
```

BASIC PROGRAMMING GUIDE

FREE

Syntax: FREE
Function: Returns the bytes of available in program RAM
Mode: Command, run
Use: PRINT FREE

DESCRIPTION

FREE returns how many bytes of RAM are available to the program and Basic variables. It does not return the amount of expanded RAM in 128K or 512K RAM systems. The amount of free memory is determined by the following formula:

$$\text{FREE} = \text{MTOP} - \text{LEN} - \text{system memory}$$

"system memory" on cards with two serial ports is 1791. Add 512 bytes for any additional serial ports on a card.

RELATED

LEN

ERROR

BAD SYNTAX Attempt to assign a value to FREE

BASIC PROGRAMMING GUIDE

GET

Syntax: A = GET
A = GET(n)
Where:
N = com port number 0, 1, 2, 3 See card for specifics
Function: Gets character from buffer.
Mode: Run
Use: A = GET(n)

DESCRIPTION

GET is similar to INKEY\$ in other Basic languages. GET returns the ASCII value of the character rather than the string. This feature makes it useful when receiving binary information.

When no characters are in the buffer, a value of 256 is returned.

To receive a control-C value (3), set bit 1, address 26H.

DBY(38) = DBY(38) .OR. 1

This disables program breaks when a <Ctrl-C> is received.

GET can extract characters from serial and keypad ports. Not all boards support keypads and multiple serial ports.

RELATED INPUT

BASIC PROGRAMMING GUIDE

GOSUB

Syntax: GOSUB *line number*

...
line number program statements

RETURN

Function: Transfers program control to the specified *line number*. The RETURN causes execution to resume at the program statement after GOSUB.

Mode: Run

Use: 100 FOR A=1 to 20 : GOSUB 200 : NEXT A : END
200 PRINT A, SQR(A) : RETURN

DESCRIPTION

GOSUB provides subroutine capability within BASIC-52 programs. A subroutine may be called from within another subroutine.

GOSUB saves the location of the program statement after GOSUB on the C-Stack and immediately transfers program control to *line number*. When a RETURN is encountered, program execution resumes at program statement after GOSUB.

GOSUBs can be nested. The number nesting is limited by available C-Stack RAM, but is usually enough for at least 30 routines.

RELATED

GOTO, ON-GOTO, ON-GOSUB

ERROR

C-STACK An unexpected RETURN is encountered or the number of subroutines executed was excessive.

EXAMPLE

```
10 GOSUB 100
20 PRINT "Back from routine"
30 END
100 PRINT "In subroutine"
110 RETURN
```

>run

```
In subroutine
Back from routine
```

BASIC PROGRAMMING GUIDE

GOTO

Syntax: GOTO *line number*
Function: Routes program execution to *line number*
Mode: Command, run
Use: GOTO 100

DESCRIPTION

When *line number* is the line number of an executable statement, that statement and those following are executed. GOTO can be used in the command mode to re-enter a program at a desired point.

RELATED

GOSUB, ON-GOTO, ON-GOSUB, RUN

ERROR

INVALID LINE NUMBER Specified line number does not exist.

EXAMPLE

```
100   PRINT "At line 100"  
200   GOTO 100
```

BASIC PROGRAMMING GUIDE

IF THEN ELSE

Syntax: IF *expr* [THEN] *statement(s)* [ELSE *statement(s)*]

Where: *expr* = any logical evaluation or variable
statement(s) = any number of Basic statements

Function: When *expr* is TRUE (not zero), the instruction following THEN is executed, otherwise the instruction following ELSE is executed.

Mode: Run

Use: 10 IF A<>B THEN PRINT "A=B" ELSE PRINT "A<>B"

DESCRIPTION

THEN is implied by IF. You may omit THEN. ELSE is optional. It is included when an "either - or" situation is encountered.

In the case of multiple statements per line following an IF-THEN-ELSE, Basic executes the following statements only if *expr* was true. This enables you to conditionally execute multiple statements with a single *expr* test. Remember this applies only to Basic statements separated by the {} delimiter and on the same program line.

expr can be either a logical evaluation (=, <, >, <>, .AND., .OR., .XOR., or .NOT.) or a variable. Using a simple variable as a flag can speed up program execution. The following examples illustrate different execution speeds.

```
10 A = 1000
20 CLEAR TICK(0)
30 IF A<>0 THEN A=A-1 : GOTO 30
40 PRINT TICK(0)
```

The above program takes about 1 second to execute, which translates to about 1 ms/ line for this example. If line 30 were re-written as:

```
30 IF A THEN A=A-1 : GOTO 30
```

Execution time is reduced by about 20% by taking away the "<>0" evaluation.

RELATED none

ERRORS none

EXAMPLE

```
10 A = 1
20 IF A=0 THEN PRINT "A is 0" ELSE PRINT "A is non-zero"
```

>run

Is non-zero

BASIC PROGRAMMING GUIDE

INPUT

Syntax: INPUT ["*prompt text*"] [,] [,*variable*...]
Where: *prompt text* = optional text
variable = list of variables to assign

Function: Program pauses to receive data entered from the console input.

Mode: Run

Use: 100 INPUT "Enter batch number",\$(0)

DESCRIPTION

INPUT brings in numeric and string data from the console serial port during execution. Variables are string, numeric, or both. Variables are separated by a comma. Optional *prompt text* must be enclosed in quotes.

When an optional comma precedes the first *variable*, the question mark prompt character is suppressed and data entry is on the same line as *prompt text*.

Multiple numeric data may be entered by separating individual values with commas and using <cr> on the last one. Or, each data entry may be entered using a <cr>.

Strings must be entered with a carriage return.

If you do not enter enough data or the correct type, Basic sends the message TRY AGAIN and *prompt text* after which you must enter **all** the data. If you enter too many characters for the size of allocated STRING memory, or more numeric values than were requested, Basic discards the extra data, emits the message EXTRA IGNORED, and continues execution.

RELATED

COM\$, GET, STRING

ERRORS none

EXAMPLE

```
10 STRING 200,20
20 INPUT "Enter a number, string, and 2 more numbers: ",A,$(0),B,C
30 PRINT "String:",$(0)
40 PRINT "Numbers:",A,B,C

>run

Enter a number, string, and 2 more numbers: 4,Bob
?7,9
String:Bob
4 7 9
```

BASIC PROGRAMMING GUIDE

INT

Syntax: INT(*expr*)
Function: Returns an integer portion of *expr*
Mode: Command, run
Use: PRINT INT(PI)

DESCRIPTION

The integer portion is stored as a floating point number.

RELATED none

ERRORS none

EXAMPLE

```
print int(45.67)
45

print int(-16.9999)
-16
```

To produce true rounding to the closest whole number, use the following formula:

$$A = \text{INT}(B+0.5)$$

BASIC PROGRAMMING GUIDE

LD@

Syntax: LD@ *expr*
Where: *expr* = valid integer address of 00H through 0FFFFH (65535)
Function: Retrieves a floating-point number previously saved with ST@
Mode: Command, run
Use: LD@3000

DESCRIPTION

LD@ is used in conjunction with PUSH, POP, and ST@. Use these commands to save and retrieve floating point numbers to program RAM.

NOTE: LD@ and ST@ cannot use extended RAM. Only segment 0 RAM (used for running Basic programs) is used. Use PEEKF and POKEF commands to access this memory.

WARNING: When 128K and 512K RAM are installed, all of memory is cleared on power up and reset. Do not use LD@ or ST@ to save floating point numbers in segment 0. Use POKE and PEEK type commands instead.

32K RAM systems have address 7E00H set as MTOP. This location up to 7FFFH may be used to store variables.

expr is the address in RAM of where a number is stored.

Each floating-point number requires six bytes of memory. *expr* in the ST@ and LD@ instructions specify the high address. A number is stored at locations *expr* through *expr-6*.

RELATED

ST@, PUSH, POP, PEEKF POKEF

ERROR

BAD ARGUMENT when *expr* > 65535

EXAMPLE

```
100  A=REGREAD(9)*.007234
110  PUSH A
120  ST@7F00H
.
.
300  LD@7F00H
310  POP B
320  PRINT "Analog value retrieved=",B

>run

Analog value retrieved=",B
```

BASIC PROGRAMMING GUIDE

LEN

Syntax: LEN
Function: Returns length of the current program in RAM
Mode: Command
Use: PRINT LEN

DESCRIPTION

The LEN function tells you the length of the program in RAM. LEN returns a value of 1 when no program is in RAM memory (1 is the length of the end-of-program marker).

RELATED

FREE

ERROR BAD SYNTAX Attempt to assign a value to LEN

BASIC PROGRAMMING GUIDE

LIST

Syntax: LIST
LIST *line number*
LIST *line number - line number*
Where: *line number* is a program line number
Function: Prints all or some of a program to the console.
Mode: Command
Use: LIST 10-100
Card: All

DESCRIPTION

The LIST command prints the program in RAM to the console device. LIST inserts spaces after the line number and before and after instructions. Program listings are terminated with a <Ctrl-C>.

LIST *line number* lists the program line number to the end of the program. LIST *line number-line number* lists the program from the first line number to the second line number.

RELATED

LIST#

BASIC PROGRAMMING GUIDE

LOG

Syntax: LOG (*expr*)

Function: Returns the natural logarithm (base "e") of *expr* which must evaluate to greater than zero. Calculated to seven significant digits.

Mode: Command, run

Use: PRINT LOG(COS(0))

ERRORS

ARITH. UNDERFLOW *expr* or result is less than BASIC-52's smallest floating-point value of $\pm 1\text{E-}127$

ARITH. OVERFLOW *expr* or result is greater than BASIC-52's largest floating point value of $\pm .99999999\text{E+}127$

BAD ARGUMENT Attempt to take LOG() of zero

EXAMPLE

```
100 PRINT EXP(-200), LOG(1.383901E-87)
```

```
>run
```

```
1.383901 E-87 -200
```

BASIC PROGRAMMING GUIDE

MTOP

Syntax: MTOP
 MTOP = *last valid RAM address*

Function: Reads or assigns the top of external data memory which will be used by Basic for variable, string, and RAM program storage

Mode: Command, run

Use: MTOP=30000
 PRINT MTOP

DESCRIPTION

The MTOP system control value is the maximum external data memory address which BASIC-52 will use for RAM program space and variable and string storage. MTOP is not necessarily the top of available external data memory. On cards with 32K of RAM, MTOP is automatically set to 7E00H on power up. On cards with 128K or more of RAM, MTOP is set to 0FFFFH on power up.

RELATED

ST@, LD@

ERROR

MEMORY ALLOCATION MTOP has been assigned a value greater than top of external data memory.

EXAMPLE

```
? MTOP
65535
```

BASIC PROGRAMMING GUIDE

NEW

Syntax: NEW

Function Erases current program in RAM. All variables and strings are cleared.

Mode: Command

Use: NEW

DESCRIPTION

The NEW command deletes the program currently in RAM, sets all variables equal to zero, and clears all strings and multi-tasking interrupts. NEW does not effect the real-time clock or string allocation.

RELATED

CLEAR

BASIC PROGRAMMING GUIDE

NULL

Syntax: NULL *integer*
Where: *integer* = 0 -255
Function: Sets number of NULL characters output to user after a carriage return
Mode: Command
Use: NULL 100

DESCRIPTION

The NULL command controls how many NULL characters (00H) are output following a carriage return. After a reset, NULL = 0. Because this is a command mode command, it cannot be used as part of a program. The NULL count is stored at external data memory location 15H. Change the value of NULL in a program using the DBY(21)=*expr* instruction, where *expr* is any value between 0 and 255. No error is returned if it is greater than 255.

NULL is generally needed only if you have a slow printer connected to the serial printer port. Note that NULL affects all serial ports.

Some terminal programs will advance the cursor when a null character is received. This may result in an strange looking display.

RELATED

LIST, PRINT

ERROR

BAD SYNTAX When *integer* is negative.

BASIC PROGRAMMING GUIDE

ONERR

Syntax: ONERR *line number*

Function: Goes to *line number* on arithmetic error, bad argument, and hardware errors.

Mode: Run

Use: ONERR 1000

DESCRIPTION

The ONERR instruction traps arithmetic errors and hardware problems, transferring control to *line number*. ONERR can be used to handle errors generated due to bad user input from an INPUT instruction. ONERR is a GOTO, not a GOSUB. Consequently, there is no easy way to resume program execution. The control and argument stacks are cleared so all GOSUB's, FOR-NEXT loops, etc. are cleared.

Error codes are stored at external memory location 257 (101H) and are accessed using the XBY instruction.

Code	Error
0AH (10)	DIVIDE BY ZERO
14H (20)	ARITH OVERFLOW
1EH (30)	ARITH UNDERFLOW
28H (40)	BAD ARGUMENT
32H (50)	HARDWARE

EXAMPLE

```
100 ONERR 1000
110 A=1/0
1000 PRINT "Error code:",XBY(257)
```

```
>run
Error code: 10
```

BASIC PROGRAMMING GUIDE

ON GOSUB

Syntax: ON *expr* GOSUB *line0*[,*line1*[,*line2*...]]
 Where: *expr* = 0 to number of subroutines after GOSUB
 line_n = subroutine line number to execute

Function: Calls subroutine based on value of *expr*.

Mode: Run

Use: ON A GOSUB 100, 200, 500

DESCRIPTION

The ON-GOSUB instruction conditionally branches to one of several possible subroutines depending on the value of *expr*. *expr* must evaluate to greater than or equal to zero. If *expr* evaluates to zero, execution branches to *line0*. When *expr* evaluates to one, execution branches to *line1*, etc. If necessary, *expr* is truncated to an integer.

ON-GOSUB saves the location of the program statement after ON-GOSUB on the control stack and immediately transfers program control to the selected subroutine. The subroutine is then executed. When Basic encounters the RETURN instruction, program execution resumes at the program statement after ON-GOSUB. ON-GOSUB instructions can be nested.

One or more of *line_n* may be the same, to execute the same subroutine with different *expr* values. At least one *line_n* must be provided. *line_n* can be in any order.

RELATED

ON GOTO, GOSUB, RETURN

ERRORS

BAD ARGUMENT The value of *expr* is less than 0

BAD SYNTAX The *expr* value is larger than the number of subroutine locations provided, or commas were omitted between {subr n line#} values, or no subroutine locations were given.

C-STACK Attempted recursion caused control stack overflow

EXAMPLE

```
10    P=2
20    ON P GOSUB 1000, 3000, 2000
30    END
1000 PRINT "Line 1000"
1010 RETURN
2000 PRINT "Line 2000"
2010 RETURN
3000 PRINT "Line 3000:"
3010 RETURN

>run
Line 3000
```

BASIC PROGRAMMING GUIDE

ON GOTO

Syntax: ON *expr* GOTO *line0*[,*line1*[*line2*...]]

Function: Branches to a program line based on *expr* value.

valuate to greater than or equal to zero; if *expr* evaluates to zero, execution branches to {0th line#}; if *expr* evaluates to one, execution branches to {1st line#}, etc. Commas shown are required.

Mode: Run

Use: ON A/5 GOTO 100, 200, 500

DESCRIPTION

The ON-GOTO instruction conditionally branches to *linen* where 'n' is the value of *expr*. The *expr* must evaluate to greater than or equal to zero. When *expr* evaluates to zero, execution branches to *line0*. When *expr* evaluates to one, execution branches to *line1*, etc. If necessary, *expr* is truncated to an integer.

One or more of the program lines may be the same, to GOTO the same location with different *expr* values. At least one program line must be provided. Program lines may occur in any order, for example, ON A GOTO 500,700,600.

RELATED

GOTO, GOSUB, ON-GOSUB

ERRORS

BAD ARGUMENT The value of *expr* is less than 0.

BAD SYNTAX The *expr* value is greater than the number of {"nth" line#} numbers provided, or commas were omitted between {line#} values, or no line numbers were provided after the ON-GOTO.

EXAMPLE

```
10 P=2
20 ON P GOTO 1000,2000,3000
30 END
1000 PRINT "Line 1000"
1010 END
2000 PRINT "Line 2000"
2010 END
3000 PRINT "Line 3000"
3010 END

>run
Line 3000
```

BASIC PROGRAMMING GUIDE

ONTICK

Syntax: ONTICK *time,line number*
 Where: *time* = time interval from 0.01 to 327
 line number = line to branch

Function: Calls subroutine at *line number* every *time* interval.

Mode: Run

Use: ONTICK 1.25,500

DESCRIPTION

ONTICK calls a subroutine every *time* interval. *time* ranges from 0.010 seconds to 327.7 seconds (approximately 5.5 minutes). *time* can be specified in increments as small as 0.005 seconds. ONTICK interrupts are turned off when *time* = 0. A line number must still be provided even though it is not used.

The interval period can be reset at any time in a program. When an ONTICK statement is executed, an interrupt will occur in *time* seconds. Time accumulated since the last interrupt is discarded.

NOTE: Use the RETI command to exit this subroutine. Failure to do so prevents future ONTICK interrupts.

Make sure your ONTICK subroutine can finish before the next interrupt. If the program is in the subroutine longer than the specified time interval, the next one will be missed.

This interrupt has the highest priority of any others. ONITR can interrupt any other routine, but no other interrupt can take over this one.

RELATED

RETI

ERRORS

BAD ARGUMENT When *time* > 327.6 or negative
BAD SYNTAX When any parameters left out
INVALID LINE When *line number* not found

EXAMPLE

The following example will interrupt 5 times before it is canceled at line 220.

```
10 A = .15
20 ONTICK A,200
30 IF C<4 THEN A=A+1 : GOTO 30
40 END
200 PRINT A
210 C = C + 1
220 IF C = 5 THEN ONTICK 0,200
230 RETI

>run
145.15
286.15
431.15
575.15
```

The IDLE command may be used to "wait" for an ONTICK interval interrupt.

BASIC PROGRAMMING GUIDE

PI

Syntax: PI
Function: Stored constant 3.1415926
Mode: Command, run
Use: PRINT PI

DESCRIPTION

PI is closer to 3.141592653, so proper rounding should be 3.1415927. However, trig errors were greater when 7 was used than 6 for the last digit.

BASIC PROGRAMMING GUIDE

POP

Syntax: POP *variable* [,*variable*,...]
Function: Takes a value PUSHed to a stack and assigns it to the variable.
Mode: Command, run
Use: POP X,Y,Z

DESCRIPTION

Multiple variables can be POPped off the stack by separating the variables with commas. The first value POPped is the last value PUSHed.

POP and PUSH are useful for transferring data values between subroutines. They allow you to write a subroutine with arbitrary variable names. Data transfers to and from the subroutine can be performed by PUSH and POP, rather than by equating variable names.

RELATED

PUSH, LD@, ST@

ERROR

A-STACK No *variable* on the stack when the POP instruction executed.

EXAMPLE

```
100 FOR N=2 TO 7
110 PUSH REGREAD(n)
120 NEXT
130 FOR N=2 TO 7
140 POP A
150 PRINT A*.00214
160 NEXT
```

>run

```
0
0
0
0
0
.536
3.445
2.334
```

BASIC PROGRAMMING GUIDE

PH0.

PH1.

Syntax: PH0. *expr*

PH1. *expr*

Where: *expr* = any mathematical expression

Function: Print in hexadecimal format following the number with an "H".

Mode: Command, run

Use: PH0. PEEKB(1,3000)

DESCRIPTION

The PH0. and PH1. instructions work like PRINT instruction except that it print values in HEX. The value printed is always a truncated integer and is followed with an "H" to indicate hexadecimal format. If *expr* evaluates to a fractional number within integer range, *expr* is truncated and displayed in hex format. If *expr* is not within integer range (0 through 0FFFFH/65535), the normal decimal PRINT mode is used. PH0. suppresses two leading zeros if *expr* evaluates to less than 0FFH. PH1. always prints four hexadecimal digits.

If there is no *expr*, a carriage return - line feed combination (a blank line) will be output. An *expr* may be any combination of instructions/operators and variables, strings, or literal values. More than one *expr* may be output by separating them with commas. Values are printed with a leading space; a list of values separated by commas will thus print with one intervening. This is different from the decimal PRINT instruction which prints values with a trailing blank. Strings and literals are output with no added blanks. If a comma is the last character in the list then a carriage return/linefeed is suppressed.

EXAMPLE

```
100 PH0. A
```

BASIC PROGRAMMING GUIDE

PRINT

PRINT #,

P.

?

Syntax: PRINT *expr*

P. *expr*

? *expr*

PRINT#*port,expr*

P.#*port,expr*

?#*port,expr*

Where: *expr* = any string, mathematical number, or calculation

port = serial output port 0 or 1. Your card may have more ports.

Function: Prints value of *expr* to a serial port

Mode: Command, run

Use: PRINT "String",\$(0),REGREAD(9)*.007214

DESCRIPTION

PRINT is used to send serial data to any port. Default is COM 0. PRINT #*n* may be used to write data out of a modbus port.

If there is no *expr*, a carriage return - line feed combination is sent. *expr* is any combination of instructions/operators and variables, strings, or literal values. More than one *expr* may be output by separating them with commas. Values are printed with a leading and trailing space; a list of positive values separated by commas will thus print with two intervening blanks. A "+" is implied. The "-" symbol precedes negative values and takes the place of the normal preceding space. Strings and literals are output with no added blanks. If a comma is the last character in the list then the normal <CR><LF> is suppressed.

The shorthand versions P. and ? are converted to PRINT after each program line is entered, so a P. or ? is never listed.

The PRINT#*port*, instruction functions exactly like the PRINT instruction, but it directs output to the designated serial port. When using this syntax, any output directed by the UO command is bypassed.

P.# and ?# are shorthand for PRINT#.

When PRINT #*n* is used for a modbus port, a required number of parameters in a specific sequence is required. Modbus transaction types supported are 3 and 16. These are query and write, respectively. Generally, a modbus ID and register number are required,

A modbus query (read from a slave or other device on a network) is as follows:

```
PRINT #n,id,3,address,length
```

Where: *n* = modbus port number on board. This is usually 1 or 3 or both. Refer to your board manual.

id = modbus ID of device you want to get data from

address is valid modbus address

length is number of registers to return

After a query, you will want to retrieve the data. This is done through a number of registers and is board dependent. Generally registers 4781-4786 are used to retrieve data. Refer to your boards manual for specific information.

A modbus write (write to a slave on a modbus network) is as follows:

BASIC PROGRAMMING GUIDE

`PRINT #n,ID,16,start register,length,data,data,data...`

Where: `n` = modbus port number on board. This is usually 1 or 3 or both. Refer to your board manual.

`Id` = modbus ID of device you want to get data from

`start register` = is start of modbus address you want to write to

`length` = number of registers you will write to

`data` = number information (integer or float, as appropriate to register) you wish to write.

Number of `data` elements must be the same as `length` and must all be included in the same line.

EXAMPLE

```
100  STRING 200,20 : $(0)="String" : B=PI*5
110  PRINT $(0),B,REGREAD(2)*.00215
```

>run

String 15.707963 0

For modbus read (query)

```
rem send out id, function number, address, length
70 print #3,id,fn,ad,le
```

For modbus write

```
rem print syntax is id, function, start register, length, data, ..
```

```
10  PRINT #3,11,16,4501,3,1,2,3
```

BASIC PROGRAMMING GUIDE

PROG

FPROG

Syntax: PROG *expr*

 FPROG 0

 Where: *expr* = programming command

Function: Clears or saves program to CPU flash and enables autorun on power up or reset

Mode: Command

Use: FPROG 0

DESCRIPTION

This group of commands erase and save basic programs to CPU flash EPROM, depending upon the expression following the command. They operate differently than Basic-52 due to the flash in the CPU. Not all original Basic-52 programming features are available.

FPROG0 erases all basic programs in CPU flash.

PROG creates a programming slot and copies the program in RAM to CPU flash. After executing this command, you will have a number and address returned similar to below.

```
>prog
```

```
1
```

```
8013H
```

You should then execute a ROM 1 statement to continue to enter a program or run it.

PROG 0 disables program 1 from autorunning on power up or reset.

PROG 2 enables program 1 to autorun on power up or reset.

RELATED

NEW, ROM, RAM

BASIC PROGRAMMING GUIDE

PUSH

Syntax: **PUSH** *expr1* [*,expr2,...*]

Where: *expr* is a numeric value

Function: Puts the value of *expr* to the argument stack. The first value PUSHed and is the last POPped.

Mode: Command, run

Use: **PUSH** X,Y

DESCRIPTION

PUSH and POP instructions pass values to Basic subroutines. The last value pushed is the last expression in the PUSH instruction, and is also the first popped off the stack. Multiple expressions can be pushed onto the argument stack by separating the expressions with commas.

The PUSH and POP instructions alleviate some of the problems of global variables in Basic. They eliminate the need to equate subroutine variables to global variables used by the program which called the subroutine.

RELATED

POP, LD@, ST@

ERROR

A-STACK Attempt to push too many values on the argument stack. Typically no more than 32 values may be PUSHed onto the stack before it is full.

EXAMPLE

Please refer to the POP example.

BASIC PROGRAMMING GUIDE

RAM

Syntax: RAM

Function: Enter RAM mode, usually from ROM.

Mode: Command

Use: RAM

DESCRIPTION

Command instructs the OS to exit ROM and use RAM for its current program. The program in RAM may then be listed and ran.

RELATED

ROM, FPROG, PROG

BASIC PROGRAMMING GUIDE

READ

Syntax: READ *variable* [,*variable*, ...]

Function: Sequentially assigns the values of data provided in the DATA statement to the variables in a list.

Mode: Run

Use: READ X,Y,Z

DESCRIPTION

Multiple variables following one READ instruction must be separated by commas. READ must always be followed by at least one variable.

See RESTORE for examples and more information.

BASIC PROGRAMMING GUIDE

REGREAD

Syntax: REGREAD(*register*)
Function: Return value from a modbus register
Mode: Command, run
Use: PRINT REGREAD(1)
100 A = REGEAD(n)

DESCRIPTION

REGREAD function returns an integer or float from a modbus register. Variable type conversion is automatically converted to a format compatible with Basic 52 (BCD floating point).

register range can be between 1 and 65535. However, this is board dependent. Consult your board manual for valid registers. Most boards have valid ranges and gaps between ranges and are not contiguous.

Floating point numbers stored in modbus registers are in IEE-754 format. This format is known as single precision. Effectively 7 digits plus exponent can be stored in this format. There are some numbers that, when stored using REGWRITE, will return a slightly different number.

For example, if you stored one of the original numbers below using REGWRITE, REGREAD will return a slightly different number.

Original	REGREAD
80.3	80.30001
80.6	80.59999

For all intents and purposes the differences are not significant, less than 0.0001245%.

ERROR

BAD ARGUMENT - when *register* is not valid or out of range.

EXAMPLE

```
100 REGWRITE 7033,A
120 PRINT REGEAD(7033)

>run
12
```

BASIC PROGRAMMING GUIDE

REGWRITE

Syntax: REGWRITE *address,data*
Function: Write a value to a modbus register
Mode: Command, run
Use: REGWRITE 7033,n

DESCRIPTION

REGWRITE function writes an integer or float to a modbus register. Variable type conversion is automatically converted from a format compatible with Basic 52 (BCD floating point) to the register type.

address range can be between 1 and 65535. However, this is board dependent. Consult your board manual for valid registers. Most boards have valid ranges and gaps between ranges and are not contiguous.

Values for *data* are register dependent. Registers that are integer type accept numbers between 0 - 65535. Float registers accept numbers in the range of +/- E¹²⁷. When you try to store a floating point number to an integer, all numbers to the right of the decimal are truncated. Only the whole number is stored.

Floating point numbers stored in modbus registers are in IEE-754 format. This format is known as single precision. Effectively 7 digits plus exponent can be stored in this format. There are some numbers that, when stored using REGWRITE, will return a slightly different number.

For example, if you stored one of the original numbers below using REGWRITE, REGREAD will return a slightly different number.

Original	REGREAD
80.3	80.30001
80.6	80.59999

For all intents and purposes the differences are not significant, less than 0.0001245%.

ERROR

BAD ARGUMENT - when *address* is not valid or out of range or when integer *data* is out of range.

EXAMPLE

```
100 REGWRITE 7033,A
120 PRINT REGEAD(7033)

>run
12
```

BASIC PROGRAMMING GUIDE

REM

Syntax: REM *any ASCII text*
Function: Allows remarks in a program or on command line
Mode: Command, run
Use: 100 REM You can put any thing you want here
REM This remark has no line number so will be discarded by BASIC-52

DESCRIPTION

The REM instruction lets you add comments to your program. Any text after a REM is ignored. REM instructions cannot be terminated with a colon, but they can follow colons. BASIC-52 lets you use REM in command mode and while downloading programs. A REM without a preceding line number is ignored by BASIC-52. This allows you to place comments in an off-line source code text file and have them stripped out when you download the text file to the card.

Appropriate comments make your programs easier to understand and maintain, but do slow program execution and consume program memory.

Multiple statements per line following a REM are ignored since they are considered part of the remark. Refer to the example.

EXAMPLE

```
100 REM A comment
120 PRINT A :REM PRINT A+2

>run
0
```

BASIC PROGRAMMING GUIDE

RESTORE

Syntax: RESTORE
Function: Resets the READ instruction pointer to the beginning of the DATA list.
Mode: Run
Use: RESTORE

DESCRIPTION

After a RESTORE statement is executed, the next READ statement accesses the first item in the first data statement in the program.

ERROR

NO DATA - no DATA list provided.

EXAMPLE

```
100    READ A,B,C
110    PRINT A,B,C
120    RESTORE
130    READ X,Y,Z
140    PRINT X,Y,Z
150    READ A,B,C
160    PRINT A,B,C
150    DATA 1,2,3*2
150    DATA 6,9,12
```

>run

```
1 2 6
1 2 6
6 9 12
```

BASIC PROGRAMMING GUIDE

RETI

Syntax: RETI

Function: Return from ONITR or ONTICK interrupt. RETI must be the last instruction of the interrupt subroutine.

Mode: Run

Use: RETI

DESCRIPTION

The RETI instruction causes you to exit from ONTICK and ONITR (if available) interrupts. RETI functions like RETURN, but it clears software interrupt flags so that BASIC-52 can acknowledge subsequent interrupts. If you don't execute the RETI instruction in the interrupt procedure, all future interrupts, hardware and software, are ignored.

RELATED

ONITR, ONTICK

EXAMPLE

Refer to ONTICK and ONITR examples.

BASIC PROGRAMMING GUIDE

RETURN

Syntax: RETURN

Function: Returns program to next instruction following a GOSUB command or software interrupt (ON LINE, ON KEYPAD, etc.)

Mode: Run

Use: RETURN

DESCRIPTION

RETURN is used as a return from a GOSUB. Program execution continues at the statement following the GOSUB.

BASIC PROGRAMMING GUIDE

RND

Syntax: RND
Function: Returns a pseudo-random fractional number between zero and one inclusive.
Mode: Command, run
Use: A=RND

DESCRIPTION

The RND operator uses a 16-bit binary seed and repeats after 65535 pseudo-random numbers. The initial seed is the value of MTOP. The seed can be changed by writing to address 10CH and 10DH using the XBY command.

EXAMPLE

```
100 A=RND
110 PRINT A
```

BASIC PROGRAMMING GUIDE

ROM

Syntax: ROM *n*
Where: *n* = rom number slot to load
Function: Enters ROM stored in CPU flash memory.
Mode: Command
Use: ROM

DESCRIPTION

Command instructs the OS to enter numbered ROM and use it for its current program. The program in ROM may be listed and ran.

If there is only one program in ROM, this program may be edited as if it was RAM. However, when editing programs, only the highest numbered ROM can be changed.

If *n* is not supplied, 1 is assumed.

ROM numbers, or slots, were created using PROG.

RELATED

RROM, FPROG, PROG

BASIC PROGRAMMING GUIDE

RROM

Syntax: RROM *n*
Where: *n* = rom number slot to load
Function: Runs specified ROM stored in CPU flash memory.
Mode: Command, run
Use: RROM *n*

DESCRIPTION

Command instructs the OS to enter numbered ROM and begin to execute the program. Variable *n* is a number equal to or less than the number of Basic programs. If *n* is not supplied, 1 is assumed.

This command is the way one Basic program calls another. In command mode, this is the same as entering ROM *n*, RUN.

Executing this command clears all interrupts and sets all basic variables and strings to zero. Note that register variables remain intact.

ROM numbers, or slots, were created using PROG.

RELATED

RROM, FPROG, PROG

BASIC PROGRAMMING GUIDE

SGN

Syntax: SGN(*expr*)

Function: Returns +1 if *expr* is greater than zero, zero if the *expr* equals zero, and -1 if *expr* is less than zero.

Mode: Command, run

Use: PRINT SGN(SIN(X))

DESCRIPTION

Use SGN in level control applications. If a level is high or low, it can direct control to the appropriate program.

EXAMPLE

```
100 ON SGN(A)+1 GOSUB 2000,3000,4000
```

BASIC PROGRAMMING GUIDE

SIN

Syntax: `SIN(expr)`

Function: Returns the trigonometric SINE of *expr* which is assumed to be in radians. The value of *expr* must be in the range of +/- 200,000.

Mode: Command, run

Use: `PRINT SIN(PI/2)`

DESCRIPTION

SIN, COS, and TAN operators use a Taylor series to calculate the function. These operators first reduce the argument to a value that is between 0 and $\pi/2$. the algorithm used to reduce the value will reduce accuracy when *value* is large. To maintain accuracy, keep the arguments for trig functions as small as possible.

ERRORS

ARITH. UNDERFLOW *value* or result is less than BASIC-52's smallest floating-point value of $\pm 1E-127$

ARITH. OVERFLOW *value* or result is greater than BASIC-52's largest floating-point value of $\pm .9999999E+127$

DIVIDE BY ZERO Attempt to take TAN(X) when $\text{COS}(\pi/2) = 0$

EXAMPLES

```
10 PRINT SIN(PI/2),COS(10*PI),TAN(8*PI/4)
20 PRINT ATN(PI)
```

>run

```
1 1 0
1.2626272
```

BASIC PROGRAMMING GUIDE

SPC

Syntax: PRINT SPC(*expr*)

Where: *expr* = number of spaces to print

Function: Sends *expr* number of space characters (20H) from the serial port.

Mode: Command, run

Use: PRINT SPC(A*4),

DESCRIPTION

SPC must be used in conjunction with a print statement.

EXAMPLE

```
100 PRINT SPC(80-A*4),
```

BASIC PROGRAMMING GUIDE

STOP

Syntax: STOP
Function: Breaks program execution; resume with the CONT command.
Mode: Run
Use: STOP

DESCRIPTION

The STOP instruction lets you break program execution at specific locations in a program. You can display and modify variables after STOPping a program. STOP is useful for program debugging. The CONT command lets you resume program execution.

The line number printed after execution of a STOP instruction is the line number following the instruction and not the line number containing the STOP instruction.

If you modify a STOPped program, CONT will be unable to continue execution.

RELATED

CONT, GOTO

ERROR

CAN'T CONTINUE Attempt to continue after editing a stopped program, or attempt to execute CONT without a prior STOP or <Ctrl-C>.

EXAMPLE

```
100 PRINT "Tick=",TICK(0)
110 STOP
110 GOTO 100

>run

A= 0
STOP - IN LINE 120
```

BASIC PROGRAMMING GUIDE

STR

Syntax: A = STR(*function*,\$(*n*)[,\$(*n*)])

Where: *function* = 0 to 14, specifies string function to perform as described below.

Function: Performs string manipulation, described below, per function number.

Mode: Command,Run

Use: A = STR(0,\$(0))

Cards: RPC-320, RPC-330

DESCRIPTION

There are 11 string manipulation functions using STR. Each function is described below.

NOTE: Most of these functions require a string variable (such as \$(0)) rather than a quoted string. Functions which will allow quoted strings offer an alternate syntax immediately below the first one.

Syntax: A = STR(0,\$(n))

Description:

Returns number of characters in a string. When string is not set equal to something, or the string number is out of bounds, erroneous data is returned. Length limit is 254 characters.

Example:

```
10 STRING 100,20
20 $(0)=" 123456789"
30 PRINT STR(0,$(0))
run
10
```

Syntax: A = STR(1,\$(n))

Description:

Convert letters A - Z to lower case. Variable A returns length of the string.

Example:

```
10 STRING 100,20
20 $(0)="Some UPPER case"
30 A = STR(1,$(0))
40 PRINT $(0)
run
some upper case
```

Syntax: A = STR(2,\$(n))

Description:

Convert letters a - z to upper case. Variable A returns length of the string.

Example:

```
10 STRING 100,20
20 $(1) = "Some lower case."
30 A = STR(2,$(1))
40 PRINT $(1)
run
SOME LOWER CASE.
```

BASIC PROGRAMMING GUIDE

Syntax: A = STR(3,\$(n))

Description:

Returns numbers in a string as a real number. Similar to VAL in other Basics. Leading spaces are ignored. First non-number terminates conversion at last valid number. No valid numbers return 0. Number length is limited to the first 12 valid numbers and decimal in a string. This means a number no greater than 999999999999 is converted to a number.

```
Example: 10 STRING 100,20
          20 $(2) = "-23.452volts"
          30 A= STR(3,$(2))
          40 PRINT A
          run
          -23.452
```

Syntax: A = STR(4,\$(n))

Description:

Trims spaces to left of first non-space character. Variable A returns length of trimmed string.

```
Example: 10 STRING 100,20
          20 $(0) = " 1234"
          30 A = STR(4,$(0))
          40 PRINT $(0)
          50 PRINT A
          run
          1234
          4
```

Syntax: A = STR(5,\$(n))

Description:

Trims spaces from right side of string. Variable A returns length of trimmed string.

```
Example: 10 STRING 100,20
          20 $(0) = "ABCDE "
          30 A = STR(5,$(0))
          40 PRINT $(0)
          50 PRINT A
          run
          ABCDE
          5
```

Syntax: A = STR(6,\$(x),\$(y))
A = STR(6,\$(x),"string")

Description:

Appends one string into another. This function concatenates two strings in the form of \$(x) = \$(x) + \$(y). Length of new string is returned in variable A. The variable \$(y) could be a quoted *string*.

```
Example: 10 STRING 120,40
          20 $(0)="First part"
          30 $(1)=" Second part"
          40 A = STR(6,$(0),$(1))
          50 PRINT $(0)
          60 PRINT "Length:",A
          70 A = STR(6,$(0)," last part")
          80 PRINT $(0)
          90 PRINT "Length:",A
          run
          First part Second part
          Length: 22
          First part Second part last part
          Length: 32
```

Lines 50 and 80 print the concatenated string \$(0).

BASIC PROGRAMMING GUIDE

Syntax: `A = STR(7,$(put),$(get),position,length)`

Description:

Extracts a portion of a string from $$(get)$ and transfers it over to $$(put)$. The actual number of characters moved is returned. $position$ starts at 1. When $position$ is 0, no characters are placed into $$(put)$ regardless of $length$. When $length$ is 0, all characters are copied from $$(get)$ to $$(put)$ starting at $position$.

```
Example: 10 STRING 200,20
          20 $(0) = "123456.789"
          30 A = STR(7,$(1),$(0),3,5)
          40 PRINT $(1)
          50 PRINT "Length:",A
          run
          3456.
          Length: 5
```

Syntax: `A = STR(8,$(search),$(substring))`

Description:

Scans $$(search)$ for occurrence of $substring$. Returns position where entire $substring$ first matches $search$ string. Returns 0 when $substring$ is not in $search$ string.

```
Example: 10 STRING 200,20
          20 $(0) = ">05M34C3"
          30 $(1) = "05M"
          40 A = STR(8,$(0),$(1))
          50 PRINT "Position match at:",a
          run
          Position match at: 2
```

The number '0' in $$(1)$ matches $$(0)$ at position 2.

Syntax: `A = STR(9,$(string1),$(string2))`

Description:

Compares $string1$ to $string2$. Returns position of first mismatch. If both strings exactly match, then 0 is returned.

```
Example: 10 STRING 200,20
          20 $(0) = ">05M34C3"
          30 $(1) = ">05"
          40 A = STR(9,$(0),$(1))
          50 PRINT "Mismatch starting at:",a
          run
          Mismatch starting at: 4
```

Since the first three character positions matched, position 4 is returned as the longer string did not match the shorter one.

String functions 8 and 9 are useful in RS-485 network communication. In the above example, ">05" could be the RPC-320's address. Knowing the mismatch starts at position 4, the address can be assumed correct. If the mismatch started sooner, a smaller number is returned. Hence, the message was not intended for this particular card and the entire message can be flushed.

BASIC PROGRAMMING GUIDE

Syntax: `A = STR(10,$(n),format,variable)`

Description:

Converts and formats *variable* into a string and puts it into $\$(n)$. Variable A returns irrelevant data. Formatting is controlled by the *format* variable. Strings are formatted into one of 3 basic patterns, described below.

format = 0. Default free format. When number is between ± 99999999 and ± 0.1 , RPBASIC will save integers and fractions. When numbers are outside this range, the F0 format, described next, is used.

format = Fx. Floating point format. 'x' determines how many digits after the decimal point are saved. When $x = 0$, the number of trailing digits will vary and trailing 0's are not saved. 'x' is represented as a hex number. When *format* = 0F3H, 3 decimal numbers are printed. An alternate way of setting floating point output is to make *format* = the number of decimal numbers plus 240.

format = xyH. Force integer and/or fraction output. Command is same as USING(##.##), where 'x' is the number digits left of the decimal point and y is to the right. Maximum value for x and y is 7. Use the hex format to set the number.

Example:

```
10 String 200,20
20 C = 23.45
30 F = 0
40 A = STR(10,$(0),F,C)
50 PRINT "Variable value, before formatting:",C
60 PRINT "String in free format:",$(0)
70 F = 0F2H
80 A = STR(10,$(0),F,C)
90 PRINT "Using floating point format:",$(0)
100 F=52H
110 A=STR(10,$(0),F,C)
120 PRINT "Using #####.## format:",$(0)
run
Variable value, before formatting: 23.45
String in free format: 23.45
Using floating point format: 2.34 E+1
Using #####.## format: 23.45
```

ERROR

BAD ARGUMENT When *function* is out of range or string data is incorrect.

BASIC PROGRAMMING GUIDE

STRING

Syntax: STRING *total bytes, string length*
 Where: *total bytes* = total number of bytes in memory to allocate
 string length = maximum number of bytes in a string

Function: Allocate memory for strings

Mode: Command, run

Use: STRING 56,10 : REM Allocate memory for 5 10-byte strings

DESCRIPTION

Prior to using strings, you must use STRING to allocate memory for them. The STRING argument values are computed by this equation:

$$total\ bytes = ((string\ length + 1) * number_of_strings) + 1$$

The only way to recover string memory is with a "STRING 0,0" instruction. String memory is reclaimed and then reallocated each time you use the STRING operator. Strings are terminated with a carriage return (ODH or 13) which is the additional byte added to your *bytes per string expr*.

WARNING:

STRING causes BASIC-52 to execute the equivalent of a CLEAR instruction since string and numeric variables occupy the same memory space. In other words, the STRING instruction clears all variables, interrupts and stacks. Allocate string memory early in your program and don't reallocate it unless you can accept the loss of all variables.

RELATED

ASC, CHR

ERRORS

MEMORY ALLOCATION Memory not allocated for strings

C-STACK STRING used in a subroutine, clearing the stack.

EXAMPLES

```
10     STRING 1000,40
20     $(0) = "Up to 40 characters in this string"
       .
       .
100    $(2) = COM$(1)
```

BASIC PROGRAMMING GUIDE

SQR

Syntax: SQR(*expr*)
 Where: *expr* is any valid mathematical expression, number, or variable greater than 0

Function: Returns a positive square root.

Mode: Command, run

Use: PRINT SQR(A)

DESCRIPTION

expr must be positive. Any calculation is accurate to ± 5 least significant digits.

ERRORS

ARITH. UNDERFLOW *expr* or result is less than BASIC-52's smallest floating point value of $\pm 1\text{E-}127$

ARITH. OVERFLOW *expr* or result is greater than BASIC-52's largest floating point value of $\pm .99999999\text{E}+127$

BAD ARGUMENT Attempt to take SQR() of a negative number

EXAMPLE

```
100  FOR N = 1 TO 10
110  A=SQR(N)**2
120  IF (A-N)<>0 THEN PRINT A,N
130  NEXT
```

```
>run
2.0000001  2
2.9999999  3
5.0000001  5
6.0000002  6
6.9999999  7
7.9999998  8
```

BASIC PROGRAMMING GUIDE

ST@

Syntax: ST@ *expr*
Where: *expr* = 0 to 65535
Function: Takes a floating-point number from the argument stack and stores it to data memory at the address.
Mode: Command, run
Use: PUSH B : ST@7E00

DESCRIPTION

ST@ is used in conjunction with PUSH, POP, and LD@. Use these commands to save and retrieve floating point numbers to program RAM.

NOTE: LD@ and ST@ cannot use extended RAM. Only segment 0 RAM (used for running Basic programs) is used. Use PEEK and POKE commands to access this memory.

WARNING: When 128K and 512K RAM are installed, all of memory is cleared on power up and reset. Do not use LD@ or ST@ to save floating point numbers in segment 0. Use POKE and PEEK type commands instead.

32K RAM systems have address 7E00H set as MTOP. This location up to 7FFFH may be used to store variables.

expr is the address in RAM of where a number is stored.

Each floating-point number requires six bytes of memory. *expr* in the ST@ and LD@ instructions specify the high address. A number is stored at locations *expr* through *expr-6*.

RELATED

LD@, PUSH, POP

ERROR

expr location should be above MTOP. Otherwise the data may be overwritten.

EXAMPLE

```
100  A=REGREAD(9)*.007234
110  PUSH A
120  ST@7F00H
.
.
300  LD@7F00H
310  POP B
320  PRINT "Analog value retrieved=",B

>run

Analog value retrieved=",B
```

BASIC PROGRAMMING GUIDE

TAB

Syntax: PRINT TAB(*position*)
Where: *position* = 1 to 255
Function: Specifies a column number at to begin printing.
Mode: Command, run
Use: PRINT TAB(5), "Pressure", TAB (20),"Temperature"

DESCRIPTION

TAB is used with PRINT. It is used to print data in table form. If the cursor is past the requested column, the instruction is ignored.

ERROR

BAD ARGUMENT When *position* is negative or out of range.

EXAMPLE

```
100 PRINT TAB(5), "Pressure", TAB(20), "Temperature"
110 FOR N=0 TO 6
120 PRINT TAB(7), REGREAD(2)*.237,
130 PRINT TAB(23), REGREAD(3)*1.324
140 NEXT
```

```
>run
Pressure           Temperature
116.13             237.3
116.14             237.3
116.13             237.4
116.14             237.4
116.11             237.0
116.16             237.6
116.13             237.5
```

BASIC PROGRAMMING GUIDE

TAN

Syntax: TAN(*expr*)

Function: Returns the trigonometric tangent (sin/cos) of *expr* which is assumed to be in radians. *expr* must be in the range of +/- 200,000.

Mode: Command, run

Use: PRINT TAN(PI/4)

DESCRIPTION

SIN, COS, and TAN operators use a Taylor series to calculate the function. These operators first reduce the argument to a value that is between 0 and $\pi/2$. the algorithm used to reduce the value will reduce accuracy when *value* is large. To maintain accuracy, keep the arguments for trig functions as small as possible.

ERRORS

ARITH. UNDERFLOW *value* or result is less than BASIC-52's smallest floating-point value of $\pm 1E-127$

ARITH. OVERFLOW *value* or result is greater than BASIC-52's largest floating-point value of $\pm .9999999E+127$

DIVIDE BY ZERO Attempt to take TAN(X) when $\text{COS}(\pi/2) = 0$

EXAMPLES

```
100 PRINT SIN(PI/2),COS(10001*PI),TAN(5*PI/4)
110 PRINT ATN(TAN(PI/4))/PI
```

>run

```
1 -1 1
.24999996
```

BASIC PROGRAMMING GUIDE

TICK

Syntax: **TICK**(*timer*)

Where: *timer* = 0 to 3. It specifies the timer number.

Function: Returns a time from one of 4 process clocks in 5 ms increments.

Mode: Command,Run

Use: A = TICK(2)

DESCRIPTION

There are four tick timers updated 200 times per second. Each timer is independent of each other in that they may be read and cleared separately (see CLEAR TICK). All timers are updated at the same time.

This function is separate from the real time clock and is not battery backed. All timers reset to 0 on power up or reset. Timers continue to run in command mode and cannot be turned off.

TICK(n) returns time in thousandths of a second (in 5 ms intervals) up to 65535.995 seconds, or approximately 18.2 hours. The timer then starts at 0 again.

Tick timers are not affected by to the ONTICK instruction.

RELATED

CLEAR TICK, ON TICK

ERRORS

BAD SYNTAX If any parameters left out

BAD ARGUMENT When *timer* > 3 or negative or left out

EXAMPLE

The following example clears tick timer number 3, delays for a time, then prints tick timers 0 and 3.

```
10 CLEAR TICK(3)
20 FOR X = 0 TO 1000
30 NEXT
40 PRINT TICK(0),TICK(3)

124.6   .425
```

BASIC PROGRAMMING GUIDE

USING

U.

Syntax: PRINT USING (*format*)

PRINT U.(*format*)

Where: *format*

USING(Fn) n is the number of significant digits. A minimum of 3 significant digits are always printed. Maximum value of n is 8.

USING(##) The number of # symbols determines how many significant figures of the output value will be displayed before and after the decimal point. The maximum total number of "#" symbols is 8. Integers (decimals truncated) are printed when there are no "#" symbols after the decimal point or if no decimal point is given. If a value cannot be printed in the requested format, BASIC-52 outputs a "?" and prints the value in USING(0) format.

USING(0) The default output format for BASIC-52 floating-point values. Displayed as a decimal integer and fraction if the value is between +/- 99999999 and +/- 0.1.

Function: Used with PRINT to format subsequent expressions.

Mode: Command, run

Use: PRINT USING(F3),A

DESCRIPTION

Formatting is "remembered" until it is reset or changed.

RELATED STR

ERRORS BAD SYNTAX - Missing # to the left of the decimal point or a space between USING and the left parentheses.

EXAMPLE

```
110 PRINT USING(F3),PI*100
```

```
>run
```

```
3.14 E+2
```

BASIC PROGRAMMING GUIDE

XBY

Syntax: XBY(*addr*)
 XBY(*addr*)=*expr*
 Where: *addr* = 0 to 65535 (0FFFFH) is a memory address
 expr = 0 to 255 is data to save

Function: Read/write external data memory, segment 0 only.

Mode: Command, run

Use: XBY(99)=35

DESCRIPTION

XBY retrieves or assigns a value to external data memory. This command is equivalent to PEEKB and POKEB.

RELATED

CBY, DBY, PEEKB, POKEB

ERROR

BAD ARGUMENT Invalid *addr* or attempt to assign an out of range value to a XBY(*expr*).

EXAMPLE

```
100    XBY(47536) = XBY(47536) .OR. 3
```

BASIC PROGRAMMING GUIDE

APPENDIX A- ERROR MESSAGES

ThePBASIC-52 error processor helps identify errors.

When running a program, error messages are expressed as:

```
ERROR: XXX - IN LINE NNN
```

```
NNN Instruction  
_____X
```

where XXX is the type of error and NNN is the program line number where the error occurred. The "_____X" identifies the very approximate location of the error. For example, a BAD ARGUMENT error occurring at line 100 is expressed as:

```
ERROR: BAD ARGUMENT - IN LINE 100  
100   DBY(257)=5  
_____X
```

In Command mode, only the error type is printed since there are no line numbers in Command mode.

BASIC-52 errors include:

- A-STACK
- ARITH. UNDERFLOW
- ARITH. OVERFLOW
- ARRAY SIZE
- BAD ARGUMENT
- BAD SYNTAX
- C-STACK
- CAN'T CONTINUE
- DIVIDE BY ZERO
- I-STACK
- MEMORY ALLOCATION
- NO DATA
- HARDWARE

A-STACK

The argument stack pointer is out of bounds. Too many expressions were pushed or tried to pop non-existent data off the stack.

ARITH. UNDERFLOW

The result of an arithmetic operation is beyond the lower limit of BASIC-52 floating-point numbers. BASIC-52's smallest floating-point number is $\pm 1E-127$. An operation such as $1E-100/1E28$ would cause an ARITH. UNDERFLOW error.

This example produces a correct result:

```
>?1e-100/1e26  
1.0 E-126
```

BASIC PROGRAMMING GUIDE

This example produces an expected error:

```
?1e-100/1e28
ERROR: ARITH. UNDERFLOW
READY
```

This example produces an incorrect exponent:

```
>?1e-100/.9e28
1.1111111 E-0
```

ARITH. OVERFLOW

The result of an arithmetic operation exceeds the upper limit of BASIC-52 floating-point numbers. BASIC-52's largest floating-point number is $\pm .99999999E+127$. An operation such as $1E100*1E28$ causes an ARITH. OVERFLOW error.

ARRAY SIZE

An array was accessed that is outside the dimension boundaries defined by a DIM instruction. For example:

```
DIM A(100)
PRINT A(102)

ERROR:  ARRAY SIZE
READY
```

BAD ARGUMENT

The argument of an operator is out of limits. For example, $A=REGREAD(300)$ generates a BAD ARGUMENT error since the value assigned by the REGREAD operator is limited.

BAD SYNTAX

An invalid command, instruction, or operator or have attempted to use a reserved key word as part of a variable was entered. This is a generic "I don't know what this is" response by a computer.

C-STACK

More control stack memory was used than it has available. The control stack has of 158 byte of memory. A FOR-NEXT loop uses 17 bytes, and DO-UNTIL, DO-WHILE, and GOSUB each use three bytes. This means you limited to nine FOR-NEXT loops. Executing a return before a GOSUB, or a WHILE or UNTIL before a DO instruction, or a NEXT before a FOR also generates a C-STACK error.

CAN'T CONTINUE

A program was edited after stopping.

DIVIDE BY ZERO

A number was divided by zero or a statement such as $TAN(PI/2)$.

BASIC PROGRAMMING GUIDE

I-STACK

There is not enough internal stack space to evaluate an expression. Usually this is caused by an excessive number of parentheses.

MEMORY ALLOCATION

Accessing a string that is outside the defined string limits or assign MTOP a value that does not contain any RAM.

NO DATA

A READ instruction does not have valid associated DATA instruction. NO DATA - IN LINE XXX error message displays a line number where it expected to find the data.