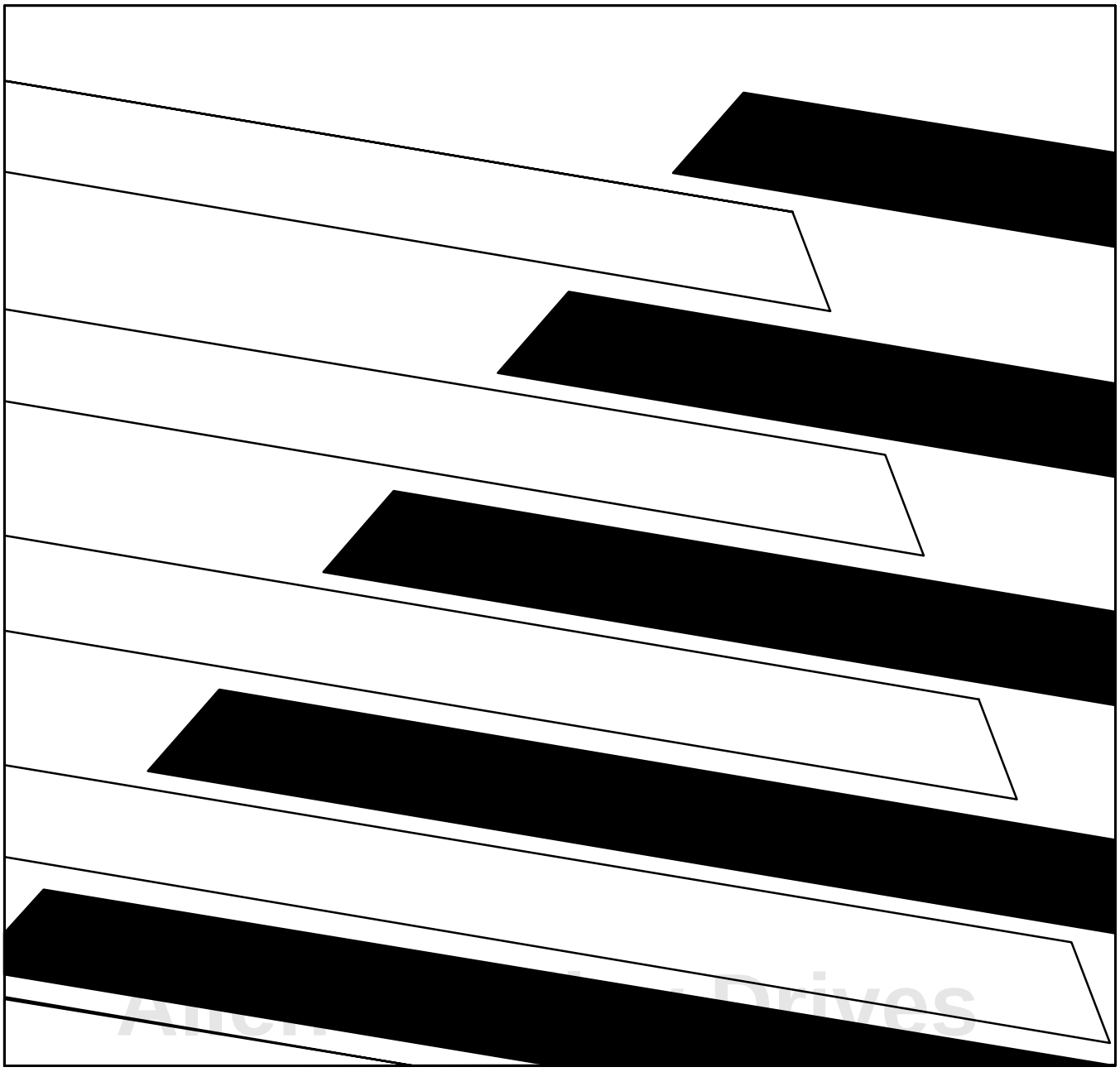
ALLEN-BRADLEY

BASIC Language Reference

(Catalog Number 1746-BAS)

Reference Manual



Important User Information

Solid state equipment has operational characteristics differing from those of electromechanical equipment. "Application Considerations for Solid State Controls (Publication SGI-1.1) describes some important differences between solid state equipment and hard-wired electromechanical devices. Because of this difference, and also because of the wide variety of uses for solid state equipment, all persons responsible for applying this equipment must satisfy themselves that each intended application of this equipment is acceptable.

In no event will the Allen-Bradley Company be responsible or liable for indirect or consequential damages resulting from the use or application of this equipment.

The examples and diagrams in this manual are included solely for illustrative purposes. Because of the many variables and requirements associated with any particular installation, the Allen-Bradley Company cannot assume responsibility or liability for actual use based on the examples and diagrams.

No patent liability is assumed by Allen-Bradley Company with respect to use of information, circuits, equipment or software described in this text.

Table of Contents

Important User Information I

Preface P-1

Manual Overview P-1
WARNINGS, CAUTIONS and Important Information P-2
Conventions P-3
Reference Page Format P-4
{BASIC Command, Statement, or Operator} P-4
BASIC Module Documentation Set P-5

Language Elements 1-1

Character Set 1-1
The BASIC Program Line 1-1
BASIC Line Numbers 1-1
BASIC Statements, Commands, and Operators 1-2
BASIC Line Length 1-2

Data Types 2-1

Data Types 2-1
Argument Stack 2-1
String Data Types 2-1
Numeric Data Types 2-2
Integer Numbers 2-3
Floating-Point Numbers 2-3
Backplane Conversion Data 2-4
Variables 2-4
Variable Names 2-5
Variable Types 2-5

Expressions and Operators 3-1

Expressions and Operators 3-1
Expressions 3-1
Operators 3-1
Hierarchy of Operators 3-2
Arithmetic Operators 3-3
Add (+) 3-3
Divide (/) 3-3
Exponentiation (**) 3-3
Multiply (*) 3-3
Subtract (–) 3-4
Negation (–) 3-4

Overflow and Division by Zero 3-4
Logical Operators 3-5
.AND. 3-5
.OR. 3-5
.XOR. 3-5
Relational Operators 3-6
Trigonometric Operators 3-7
SIN([expr]) 3-7
COS([expr]) 3-7
TAN([expr]) 3-8
ATN([expr]) 3-8
Comments on Trigonometric Functions 3-8
Functional Operators 3-9
ABS([expr]) 3-9
NOT([expr]) 3-9
INT([expr]) 3-9
PI 3-9
SGN([expr]) 3-10
SQR([expr]) 3-10
RND 3-10
Logarithmic Operators 3-11
LOG([expr]) 3-11
EXP([expr]) 3-11
String Operators 3-11
ASC([expr]) 3-11
CHR([expr]) 3-14
Special Function Operators 3-15
and @ 3-15
EOF 3-15
FREE 3-15
LEN 3-16
MTOP 3-16
CBY([expr]) 3-16
DBY([expr]) 3-17
XBY([expr]) 3-17
TIME 3-18

BASIC Commands 4-1

BRKPNT 4-1
CONT 4-3
Control C 4-4
Disabling and Enabling Control C 4-5
Control S 4-6
Control Q 4-7
EDIT 4-8
ERASE 4-9
IDLE 4-9
LIST 4-10
LIST@ 4-11
LIST# 4-11
MODE 4-12
NEW 4-13
NULL 4-13
PROG 4-14
PROG1 4-15
PROG2 4-16
RAM 4-18
REM 4-18
REN 4-19
ROM 4-20
RROM 4-21
RUN 4-22
SNGLSTP 4-23
VER 4-24
XFER 4-25

Command Line Calls 5-1

CALL 73 - Battery Backed RAM Disable 5-1
CALL 74 - Battery Backed RAM Enable 5-1
CALL 77 - Protected Variable Storage 5-2
CALL 81 - User Memory Module Check and Description 5-3
CALL 82 - Check User Memory Module Map 5-4
CALL 101 - Upload User Memory Module Code to Host 5-4
CALL 103 - Print PRT1 Output Buffer and Pointer 5-5
CALL 104 - Print PRT1 Input Buffer and Pointer 5-6
CALL 109 - Print Argument Stack 5-7
CALL 110 - Print PRT2 Output Buffer Pointer 5-8
CALL 111 - Print PRT2 Input Buffer Pointer 5-9

Assignment Functions 6-1

CLEAR 6-1
CLEARI 6-2
CLEARS 6-2
DATA 6-3
DIM 6-4
LET 6-5
RESTORE 6-6

Control Functions 7-1

CLOCK1 7-1
CLOCK0 7-2
DO-WHILE 7-2
DO-UNTIL 7-4
END 7-4
FOR-TO-(STEP)-NEXT 7-5
GOTO 7-7
IF-THEN-ELSE 7-8
NEXT 7-9
ON-GOTO 7-10

Execution Control and Interrupt Support Functions 8-1

CALL 70 - ROM to RAM Program Transfer 8-1
CALL 71 - ROM/RAM to ROM Program Transfer 8-2
CALL 72 - RAM/ROM Return 8-3
GOSUB 8-4
ONERR 8-5
ON-GOSUB 8-6
ONTIME 8-7
PUSH 8-8
POP 8-10
RETI 8-11
RETURN 8-11
STOP 8-13

Math and Backplane Conversion Functions 9-1

CALL 14 - 16 Bit Signed Integer to BASIC Floating-Point 9-1
CALL 15 - 16 Bit Unsigned Integer to BASIC Floating-Point 9-1
CALL 24 - BASIC Floating-Point to 16 Bit Signed Integer 9-2
CALL 25 - BASIC Floating-Point to 16 Bit Binary 9-3

Clock/Calendar Functions 10-1

CALL 40 - Set Clock/Calendar Time 10-1
CALL 41 - Set Clock/Calendar Date 10-2
CALL 42 - Set Day of Week 10-3
CALL 43 - Retrieve Date/Time String 10-3
CALL 52 - Retrieve Date String 10-4
CALL 44 - Retrieve Date Numeric 10-4
CALL 45 - Retrieve Time String 10-5
CALL 46 - Retrieve Time Numeric 10-6
CALL 47 - Retrieve Day of Week String 10-6
CALL 48 - Retrieve Day of Week Numeric 10-7

Status Functions 11-1

CALL 36 - Get Number of Characters in PRT2 Buffers 11-1
CALL 51 - Check CPU Output Image Buffer 11-2
CALL 55 - Check CPU Input Image Buffer 11-3
CALL 58 - Check M0 File 11-4
CALL 59 - Check M1 File 11-5
CALL 75 - Check SLC 500 Controller CPU Status 11-6
CALL 80 - Check Battery Condition 11-7
CALL 86 - Check DH485 Interface File Remote Write Status . . . 11-7
CALL 87 - Check DH485 Interface File Remote Read Status . . . 11-8
CALL 95 - Get Number of Characters in PRT1 Buffers 11-9
CALL 97 - Enable Port PRT2 DTR Signal 11-10
CALL 98 - Disable Port PRT2 DTR Signal 11-10
CALL 108 - Enable DF1 Driver Communications 11-11
CALL 113 - Disable DF1 Driver Communications 11-18
CALL 120 - Clear BASIC Module Input and Output Buffers 11-19
CALL 121 - Get SLC Processor Program ID Number 11-20

Output Functions 12-1

CALL 31 - Display Current PRT2 Port Setup 12-1
CALL 37 - Clear PRT2 Input/Output Buffers 12-1
CALL 54 - Transfer BASIC Output Buffer to CPU Input Image . . . 12-2
CALL 57 - Transfer BASIC Output Buffer to CPU M1 File 12-3
CALL 85 - Transfer BASIC Output Buffer to DH-485 Common Interface File 12-4
CALL 91 - Write BASIC Output Buffer to Remote DH-485 Data File 12-5
CALL 93 - Write Output Buffer to Remote DH-485 Common Interface File 12-8
CALL 94 - Display Current PRT1 Port Setup 12-11
CALL 96 - Clear PRT1 Input/Output Buffers 12-11
CALL 112 - User LED Control 12-12
CALL 114 - Transmit DF1 Packet 12-12
CALL 115 - Check DF1 XMIT Status 12-13

PRINT 12-14
PH0., PH1. 12-16
ST@ 12-17

Input Functions 13-1

CALL 35 - Get Numeric Input Character from PRT2 13-1
CALL 53 - Transfer CPU Output Image to BASIC Input Buffer . . . 13-3
CALL 56 - Transfer CPU M0 File to BASIC Input Buffer 13-4
CALL 84 - Transfer DH-485 Interface File to BASIC Input Buffer . 13-5
CALL 90 - Read Remote DH-485 Data File to BASIC Input Buffer 13-6
CALL 92 - Read Remote DH-485 Common Interface File to BASIC Input Buffer 13-9
CALL 117 - Get DF1 Packet Length 13-11
GET 13-12
INPL 13-13
INPS 13-13
INPUT 13-14
LD@ 13-16
READ 13-18

Setup Functions 14-1

CALL 30 - Set PRT2 Port Parameters 14-1
CALL 78 - Set Program Port Baud Rate 14-2
CALL 99 - Reset Print Head Pointer 14-3
CALL 105 - Reset PRT1 to Default Settings 14-3
CALL 119 - Reset PRT2 to Default Settings 14-4
MODE 14-4

String Functions 15-1

CALL 60 - String Repeat 15-1
CALL 61 - String Append 15-2
CALL 62 - Number to String Conversion 15-3
CALL 63 - String to Number Conversion 15-4
CALL 64 - Find a String in a String 15-5
CALL 65 - Replace a String in a String 15-6
CALL 66 - Insert a String in a String 15-7
CALL 67 - Delete a String in a String 15-8
CALL 68 - Find the Length of a String 15-9
STRING 15-9

Decimal/Hexadecimal/Octal/ASCII Conversion Table A-1

BASIC Command, Statement, and CALL Quick Reference Guide B-1

Preface

Manual Overview

The BASIC Language Reference is a manual you use when programming BASIC programs. This manual is intended to be used for reference purposes only.

This manual is divided into two parts. Part 1 provides reference information on BASIC language fundamentals. Part 2 provides reference information on BASIC statements and functions. Table P.1 lists an overview of the chapters in this manual.

Table P.1
Manual Overview

Part 1	Chapter	Description
	1	BASIC language elements
	2	BASIC data types
	3	BASIC expressions and operators
Part 2	**Chapter**	**Description**
	4	BASIC commands
	5	BASIC command line calls
	6	BASIC assignment functions
	7	BASIC control functions
	8	BASIC execution control and interrupt support functions
	9	BASIC math conversion functions
	10	BASIC clock/calendar functions
	11	BASIC status functions
	12	BASIC output functions
	13	BASIC input functions
	14	BASIC set up functions
	15	BASIC string functions

The following documents are also available:

Document Name	Pub. No.
SLC 500 System Overview	1747-2.30
APS Development Software Programming Manual	1747-ND001
SLC 500 Fixed Hardware Style Installation Manual	1747-800
SLC 500 Modular Hardware Style Installation Manual	1747-804
Hand-Held Terminal Programming Manual	1747-809
SLC 500 BASIC Module Design and Integration Manual	1746-ND005
SLC 500 BASIC Development Software Programming Manual	1746-NM001

WARNINGS, CAUTIONS and Important Information

We use the labels **WARNING**, **CAUTION**, and **Important** to identify the following kinds of information:

WARNING: This warning symbol indicates circumstances or practices that may lead to personal injury as well as damage to equipment if the procedures are not followed properly.

WARNING: This warning symbol indicates circumstances or practices where there is a potential shock hazard and people may be injured if the procedures are not followed properly.

CAUTION: This caution symbol indicates circumstances or practices that may lead to equipment or property damage if the procedures are not followed properly.

CAUTION: This caution symbol indicates circumstances or practices where equipment or property may be damaged due to electrical shock or discharge, when the procedures are not followed properly.

Important: Indicates information that is especially important for successful application of the BASIC module.

Conventions

To make this manual easier for you to read and understand, full product names and features are shortened where possible. Here are the shortened terms:

- **BASIC** - the BASIC-52 programming language
- **BASIC module** - SLC 500 BASIC module (Catalog Number 1746-BAS)
- **BASIC Development Software** - BASIC Development Software (Catalog Number 1747-PBASE)
- **Console device** - the device connected to the BASIC module program port. This device is used as an interface between the user and the BASIC program
- **DH-485** - network communication protocol
- **EEPROM** - Electrically Erasable Programmable Read Only Memory
- **[expr]** - Denotes an expression used with a system command, operator, or statement
- **[ln num]** - Denotes a line number used with a system command, operator, or statement
- **memory module -** BASIC module EEPROM or UVPROM
- **Program port** - port used to program the BASIC module. Either port PRT1 or port DH485 of the BASIC module can be used as the program port. If port DH485 is the program port, programming must be done with a console device that can communicate over the DH-485 network
- **RAM** - Random Access Memory
- **ROM -** Read Only Memory, refers to the optional memory module memory space (EEPROM or UVPROM)
- **RS-423/232** - serial communication interface
- **RS-422** - differential communication interface
- **RS-485** -network communication interface
- **SCADA** - Supervisory Control and Data Acquisition
- **scalar variable** - a variable with a single value

- **SLC 500**™ - SLC 500™ fixed or modular controller
- **UVPROM** - Ultra Violet Erasable Programmable Read Only Memory
- **[var]** - Denotes a variable used with a system command, operator, or statement

Reference Page Format

Part 2 of this manual provides a reference list of BASIC commands, statements, and operators. All commands, statements, and operators are presented in the following format:

{BASIC Command, Statement, or Operator}

Purpose:

This section provides an overview of the BASIC commands, statements, or operators. The following information is covered:

- typical uses in a BASIC program
- any programming restrictions that must be followed
- interaction with other BASIC functions
- interaction with the BASIC module

Syntax:

This section lists the syntax that must be followed when programming one of the BASIC commands, statements, or operators.

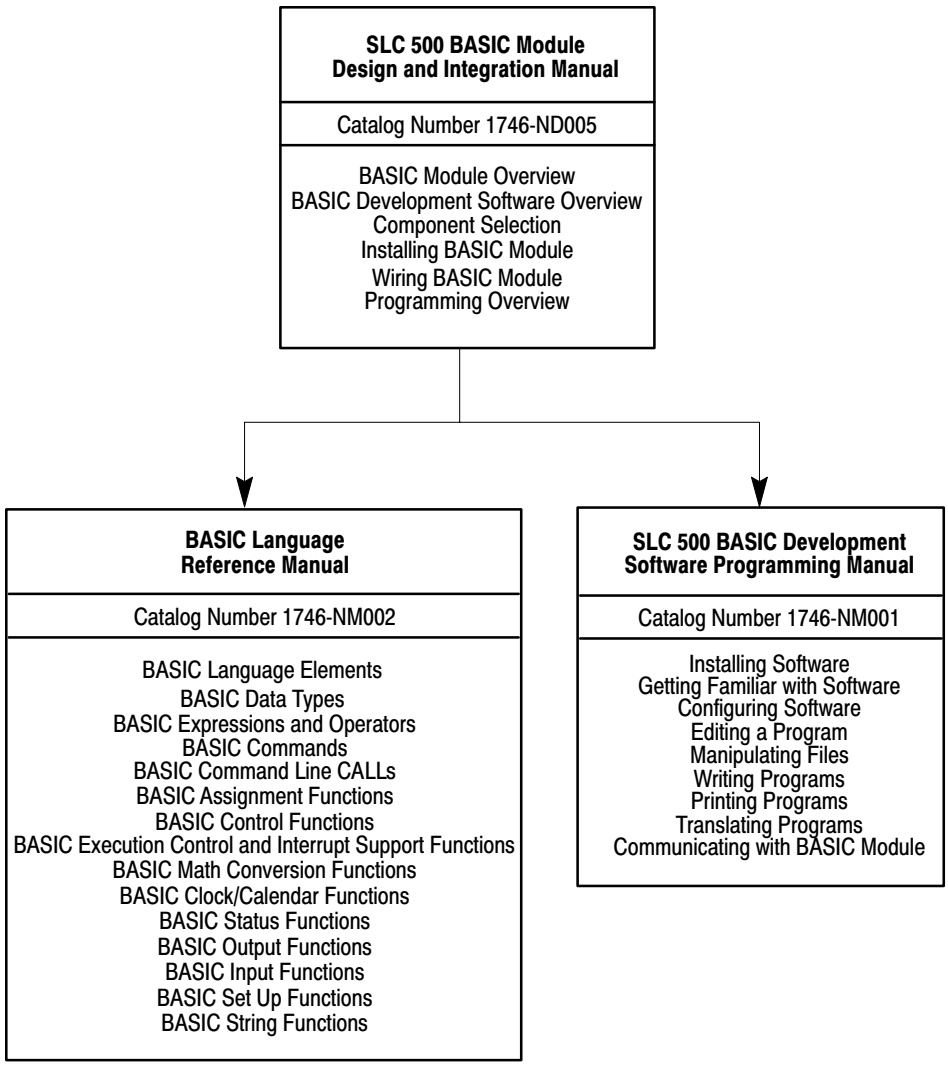
Example:

This section lists actual programming examples that incorporate the current BASIC command, statement, or operator.

BASIC Module Documentation Set

Your BASIC module documentation set is organized into manuals according to the tasks you must perform to design, integrate, and program your BASIC module. Figure P.1 shows the documents in the BASIC module documentation set and the information contained in each manual.

Figure P.1
BASIC Module Documentation Set



Chapter 1

Language Elements

Character Set

BASIC programs are composed of a group of BASIC program lines. Each BASIC program line is composed of a group of ASCII characters. Refer to Appendix A for a complete listing of ASCII character codes.

The BASIC Program Line

BASIC program lines consist of a BASIC line number and BASIC statements and operators. BASIC program lines are restricted to the BASIC line length.

BASIC Line Numbers

We refer to BASIC line numbers as:

[ln num]

BASIC line numbers indicate the order that the program lines are stored in memory and are also used as references when branching and editing. This number may be any whole integer from 1 to 65535. Typically you start numbering BASIC programs with line number 10 and increment by 10. This allows you to add additional lines later as you work on your program.

Since the computer runs the statements in numerical order, additional lines need not appear in consecutive order on the screen. If you enter line 35 after line 40, the computer still runs line 35 after line 30 and before line 40. This technique saves you from reentering an entire program if you forget to include a line.

Important: The first line of your program must be a comment.

Typically, the line numbers of a program start out looking like the first column and end up looking something like the second column below:

```
#1                    #2
10                    5
20                    7
30                    10
40                    15
50                    20
60                    30
70                    35
80                    40
.                     .
.                     .
.                     .
```

Important: Reuse of an existing line number causes all of the information referenced by the original line number to be lost. Be careful when entering numbers in the command mode, as you may accidentally erase some program lines. You may delete an existing line by retyping it with no information following it and pressing the [**RETURN**] key.

BASIC Statements, Commands, and Operators

BASIC program lines consist of a BASIC line number and BASIC statements and operators. Depending on the logic of your program, there may be more than one statement on a line. If so, each must be separated by a colon (:).

BASIC Line Length

A BASIC program line always begins with a line number and must contain at least one character, but no more than 68 characters. A program line ends when you press the [**RETURN**] key.

Chapter 2

Data Types

Data Types

Data types are broken down into three sections: argument stack, string and numeric elementary data types, and backplane conversion data.

Argument Stack

The argument stack (A-stack) stores all constants that the BASIC module is currently using. Operations such as add, subtract, multiply and divide always operate on the first two numbers of the argument stack and return the result to the stack. The argument stack is 203 bytes long. Each floating point number placed in the stack requires 6 bytes of storage. The argument stack can hold up to 33 floating point numbers before overflowing.

In addition, the PUSH command saves data to the argument stack and the POP command restores data from the stack. PUSHes and POPs are typically associated with CALLs. PUSHes and POPs are mechanisms used to transfer information to and from CALL routines.

PUSH makes a copy of the variable being PUSHed, then puts that copy on the top of the argument stack. POP takes the value on the top of the argument stack and copies it to the variable being POPped.

String Data Types

A string is a character or group of characters stored in memory. Usually, the characters stored in a string make up a word or a sentence. Strings allow you to use characters instead of numbers. Strings are shown as:

$([expr])

The BASIC module uses single-dimension string variables, $([expr]). The dimension of a string variable (the [expr] value) ranges from 0 to 254. This means that you can define and manipulate 255 different strings in the BASIC module. Initially, no memory is allocated for strings. Memory is allocated using the STRING statement. Strings are declared and manipulated through the $ operator.

When allocating memory for a string, you must account for the overhead bytes used by BASIC to manipulate strings. BASIC uses one overhead byte per string being declared plus one additional overhead byte.

For example:

```
String 106,20
```

Allocates space for five 20 byte strings (100 bytes) and includes five overhead bytes (1 per string) and one additional overhead byte.

In the BASIC module you can define strings with the LET statement, the INPUT statement, and with the ASC operator.

Example:

```
>10 STRING 100,20
>20 $(1)="THIS IS A STRING, "
>30 INPUT "WHAT'S YOUR NAME? - ",$(2)
>40 PRINT $(1),$(2)
>50 END

READY
>RUN

WHAT'S YOUR NAME? - FRED
THIS IS A STRING, FRED

READY
>
```

You can also assign strings to each other with a LET statement.

Example:

```
LET $(2)=$(1)
```

Assigns the string value in $(1) to the STRING $(2).

Numeric Data Types

There are two different numeric data types:

- integer numbers
- floating-point numbers

You can enter and display numbers in four formats: integer, decimal, hexadecimal and exponential.

Example:

```
129, 34.98, 0A6EH, 1.23456E+3
```

The BASIC module interprets all numbers as floating point numbers except when performing logical operations. When performing logical operations, the BASIC module converts floating point numbers to integers, performs the operation, then converts the result back to floating point.

Integer Numbers

The BASIC module operates on unsigned 16-bit integers that range from 0 to 65535 or 0FFFFH. You can enter all integers in either decimal or hexadecimal format. You indicate a hexadecimal number by placing the character "H" after the number (example: 170H). If the hexadecimal number begins with A - F, then it must be preceded by a zero (example: you must enter A567H as 0A567H). When an operator, such as .AND. requires an integer, the BASIC module truncates the fraction portion of the number so it fits the integer format. Integers are shown as:

[integer]

Important: The SLC 500 processor operates on signed 16-bit integers that range from -32768 to 32767. If an integer value larger than 32767 is passed to the processor from the BASIC module, that value will be interpreted as negative by the processor.

Floating-Point Numbers

In the BASIC module, all numbers are stored as floating-point numbers. Floating-point numbers are numbers in which the decimal point "floats" depending on the significant digits of a specific number. The processor accounts for the location of the decimal point. This allows the processor to store only the significant digits of a value, thus saving memory space.

You can represent the following range of numbers in the BASIC module:

+1E -127 to +.99999999 +127

There are eight significant digits. Numbers are internally rounded to fit this precision.

Backplane Conversion Data

The BASIC module communicates with the local processor through the SLC 500 I/O backplane. All data communicated to and from the SLC 500 is in SLC 500 format. The SLC 500 formats are:

- 16 Bit Signed Integer (-32768 to 32767)
- 16 Bit Binary (0000000000000000 to 1111111111111111)

Important: Any integer larger than 32767 is interpreted as a negative number by the SLC 500 CPU.

Variables

Variables that include a single-dimension expression [expr] are dimensioned or arrayed variables. Variables that contain a letter or a letter and a number are scalar variables. Any variables entered in lower case are changed to upper case. Variables are shown as:

[var]

The BASIC module allocates variables in a “static” manner, which means the first time a variable is used, the BASIC module allocates a portion of memory (8 bytes) specifically for that variable. This memory cannot be de-allocated on a variable to variable basis. This means that if you execute a statement (example: >10 Q = 3), you cannot tell the BASIC module that the variable Q no longer exists to “free up” the 8 bytes of memory that belong to Q. You can clear the memory allocated to variables by executing a CLEAR statement. The CLEAR statement “frees” all memory allocated to variables. Variables may be set aside for reuse to save memory.

Important: The BASIC module requires less time to find a scalar variable because there is no expression to evaluate. To run a program as fast as possible, use single-dimension variables only when necessary. Use scalar variables for intermediate variables and assign the final result to a dimensioned variable. Also, put the most frequently used variables first. Variables defined first require the least amount of time to locate.

Variable Names

Variables may represent either numeric values or strings. Variable names can only be eight (8) characters long. The BASIC module compares the first, last, and number of characters in a variable name with the first, last, and number of characters in other variable names to determine if it is a unique variable name. The characters allowed in a variable name are letters, numbers, and the decimal point. Special type declaration characters are also allowed.

A variable can be:

- a letter (example: `A, X, I`)
- a letter followed by a single-dimension expression, (example: `J(4), G(A+6), I(10*SIN(X))`)
- a letter followed by a number followed by a single-dimension expression (example: `A1(8), P7(10*SIN(X)), W8(A + C`).
- a letter followed by a number (0 to 9) or letter (example: `AA, AC, XX, A1, X3, G8`) except for the following combinations: `CR, DO, IE, IF, IP, ON, PI, SP, TO, UI, UO`.

Important: Reserved words (words already used in BASIC functions or statements) cannot be used as variable names.

Variable Types

Type declaration characters indicate what a variable represents. The following type declaration character is recognized:

Character	Variable Type
$	String variable

The only other legal variable type is a floating point variable. Floating point variables do not require a type declaration.

Chapter 3

Expressions and Operators

Expressions and Operators

An expression is a logical mathematical expression that involves operators, constants, and variables. There are eight types of operators that may act on an expression:

- arithmetic
- logical
- relational
- trigonometric
- functional
- logarithmic
- string
- special function

Expressions

Expressions are simple or complex.

Example:

Simple expression: `12*EXP(A)/100,H(1) + 55,`

or

Complex expression: `(SIN(A)*SIN(A)+COS(A)* COS(A)/2)`

A “stand alone” variable [var] or constant [const] is also considered an expression. Expressions are shown as:

[expr]

Operators

An operator performs a predefined operation on variables or constants. Operators require either one or two operands. Typical two operand operators include ADD(+), SUBTRACT(-), MULTIPLY(*) and DIVIDE(/). We call operators that require only one operand, single-operand operators. Typical single-operand operators are SIN, COS and ABS.

Hierarchy of Operators

The hierarchy of operators is the order that the operations in an expression are performed. You can write complex expressions using only a small number of parentheses. To illustrate the hierarchy of operators, examine the following equation:

4+3*2 = ?

In this equation, multiplication has precedence over addition. Therefore, multiply (3*2) and then add 4.

4+3*2 = 10

When an expression is scanned from left to right, an operation is not performed until an operator of lower or equal precedence is encountered. In the example, you cannot perform addition until the multiplication operation is complete because multiplication has a higher precedence. Use parentheses if you are in doubt about the order of precedence or to enhance program readability. The precedence of operators from highest to lowest in the BASIC module is:

1. Operators that use parentheses ( )
2. Exponentiation (**)
3. Negation (-)
4. Multiplication (*) and division (/)
5. Addition (+) and subtraction (-)
6. Relational expressions (=, <>, >, >=, <, <=).
7. Logical AND (.AND.)
8. Logical OR (.OR.)
9. Logical XOR (.XOR.)

Arithmetic Operators

The BASIC module contains a complete set of arithmetic operators. We divided the operators into two groups: dual-operand operators and single-operand operators.

The general form of all dual operand instructions is:

(expr) OP (expr), where OP is one of the following arithmetic operators

Add (+)

Use the Addition operator to add the first and the second expressions together.

Example: `>PRINT 3+2`

Result: `5`

Divide (/)

Use the Division operator to divide the first expression by the second expression.

Example: `>PRINT 100/5`

Result: `20`

Exponentiation (**)

Use the Exponentiation Operator to raise the first expression to the power of the second expression. The maximum power to which you can raise a number is 255.

Example: `>PRINT 2**3`

Result: `8`

Multiply (*)

Use the Multiplication operator to multiply the first expression by the second expression.

Example: `>PRINT 3*3`

Result: `9`

Subtract (–)

Use the Subtraction operator to subtract the second expression from the first expression.

Example: `>PRINT 9-6`

Result: `3`

Negation (–)

Use the Negation operator to change an expression from positive to negative.

Example: `>PRINT -(9+4)`

Result: `-13`

Overflow and Division by Zero

During the evaluation of an expression if an overflow, underflow, or division by zero error occurs, the BASIC module generates error messages and reverts to command mode. Refer to the ONERR operation on page 8–5 for more information on how to trap these errors.

The largest result allowed from any calculation is 0.99999999 E+127. If this number is exceeded, the BASIC module generates the “ERROR: ARITH. OVERFLOW” error message and returns to command mode.

The smallest result allowed from any calculation is 0.99999999 E–128. If this number is exceeded, the BASIC module generates the “ERROR: ARITH. UNDERFLOW” error message and returns to command mode.

If an attempt is made to divide any number by zero, the BASIC module generates the “ERROR: DIVIDE BY ZERO” error message and returns to command mode.

```
>10 PRINT 9/0
>20 PRINT “PROGRAM SHOULD NOT GET HERE.”

READY
>RUN

ERROR: DIVIDE BY ZERO – IN LINE  10

10     PRINT 9/0
----------------X
READY
>
```

```
>10 PRINT 9.9E126*(2)
>

READY
>RUN

ERROR: ARITH. OVERFLOW – IN LINE  10

10     PRINT 9.9E126*(2)
------------------------X
READY
>
```

Logical Operators

The BASIC module contains a complete set of logical operators. We divided the operators into two groups: dual operand operators and single operand operators.

The general form of all dual operand instructions is:

`(expr) OP (expr)`, where OP is one of the following logical operators.

.AND.

Use the logical .AND. operator to logically “AND” expressions together.

Example: `>PRINT 3.AND.2`

Result: `2`

.OR.

Use the logical .OR. operator to logically “OR” expressions together.

Example: `>PRINT 1.OR.4`

Result: `5`

.XOR.

Use the logical exclusive .XOR. operator to logically “XOR” expressions together.

Example: `>PRINT 7.XOR.6`

Result: `1`

Comments on logical operators .AND., .OR., and .XOR.

These operators perform BIT-WISE logical operations on numbers between 0 (0000H) and 65535 (0FFFFH) inclusive. If the argument is outside this range, then the BASIC module generates an "ERROR: BAD ARGUMENT" error message and returns to command mode. All decimal places are truncated, not rounded. Use the following chart for bit manipulations on 16 bit values.

X	Y	X .AND.Y	X .OR.Y	X .XOR.Y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Relational Operators

Relational expressions involve the operators =, < >, >, >=, <, and <=. In the BASIC module, relational operations are typically used to "test" a condition. The BASIC Module relational operators return a result of 65535 (0FFFFH) if the relational expression is true, and a result of 0 if the relation expression is false. The result returns to the argument stack. Because of this, it is possible to display the result of a relational expression. Relational expressions are shown as:

[rel expr]

Examples:

```
>PRINT 1=0
0
>PRINT 1>0
65535
>PRINT A<>A
0
>PRINT A=A
65535
```

You can "chain" relational expressions with the logical operators .AND., .OR., and .XOR.. This makes it possible to test a complex condition with ONE statement.

Example:

```
>10 IF (A>E).AND.(A>C).OR.(A>D)THEN...
```

Additionally, you can use the NOT([expr]) operator.

Example:

```
>10 IF NOT(A>E).AND.(A>C)THEN...
```

By “chaining” relational expressions with logical operators, you can test particular conditions with one statement.

Important: When using logical operators to link relational expressions, you must be sure operations are performed in the proper sequence. When in doubt, use parentheses.

Trigonometric Operators

The BASIC module contains a complete set of trigonometric operators. These operators are single operand operators.

SIN([expr])

Use the SIN operator to return the sine of the argument. The argument is expressed in radians. Calculations are carried out to 7 significant digits. The argument must be between ±200000.

Examples:

```
>PRINT SIN(PI/4)      >PRINT SIN(0)
.7071067              0
```

COS([expr])

Use the COS operator to return the cosine of the argument. The argument is expressed in radians. Calculations are carried out to 7 significant digits. The argument must be between ±200000.

Examples:

```
>PRINT COS(PI/4)      >PRINT COS(0)
.7071067              1
```

TAN([expr])

Use the TAN operator to return the tangent of the argument. The argument is expressed in radians. The argument must be between ±200000.

Examples:

```
>PRINT TAN(PI/4)      >PRINT TAN(0)
1                     0
```

ATN([expr])

Use the ATN operator to return the arctangent of the argument. The result is in radians. Calculations are carried out to 7 significant digits. The ATN operator returns a result between -PI/2 (3.1415926/2) and PI/2.

Examples:

```
>PRINT ATN(PI)        >PRINT ATN(1)
1.2626272             .78539804
```

Comments on Trigonometric Functions

The SIN, COS and TAN operators use a Taylor series to calculate the function. These operators first reduce the argument to a value between 0 and PI/2. This reduction is accomplished by the following equation:

```
reduced argument=(user arg/PI - INT(user arg/PI) *PI
```

The reduced argument, from the above equation, is between 0 and PI. The reduced argument is then tested to see if is greater than `PI/2`. If it is, then it is subtracted from `PI` to yield the final value. If it is not, then the reduced argument is the final value.

Although this method of angle reduction provides a simple and economical means of generating the appropriate arguments for a Taylor series, there is an accuracy problem associated with this technique. The accuracy problem is noticed when the user argument is large (example: `greater than 1000`). This is because significant digits in the decimal (fraction) portion of the reduced argument are lost in the `(user arg/PI - INT (user arg/PI))` expression. As a general rule, keep the arguments for the trigonometric functions as small as possible.

Functional Operators

The BASIC module contains a complete set of functional operators. These operators are single-operand operators.

ABS([expr])

Use the ABS operator to return the absolute value of the expression.

Examples:

```
>PRINT ABS(5)          >PRINT ABS(-5)
5                      5
```

NOT([expr])

Use the NOT operator to return a 16 bit one's complement of the expression. The expression must be a valid integer (example: `between 0 and 65535 (0FFFFH) inclusive`). Non-integers are truncated, not rounded.

Examples:

```
>PRINT NOT(65000)      >PRINT NOT(0)
535                    65535
```

INT([expr])

Use the INT operator to return the integer portion of the expression.

Examples:

```
>PRINT INT(3.7)        >PRINT INT(100.876)
3                      100
```

PI

PI is a stored constant. In the BASIC module PI is stored as 3.1415926

SGN([expr])

Use the SGN operator to return a value of +1 if the argument is greater than zero, zero if the argument is equal to zero, and -1 if the argument is less than zero.

Examples:

```
>PRINT SGN(52)              >PRINT SGN(0)          >PRINT SGN(-8)
     1                          0                      -1
```

SQR([expr])

Use the SQR operator to return the square root of the argument. The argument may not be less than zero.

Examples:

```
>PRINT SQR(9)     >PRINT SQR(45)     >PRINT SQR(100)
3                 6.7082035          10
```

RND

Use the RND operator to return a pseudo-random number in the range between 0 and 1 inclusive. The RND operator uses a 16-bit binary seed and generates 65536 pseudo-random numbers before repeating the sequence. The numbers generated are specifically between 0/65535 and 65535/65535 inclusive.

Important: Unlike most BASIC languages, the RND operator in the BASIC module does not require an argument or a dummy argument. If an argument is placed after the RND operator, a BAD SYNTAX error occurs.

Examples:

```
>PRINT RND
.26771954
```

Logarithmic Operators

The BASIC module contains a complete set of logarithmic operators. These operators are single operand operators.

LOG([expr])

Use the LOG operator to return the natural logarithm of the argument. The argument must be greater than 0. This calculation is carried out to 7 significant digits.

Examples:

```
>PRINT LOG(12)          >PRINT LOG(EXP(1))
2.484906                1
```

EXP([expr])

Use the EXP operator to raise the number “e” (2.7182818) to the power of the argument.

Examples:

```
>PRINT EXP(1)           >PRINT EXP(LOG(2))
2.7182818               2
```

String Operators

Two operators in the BASIC module can manipulate STRINGS. These operators are ASC( ) and CHR( ).

ASC([expr])

Use the ASC operator to return the integer value of the ASCII character placed in the parentheses.

Example:

```
>1  REM EXAMPLE PROGRAM
>10 PRINT ASC(a)
>20 PRINT ASC(A)


READY
>RUN

 65
 65

READY
>
```

The decimal representation for the ASCII character “A” is 65. The decimal representation for the ASCII character “a” is 97. However, the BASIC module capitalizes all ASCII characters not contained within quotation marks. Similarly, special ASCII characters whose decimal value is greater than 127 should not be used.

In addition, you can evaluate individual characters in a pre-defined ASCII string with the ASC operator.

Example:

```
>1  REM EXAMPLE PROGRAM
>5  STRING 1000,40
>10 $(1) =“THIS IS A STRING”
>20 PRINT $(1)
>30 PRINT ASC($(1),1)
>40 END


READY
>RUN

THIS IS A STRING
 84

READY
>
```

When you use the ASC operator as shown above, the $([expr]) denotes what string is accessed. The expression after the comma selects an individual character in the string. In the above example, the first character in the string is selected. The decimal representation for the ASCII character “T” is 84. String character position 0 is invalid.

Example:

```
>NEW

>1  REM EXAMPLE PROGRAM
>5  STRING 1000,40
>10 $(1)="ABCDEFGHIKJL"
>20 FOR X = 1 TO 12
>30 PRINT ASC($(1),X),
>40 NEXT X
>50 END


READY
>RUN

65  66  67  68  69  70  71  72  73  75  74  76
READY
>
```

The numbers printed in the previous example represent the ASCII characters A through L.

You can also use the ASC operator to change individual characters in a defined string.

In general, the ASC operator lets you manipulate individual characters in a string. A simple program can determine if two strings are identical.

Example:

```
>NEW

>1  REM EXAMPLE PROGRAM
>5  STRING 1000,40
>10 $(1) = "ABCDEFGHIJKL"
>20 PRINT $(1)
>30 ASC($(1),1) = 75 : REM DECIMAL EQUIVALENT OF K
>40 PRINT $(1)
>50 ASC($(1),2) = ASC($(1),3)
>60 PRINT $(1)

READY
>RUN

ABCDEFGHIJKL
KBCDEFGHIJKL
KCCDEFGHIJKL

READY
>
```

CHR([expr])

Use the CHR operator to convert a numeric expression to an ASCII character.

Example:

```
>PRINT CHR(65)
A
```

Like the ASC operator, the CHR operator also selects individual characters in a defined ASCII string.

Example:

```
>NEW


>1  REM EXAMPLE PROGRAM
>5  STRING 1000,40
>10 $(1) = “The BASIC Module”
>20 FOR I = 1 TO 16 : PRINT CHR($(1),I),: NEXT I


READY
>RUN


The BASIC Module
READY
>
```

In the above example, the expressions contained within the parentheses, following the CHR operator have the same meaning as the expressions in the ASC operator.

Unlike the ASC operator, you CANNOT assign the CHR operator a value. A statement such as CHR ($(1),1)= H is INVALID and generates a BAD SYNTAX ERROR causing the BASIC module to go into command mode. Use the ASC operator to change a value in a string, or use the string support call routine - replace string in a string.

Important: Use the CHR function only in a print statement.

Special Function Operators

The BASIC module contains a complete set of special function operators. These operators are called special function operators because they manipulate the I/O hardware and memory addresses of the BASIC module.

and @

Use the # and @ operators to direct communications. Communication takes place through port PRT1 when the @ operator is programmed, and through port PRT2 when the # operator is programmed. The absence of either the # or @ operators indicates that communication should take place through a program port (port PRT1 or port DH485).

 # Example:

```
>10 A = GET#
```

Causes the next character in the PRT2 input buffer to be assigned to variable A.

@ Example:

```
>10 A = GET@
```

Causes the next character in the PRT1 input buffer to be assigned to variable A.

EOF

Use the EOF operator to test for an empty input buffer before executing an input statement or function. This prevents input statements from waiting indefinitely on empty input buffers. Use the EOF# statement to test for an empty input buffer for port PRT2. Use the EOF@ statement to test for an empty input buffer for port PRT1.

Example:

```
>10 REM EXAMPLE PROGRAM
>20 IF (NOT(EOF)) THEN A=GET
>30 REM IF BUFFER NOT EMPTY, READ SINGLE CHARACTER
```

FREE

Use the system control value FREE to tell you how many bytes of RAM memory are available to the user. When the current selected program is in RAM memory, the following relationship is true:

```
FREE = MTOP - LEN - 511
```

LEN

Use the system control value LEN to tell you how many bytes of memory the currently selected program occupies. This is the length of the program and does not include the size of string memory or the variables and array memory usage. You cannot assign LEN a value, it can only be read. A NULL program (example: `no program`) returns a LEN of 1. The 1 represents the end of program file character.

Important: The BASIC module does not require any "dummy" arguments for the system control values.

MTOP

Use the MTOP operator to retrieve the last valid memory address in RAM that is available to the BASIC module. After reset, the BASIC module sizes the external memory and assigns the last valid memory address to the system control value MTOP. The module does not use any external RAM memory beyond the value assigned to MTOP. If this value has not been changed by CALL 77, then the last valid BASIC address is 5FFFH (24575).

Examples:

```
>PRINT MTOP          or        PH0.MTOP
24575                          5FFFH
```

CBY([expr])

Use the CBY operator to retrieve data from the program or code memory address location of the BASIC module. You cannot assign CBY a value, it can only be read. The argument for the CBY operator must be a valid integer between 0 and 65535 (0FFFFH). If it is not a valid integer, a BAD ARGUMENT ERROR occurs.

Important: Improper use of this operator may cause a malfunction of the BASIC module.

Example:

```
A = CBY(1000)
```

Causes the value in the program or code memory address location 1000 to be assigned to variable A.

DBY([expr])

Use the DBY operator to retrieve or assign data to or from the internal data memory of the BASIC module. Both the value and the argument in the DBY operator must between 0 and 255 inclusive. This is because there are only 256 internal memory locations in the BASIC module and one byte can only represent a quantity between 0 and 255 inclusive.

Important: Improper use of this operator may cause a malfunction of the BASIC module.

Example:

```
A = DBY(B)
```

Causes the value in internal memory location B to be assigned to variable A. B must be between 0 and 255.

```
DBY(250) = CBY(1000)
```

Causes the value in program or code memory location 1000 to be assigned to internal memory location 250.

XBY([expr])

Use the XBY operator to retrieve or assign data to or from the external data memory of the BASIC module. The argument for the XBY operator must be a valid integer between 0 and 65535 (0FFFFH). The value assigned to the XBY operator must between 0 and 255 inclusive. If it is not, a BAD ARGUMENT ERROR occurs.

Important: Improper use of this operator may cause a malfunction of the BASIC module.

Example:

```
A = XBY(0F000H)
```

Causes the value in external memory location 0F00H to be assigned to variable A.

```
XBY(4000H) = DBY(100)
```

Causes the value in internal memory location 100 to be assigned to external memory location 4000H.

TIME

Use the TIME operator to retrieve or assign a value to the free running clock resident in the BASIC module. After reset, time is equal to 0. The CLOCK1 statement enables the free running clock. When the free running clock is enabled, the special function operator TIME increments once every 5 milliseconds. The units of time are in seconds.

When TIME is assigned a value with a LET statement (example: `TIME=100`), only the integer portion of TIME is changed.

Example:

```
>CLOCK1                (enable FREE RUNNING CLOCK)

>CLOCK0                (disable FREE RUNNING CLOCK)

>PRINT TIME            (display TIME)
3.315

>TIME = 0              (set TIME = 0)

>PRINT TIME            (display TIME)
.315                   (only the integer is changed)
```

You can change the “fraction” portion of TIME by manipulating the contents of internal memory location 71 (47H). You can do this by using a DBY(71) statement. Note that each count in internal memory location 71 (47H) represents 5 milliseconds of TIME.

Continuing with the example:

```
>DBY(71) = 0           (fraction of TIME = 0)

>PRINT TIME
0

>DBY(71) = 3           (fraction of TIME = 3*5ms = 15 ms)

>PRINT TIME
1.5 E-2
```

Only the integer portion of TIME changes when a value is assigned. This allows you to generate accurate time intervals. For example, if you want to create an accurate 12 hour clock: there are 43200 seconds in a 12 hour period, so an ONTIME 43200, [ln num] statement is used. When the TIME interrupt occurs, the statement TIME 0 is executed, but the millisecond counter is not re-assigned a value. If interrupt latency exceeds 5 milliseconds, the clock remains accurate.

Chapter 4

BASIC Commands

BRKPNT

Purpose:

Use the BRKPNT command to set a program break point at the line number specified by this command. Program execution stops just before the line number specified by the BRKPNT command. If the line number is zero, the break point is disabled. After the break point is reached, you can examine variables by using assignment statements. Continue from the break point by using the CONT command. Once the break point is reached, it is disabled. To stop at the same place twice, set the break point twice. The BRKPNT command works only on programs executing from RAM. It does not stop a program executing from ROM.

Syntax:

BRKPNT[ln num]

Example:

```
>1  REM EXAMPLE PROGRAM
>10    D=0 : SU=0 : AV=0
>20     REM GET 100 DATUM POINTS
>30     FOR I=1 TO 100
>40     REM GET ANOTHER DATUM
>50     GOSUB 140
>60     REM SUM THE DATA
>70     GOSUB 170
>80     NEXT I
>90     REM AVERAGE THE DATA
>100    GOSUB 200
>110    REM PRINT RESULT
>120    PRINT “THE AVERAGE VALUE IS ”,AV
>130    END
>140    REM THIS SUBROUTINE GENERATES RANDOM DATA
>150   D=RND
>160    RETURN
>170    REM THIS SUBROUTINE SUMS THE DATA
>180   SU=SU+D
>190    RETURN
>200    REM THIS SUBROUTINE AVERAGES THE DATA
>210   AV=SU/I
>220    RETURN

READY
>BRKPNT 160
```

```
Breakpoint enabled.

READY
>RUN

STOP – IN LINE  160
READY
>PRINT D,SU,AV
 .86042573  0  0

>D = .5

>PRINT D,SU,AV
 .5  0  0

>CONT

THE AVERAGE VALUE IS  .48060383

READY
>
```

CONT

Purpose:

Use the CONT command to resume execution of a program stopped by a Control C, BRKPNT command, or a STOP statement. If you stop a program by typing Control C on the console device or by execution of a STOP statement, you can resume execution of the program by typing CONT. If you enter a Control C while Control C is enabled, it stops the program. Use CONT to continue. Between the stopping and the re-starting of the program you may display the values of variables or change the values of variables. However, you cannot CONTinue if the program is modified during the STOP or after an error.

Important: Control C clears all input and output buffers.

Syntax:

CONT

Example:

```
>NEW

>1  REM EXAMPLE PROGRAM
>10 FOR I = 1 TO 10000
>20 PRINT I
>30 NEXT I
>40 END

READY
>RUN


 1
 2
 3
 4
 5
 6
 7
 8
 9
 10 [Typed [Ctrl-C] here]


STOP – IN LINE 15
READY
>CONT

20
21
22
```

Control C

Purpose:

Use the Control C command to stop execution of the current program and return the BASIC module to the COMMAND mode. In some cases you can continue execution of the program using a CONTinue. See the explanation for CONTinue for more information.

Important: Control C clears all input and output buffers.

Syntax:

[**CTRL-C**]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 FOR I = 1 TO 10000
>20 PRINT I
>30 NEXT I
>40 END
>RUN
  1
  2
  3
  4
  5 - (CONTROL C typed)

STOP - IN LINE 20

READY
>PRINT I
  27

>I =10

>CONT

  10
  11
  12
```

Notice that after Control C is typed and I is printed the value of I is 27. The value of I is incremented several times before Control C is detected.

Disabling and Enabling Control C

Important: Control C is enabled by default.

Purpose:

Use the Disabling Control C command to disable the Control C break function. You can do this by executing the following statement in a BASIC module program or from the command mode:

CALL 19

When CALL 19 is executed, the Control C break function for both LIST and RUN operations is disabled. Cycling power returns the Control C function to normal operation if it is disabled from the command mode.

Important: When Control C is disabled, you are unable to stop program execution through a BASIC command. Cycling power will re-enable Control C checking until the program once again disables Control C. To stop program execution, you must cycle power and type [**CTRL-C**] before the line that disables Control C is executed.

To re-enable the Control C break function, execute the following statement in a BASIC module program or from the command mode:

CALL18

Syntax:

CALL 18 or CALL 19

Example:

```
>1  REM EXAMPLE PROGRAM
>10 CALL19
.
.
.
>90 CALL 18
```

Control S

Purpose:

Use the Control S command to interrupt the scrolling of a BASIC program during the execution of a LIST command. Control S stops output from the transmitting port if you are running a program. In this case XOFF(Control S) operates as follows:

Important: Control S only works if you have enabled software handshaking on the program port.

1. XOFF only operates on PRINT statements.

2. When received during a PRINT, data output is suspended immediately but program execution continues.

3. When received at any other time, the program continues until encountering a PRINT statement. At this time data output is suspended. The program continues to fill the output buffer until the buffer is full.

4. XON (Control Q) is required to resume data output operation.

Syntax:

[**CTRL-S**]

Example:

```
> LIST
1  REM EXAMPLE PROGRAM
10 A = 1
20 DO
[CTRL-S]
.
.
.
[CTRL-Q]
30 A = A+1
40 PRINT A
50 WHILE A < 20

READY
>
```

In this example, the output is suspended when [**CTRL-S**] is typed. The output is continued after the [**CTRL-Q**] is typed.

Control Q

Purpose:

Use the Control Q command to restart a LIST command or PRINT output that is interrupted by a Control S.

Syntax:

[**CTRL-Q**]

Example:

```
> LIST
1  REM EXAMPLE PROGRAM
10 A = 1
20 DO
[CTRL-S]
.
.
.
[CTRL-Q]
30 A = A+1
40 PRINT A
50 WHILE A < 20

READY
>
```

In this example, the output is suspended when [**CTRL-S**] is typed. The output is continued after the [**CTRL-Q**] is typed.

EDIT

Purpose:

Use the EDIT command to access the BASIC line editor. Use this editor to edit a line of the current program in RAM. Table 4.A lists the BASIC editor operations.

Table 4.A
BASIC Editor Operations

Operation	Function	Key Strokes
Move	Use the Move operation to provide right/left cursor control.	[**Space bar**] - moves the cursor one space to the right. [**Backspace**] - moves the cursor one space to the left.
Replace	Use the Replace operation to replace the character at the current cursor position.	Press the key that corresponds to the character that will replace the character at the current cursor position.
Insert	Use the Insert operation to insert text at the current cursor position. **Important**: When you use the Insert operation, all text to the right of the cursor disappears until you type the second [**Ctrl-A**]. Total line length is 79 characters.	[**Ctrl-A**] **Important:** You must type a second [**Ctrl-A**] to terminate the Insert command.
Delete	Use the Delete operation to delete the character at the cursor position.	[**Ctrl-D**]
Exit	Use the Exit operation(s) to exit the editor with or without saving the changes.	[**Ctrl-Q**] - exits the editor and replaces the old line with the edited line. [**Ctrl-C**] - exits the editor without saving any changes made to the line.
Retype	Use the Retype operation to copy the current line of text and insert it at the line following the current line. The cursor is moved to the first character on the new line.	[**RETURN**]

Syntax:

EDIT

Example:

```
>EDIT 150
```

Displays program line number 150 for editing

ERASE

Purpose:

Use the ERASE command to delete the last BASIC program stored in EEPROM through a PROG command.

Syntax:

ERASE

Example:

```
>ERASE

>ROM 13 ERASED
```

The last program stored in EEPROM (ROM 13 in this example) is erased.

IDLE

Purpose:

Use the IDLE command to force the BASIC module to enter “wait until interrupt mode.” Program execution halts until an ONTIME condition is met. The ONTIME interrupt must be enabled before executing the IDLE command or else the BASIC module will enter a “wait forever mode.” Control C will exit the IDLE command if Control C is enabled.

Syntax:

IDLE

Example:

```
>1  REM EXAMPLE PROGRAM
>10 TIME = 0
>20 CLOCK1
>30 ONTIME 2,70
>40 IDLE
>50 PRINT “END OF TEST!!!”
>60 END
>70 PRINT “TIMER INTERRUPT AT - ”,TIME,“SECONDS.”
>80 RETI

READY
>RUN

TIMER INTERRUPT AT - 2.005 SECONDS.
END OF TEST!!!
```

LIST

Purpose:

Use the LIST command to print the program to the console device. Spaces are inserted after the line number, and before and after statements. This helps in the debugging of BASIC module programs. You can terminate the "listing" of a program at any time by typing a Control C on the console device. You can interrupt and continue the listing using Control S and Control Q.

Important: [**CTRL-C**] terminates the listing if [**CTRL-C]** checking is enabled. [**CTRL-S**] halts the listing until [**CTRL-Q**] is typed if software handshaking in enabled.

Syntax:

LIST [ln num]

LIST [ln num] – [ln num]

The first variation causes the program to print from the designated line number [ln num] to the end of the program. The second variation causes the program to print from the first designated line number [ln num] to the second designated line number [ln num].

Important: You must separate the two line numbers with a dash (–).

Example:

```
>LIST
1  REM EXAMPLE PROGRAM
10 PRINT "LOOP PROGRAM"
20 FOR I = 1 TO 3
30 PRINT I
40 NEXT I
50 END

READY
>LIST 30
1  REM EXAMPLE PROGRAM
30 PRINT I
40 NEXT I
50 END
```

```
READY
>LIST 20-40
1  REM EXAMPLE PROGRAM
20 FOR I = 1 TO 3
30 PRINT I
40 NEXT I
```

LIST@

Purpose:

Use the LIST@ command to print the program to the device attached to port PRT1. All comments that apply to the LIST command apply to the LIST@ command. We include these commands so that you can make "hard copy printouts" of a program. You must configure PRT1 port parameters to match your particular list device. The PRT1 parameters (baud rate, parity, stop bits, and so on) can be set using the MODE command.

Syntax:

LIST@

Examples:

```
LIST@ :REM LISTS ALL LINES IN THE PROGRAM

LIST@10-20 :REM LISTS LINES 10 THROUGH 20
```

LIST#

Purpose:

Use the LIST# command to print the program to the device attached to port PRT2. All comments that apply to the LIST command apply to the LIST# command. We include these commands so that you can make "hard copy printouts" of a program. You must configure the PRT2 port parameters to match your particular list device. The PRT2 parameters (baud rate, parity, stop bits, and so on) can be set using the MODE command.

Syntax:

LIST#

Example:

Refer to LIST@ examples.

MODE

Purpose:

Use the MODE command to set the port parameters of ports PRT1, PRT2, and DH485. Table 4.B lists the port parameters for ports PRT1 and PRT2.

Table 4.B
PRT1 and PRT2 Port Parameters

Port Parameters	Selections	Default Settings
baud rate	300, 600, 1200, 2400, 4800, 9600, 19200	1200
arg1 (parity)	None (N), Even (E), Odd (O)	N
arg2 (number of data bits)	7 or 8	8
arg3 (number of stop bits)	1 or 2	1
arg4 (handshaking)	No handshaking (N) Software handshaking (S) Hardware handshaking (H) Hardware and software handshaking (B)	S
arg5 (storage type)	Store information in user ROM and RAM (E) Store information in battery backed RAM (R)	R

Important: If any argument (other than port name and baud rate) is left blank, then that argument defaults to the previously specified value for that argument.

Table 4.C lists the port parameters for port DH485.

Table 4.C
DH485 Port Parameters

Port Parameters	Selections	Default Settings
baud rate	300, 600, 1200, 2400, 4800, 9600, 19200	19200
arg1 (host node address)	0 to 31	0
arg2 (module node address)	1 to 31	1
arg3 (maximum node address)	1 to 31	31
arg4 (not used)		
arg5 (storage type)	Store information in user ROM and RAM (E) Store information in battery backed RAM (R)	R

Important: If any argument (other than port name) is left blank, then that argument defaults to the previously specified value for that argument.

Syntax:

MODE(port name, baud rate, arg1, arg2, arg3, arg4, arg5)

Example:

```
>1  REM EXAMPLE PROGRAM
>10 MODE(DH485,19200,0,1,2,,R)
.
.
.
>25 MODE(PRT1,1200,N,8,,,)
```

Important: The E storage type option cannot be used if MODE is used as a statement.

NEW

Purpose:

Use the NEW command to delete the program currently stored in RAM memory. In addition, all variables are set equal to ZERO, all strings and all BASIC evoked interrupts are cleared. The free running clock, string allocation, and internal stack pointer values are not affected. The NEW command is used to erase a program and all variables in RAM.

Syntax:

NEW

Example:

```
>NEW

>LIST

>READY
>
```

NULL

Purpose:

Use the NULL command to determine how many NULL characters (00H) the BASIC module outputs after a carriage return in a print statement. After initialization this value is set to 0. Most printers contain a RAM buffer that eliminates the need to output NULL characters after a carriage return.

Syntax:

NULL[integer]

PROG

Purpose:

Important: Before you attempt to program EEPROM, read the PROG, PROG1 and PROG2 sections of this chapter.

Use the PROG command to program the resident EEPROM with the current program in RAM. The BASIC module cannot program UVPROMs.

Important: Be sure you have selected the program you want to save before using the PROG command. Your BASIC module does not automatically copy the RAM program to EEPROM. If an error occurs during EEPROM programming, the message “ERROR - Programming sequence failure” is displayed.

After you type PROG, the BASIC module displays the number in the EEPROM FILE the program occupies. Programming takes only a few seconds.

Syntax:

PROG

Example:

```
>LIST
1  REM EXAMPLE PROGRAM
10 FOR I=1 TO 10
20 PRINT I
30 NEXT I
40 END

READY
>PROG

ROM 12

Programming sequence successful.


READY
>ROM 12


>LIST
1  REM EXAMPLE PROGRAM
10 FOR I=1 TO 10
20 PRINT I
30 NEXT I
```

```
40 END

READY
>
```

In this example, the program just placed in the EEPROM is the 12th program stored.

Important: If you exceed the available EEPROM space, you cannot continue programming until it is erased. Use the ERASE command to erase the last program stored in EEPROM. Be sure to use CALL 81 or CALL 82 to determine memory space prior to programming your EEPROM. See the following chapter.

PROG1

Purpose:

Important: Before you attempt to program an EEPROM, read the PROG, PROG1 and PROG2 sections of this chapter.

Use the PROG1 command to program the resident EEPROM with port information for all three ports as well as store MTOP information. At module power-up, the BASIC module reads this information and initializes MTOP and all three serial ports. The sign-on message is sent to the console immediately after the module completes its reset sequence. If the baud rate on the console device changes, you must re-program the EEPROM to make the module compatible with the new console. Re-program by changing the appropriate port or MTOP information, then execute PROG1 again.

Syntax:

PROG1

Example:

```
READY
>PROG1

Programming sequence successful.
```

PROG2

Purpose:

Important: Before you attempt to program an EEPROM, read the PROG, PROG1 and PROG2 sections of this chapter.

Use the PROG2 command the same as you would the PROG1 command except for the following: instead of signing-on and entering the command mode, the module immediately begins executing the first program stored in the resident EEPROM. Otherwise, it executes the program stored in RAM.

You can use the PROG2 command to RUN a program on power-up without connecting to a console. Saving PROG2 information is the same as typing a ROM 1, RUN command sequence. This feature also allows you to write a special initialization sequence in BASIC and generate a custom sign-on message for specific applications. The PROG2 command does not alter the first program in the memory module.

Important: The PROG2 command does not cause the BASIC module to RUN at power-up if PRT1 default communications are selected via JW4. Refer to the Design & Integration manual, page 3–9 for more information.

The following figure shows the BASIC module operation from a power-up condition using PROG1 and PROG2, or battery backed RAM.

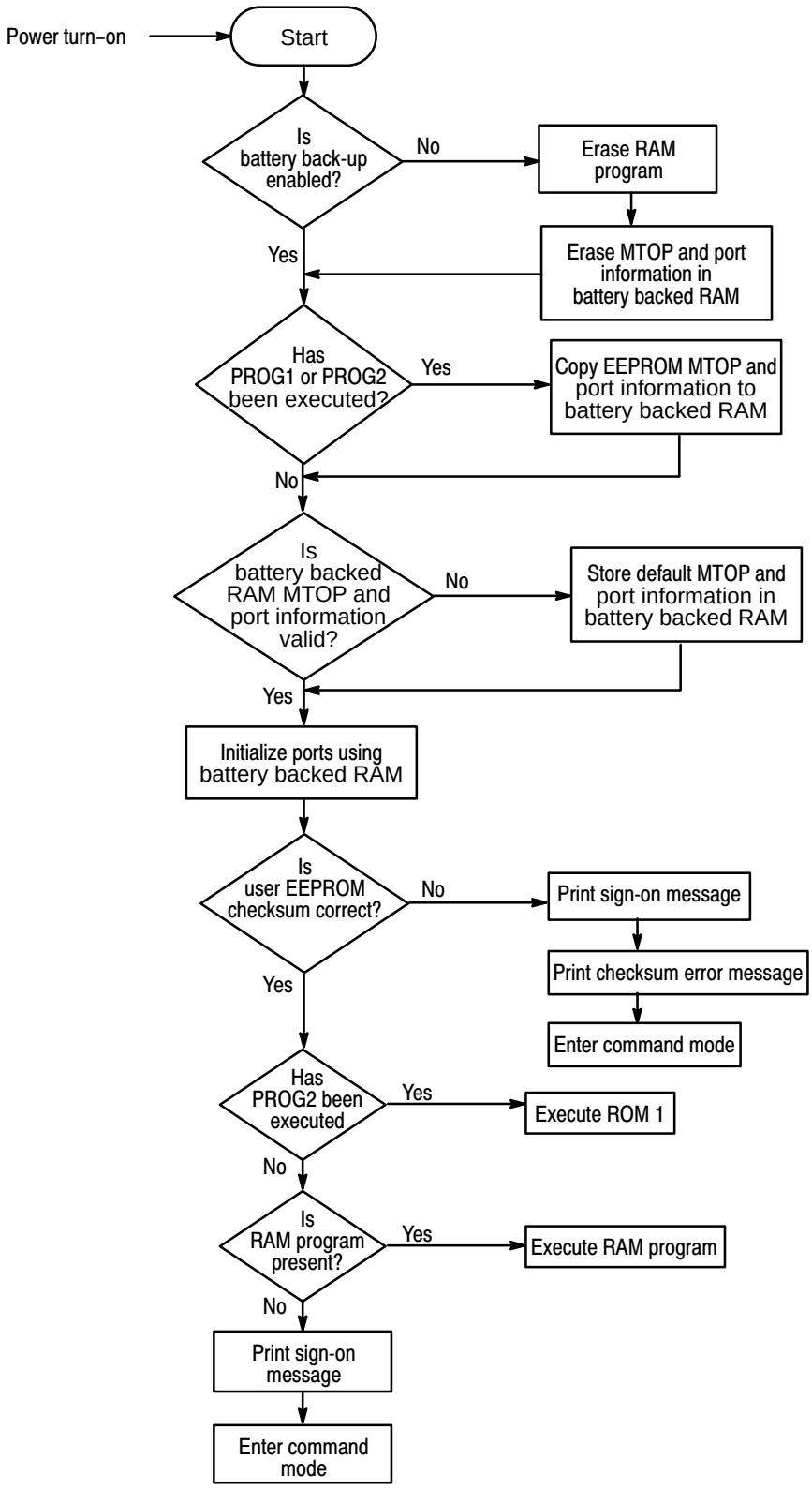
Syntax:

PROG2

Example:

```
READY
>PROG2

Programming sequence successful.
```



RAM

Purpose:

Use the RAM command to tell the BASIC module interpreter to select the current program out of RAM. The current program is displayed during a LIST command and executed when RUN is typed.

Important: RAM space is limited to 24K bytes. Use the following formula to calculate the available user RAM space:

Available user RAM = MTOP-H

H = LEN + S + 6*A + 8*V + 512

Where:

LEN = system control value that contains current RAM program length

S = number of bytes allocated for strings (first value in the STRING instruction)

A = sum of all (array sizes +1)

V = sum of all variable names used (including each array name)

Syntax:

RAM

Example:

```
READY
>RAM
```

REM

Purpose:

Use the REM command to specify a comment line in a BASIC program. Adding comment lines to a program makes the program easier to understand. Program lines that start with a REM command cannot be terminated with a colon (:). REM commands can be placed after a colon (:) in a program line. This allows you to place a comment on each line.

Important: REM commands add time to program execution. Use them selectively or place them at the end of the program where they do not affect program execution speed. Do not use REM commands in frequently called loops or subroutines.

Syntax:

REM

Example:

```
>10 REM THIS IS A COMMENT LINE
>20 NEW : REM THIS IS ALSO A COMMENT LINE
```

REN

Purpose:

Use the REN command to renumber program lines.

Syntax:

REN[new number],[old number],[increment]

Examples:

```
REN
```

Renumbers the entire program. The first new line number is 10. Line numbers increment by 10.

```
REN 20
```

Renumbers the entire program. The first new line number is 10. Line numbers increment by 20.

```
REN 300,50
```

Renumbers the entire program. The first new line number is 300. Line numbers increment by 50.

```
REN 1000,900,20
```

Renumbers the program from line 900 on up. Line number 900 becomes line number 1000. Any following line numbers increment by 20.

ROM

Purpose:

Use the ROM command to tell the BASIC module interpreter to select the current program out of EEPROM or UVPROM. The current program is displayed during a LIST command and executed when RUN is typed.

Important: Your BASIC module can execute and store up to 255 programs in EEPROM depending on the size of the programs and the capacity of the EEPROM. The programs are stored in a sequence string, referred to as the EEPROM file, in EEPROM for retrieval and execution.

When you enter ROM [integer], the BASIC module selects that program out of EEPROM memory and makes it the current program. If no integer is typed after the ROM command (example: `ROM`) the module defaults to ROM 1. Since the programs are stored in sequence in EEPROM, the integer following the ROM command selects the program the user wants to run or list. If you attempt to select a program that does not exist (example: you type to ROM 8 and only 6 programs are stored in the EEPROM) the message ERROR: PROM MODE is displayed.

The module does not transfer the program from EEPROM to RAM when the ROM mode is selected. If you attempt to alter a program in the ROM mode, by typing in a line number, the message ERROR: PROM MODE displays. The XFER command allows you to transfer a program from EEPROM to RAM for editing purposes. You do not get an error message if you attempt to edit a line of RAM program.

Important: When you transfer programs from EEPROM to RAM you lose the previous RAM contents.

Since the ROM command does NOT transfer a program to RAM, it is possible to have different programs in ROM and RAM simultaneously. You can move back and forth between the two modes when in command mode. If you are in run mode you can change back and forth using CALLS 70, 71 and 72. You can also use all of the RAM memory for variable storage if the program is stored in EEPROM. The system control value MTOP always refers to RAM. The system control value LEN refers to the currently selected program in RAM or ROM.

Syntax:

ROM[integer]

Example:

```
READY
>ROM1
```

RROM

Purpose:

Use the RROM command to tell the BASIC module interpreter to select the current program out of EEPROM or UVPROM and then execute the program . This command is equivalent to typing ROM and then RUN.

Important: Your BASIC module can execute and store up to 255 programs in EEPROM depending on the size of the programs and the capacity of the EEPROM. The programs are stored in a sequence string, referred to as the EEPROM file, in EEPROM for retrieval and execution.

When you enter RROM [integer], the BASIC module selects that program out of EEPROM memory, makes it the current program, and starts program execution. If no integer is typed after the RROM command (example: `RROM`) the module defaults to RROM 1. Since the programs are stored in sequence in EEPROM, the integer following the RROM command selects the program the user wants to run or list. If you attempt to select a program that does not exist (example: you type to RROM 8 and only 6 programs are stored in the EEPROM) the message ERROR: PROM MODE is displayed.

The module does not transfer the program from EEPROM to RAM when ROM mode is selected. If you attempt to alter a program in the ROM mode, by typing in a line number, the message ERROR: PROM MODE displays. The XFER command allows you to transfer a program from EEPROM to RAM for editing purposes. You do not get an error message if you attempt to edit a line of RAM program.

Important: When you transfer programs from EEPROM to RAM you lose the previous RAM contents.

Since the RROM command does NOT transfer a program to RAM, it is possible to have different programs in ROM and RAM simultaneously. You can move back and forth between the two modes when in command mode. If you are in run mode you can change back and forth using CALLS 70, 71 and 72. You can also use all of the RAM memory for variable storage if the program is stored in EEPROM. The system control value MTOP always refers to RAM. The system control value LEN refers to the currently selected program in RAM or ROM.

Syntax:

RROM[integer]

Example:

```
READY
>RROM
```

RUN

Purpose:

Use the RUN command to set all variables equal to zero, clear all BASIC evoked interrupts, and begin program execution with the first line number of the selected program. The RUN command and the GOTO statement are the only ways you can place the BASIC module interpreter into run mode from command mode. Terminate program execution at any time by typing a Control C on the console device.

Variations: Some BASIC interpreters allow a line number to follow the RUN command (example: `RUN 100`). The BASIC module does not permit this variation on the RUN command.

Execution begins with the first line number. To obtain a function similar to the RUN [ln num] command, use the GOTO[ln num] statement in the direct mode. See statement GOTO.

Syntax:

RUN

Example:

```
>1  REM EXAMPLE PROGRAM
>10 FOR I = 1 TO 3
>20 PRINT I
>30 NEXT I
>40 END
>RUN

  1
  2
  3

READY
>
```

SNGLSTP

Purpose:

Use the SNGLSTP command to initiate single-step program execution. If the number specified by this command is zero, single-step execution is disabled. If the number is not zero, a break point is set before each line in the program. Start the program by typing the RUN command. After each stop, type CONT to execute the next line. You can inspect variables or assign variables at each break point. SNGLSTP works only on programs executing from RAM. It does not stop a program executing from EEPROM.

Syntax:

SNGLSTP[integer]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 FOR I = 1 TO 5
>20 PRINT I
>30 NEXT I
>40 PRINT “PASSED FOR - NEXT LOOP”
>50 PRINT “THIS IS THE END”
>60 END

READY
>SNGLSTP 20

SINGLE STEP ENABLED

READY
>RUN

STOP - LINE 20
READY
>CONT

1

STOP - LINE 30
READY
>CONT

STOP - LINE 20
READY
>CONT

2

STOP - LINE 30
READY
>CONT
```

```
STOP - LINE 20
READY
>CONT

3

STOP - LINE 30
READY
>SNGLSTP 0

SINGLE STEP DISABLED

READY
>CONT

4
5
PASSED FOR - NEXT LOOP
THIS IS THE END

READY
>
```

VER

Purpose:

Use the VER command to print the BASIC module sign-on message that displays the current version of the firmware.

Syntax:

VER

Example:

```
>VER
SLC 500 BASIC Module-Catalog Number 1746-BAS
Firmware release: 1.00
Allen-Bradley Company, Copyright 1991
All rights reserved

>
```

XFER

Purpose:

Use the XFER command to transfer the current selected program in ROM to RAM and select RAM mode. After the XFER command executes, you can edit the program in the same way you edit any RAM program.

Important: The XFER command clears existing RAM programs.

Syntax:

XFER

Example:

```
READY
>XFER
```

Chapter 5

Command Line Calls

CALL 73 - Battery Backed RAM Disable

Purpose:

Use CALL 73 to disable the battery-backed RAM. When this CALL is executed, the message “Battery Backup Disabled” is printed on the host terminal. Disabling battery-backed RAM allows a purging reset. When power to the BASIC module is turned OFF, the contents of RAM are destroyed. When power is reapplied, RAM is cleared and battery back-up is re-enabled.

Syntax:

CALL 73

Example:

```
>CALL 73

Battery Backup Disabled

>REM TURNING POWER OFF, THEN BACK ON
```

CALL 74 - Battery Backed RAM Enable

Purpose:

Use CALL 74 to enable the battery-backed RAM. When this CALL is executed, the message “Battery Backup Enabled” is printed on the host terminal. Battery-backed RAM is enabled on BASIC module power-up and remains enabled until you execute a CALL 73 or until the battery fails.

Syntax:

CALL 74

Example:

```
>CALL 74

Battery Backup Enabled
```

CALL 77 - Protected Variable Storage

Purpose:

Important: Change CALL 77 from command mode only to ensure proper operation.

Use CALL 77 to reserve the top of RAM memory for protected variable storage. Values are saved if BATTERY-BACKUP is invoked. You store values with the ST@ command and retrieve them with the LD@ command. Each variable stored requires 6 bytes of storage space.

You must subtract 6 times the number of variables to be stored from MTOP reducing available RAM memory. This value is PUSHed onto the stack as the new MTOP address. All appropriate variable pointers are reconfigured. Do this only in command mode.

Important: Do not let the ST@ address write over the MTOP address. This could alter the value of a variable or string.

Syntax:

PUSH [new MTOP address]
CALL 77

Example: (For saving 2 variables)

```
>PRINT MTOP
24575
>PRINT MTOP–12
24563
>PUSH 24563:REM NEW MTOP ADDRESS
>CALL 77
```

```
>1  REM EXAMPLE PROGRAM
>10 K = 678*PI
>20 L = 520
>30 PUSH K
>40 ST@ 24575 : REM STORE K IN PROTECTED AREA
>50 PUSH L
>60 ST@ 24569 : REM STORE L IN PROTECTED AREA
>70 REM TO RETRIEVE PROTECTED VARIABLES
>80 LD@ 24575 : REM REMOVE K FROM PROTECTED AREA
>90 POP K
>100 LD@ 24569 : REM REMOVE L FROM PROTECTED AREA
>110 POP L
>120 REM USE LD@ AFTER POWER LOSS AND BATTERY BACK–UP IS USED
>
```

CALL 81 - User Memory Module Check and Description

Purpose:

Use CALL 81 to check the user memory module before burning a program into the memory module. This routine:

- determines the number of memory module programs
- determines the number of bytes left in the memory module
- determines the number of bytes in the RAM program
- prints a message indicating if enough space is available in the memory module for the RAM program
- checks memory module checksum if program is found
- prints a caution message is checksum fails

Important: CALL 81 cannot detect a defective memory module.

No PUSHes or POPs are needed.

Syntax:

CALL 81

Example:

```
>CALL 81

Number of BASIC programs in (E)EPROM.........  3
Available bytes to end of user (E)EPROM.......  7944
Available bytes to beginning of assembly pgm..  3848
Length of BASIC program in RAM...............  76


Program will fit in (E)EPROM.


READY
>
```

CALL 82 - Check User Memory Module Map

Purpose:

Use CALL 82 to check the user memory module and display a map of where all the BASIC programs are stored. The programmer can determine by using this call where the empty space in the memory module is located and how much space is available. No PUSHes or POPs are needed.

Syntax:

CALL 82

Example:

```
>CALL 82

 8010H --  805CH --> ROM 1
 805DH --  80A9H --> ROM 2
 80AAH --  80F6H --> ROM 3
 80F7H --  FFFFH --> UNUSED
>
```

CALL 101 - Upload User Memory Module Code to Host

Purpose:

Use CALL 101 to upload the code in the user memory module to the host terminal. This call requires two PUSHes and no POPs. The first PUSH is the starting address. The second PUSH is the ending address. This call converts data within the address range to Intel Hex format, then prints the information to the program port. An error message is printed if the addresses are not consistent.

Syntax:

PUSH [starting address]
PUSH [ending address]
CALL 101

Example:

```
>PUSH 8000 : PUSH 804FH : CALL 101

:108000003107021327CC3313276607005FFF473081
:108010005509000A8B41E034290D1000149C3130C1
:108020002C32302C33302C34300D0A001EA049EA9B
:1080300030A6330D0900289B41E049290D06003286
:1080400097490D0A003CA04AEA4FA6330D090046A5
:00000001F
>
```

CALL 103 - Print PRT1 Output Buffer and Pointer

Purpose:

Use CALL 103 to print the complete output buffer with address, front pointer, and number of characters in the buffer to the program port screen. No PUSHes or POPs are needed.

Use this information as a troubleshooting aid. It does not affect the contents of the buffer.

Syntax:

CALL 103

Example:

```
>CALL 103
                                PRT1 Output Queue

6D00H 3AH 31H 30H 38H 30H 34H 30H 30H 30H 39H 37H 34H 39H 30H 44H 30H
6D10H 41H 30H 30H 33H 43H 41H 30H 34H 41H 45H 41H 34H 46H 41H 36H 33H
6D20H 33H 30H 33H 30H 48H 20H 33H 33H 48H 20H 33H 30H 48H 20H 34H 38H
6D30H 48H 20H 32H 30H 48H 20H 33H 33H 48H 20H 33H 33H 48H 20H 34H 38H
6D40H 48H 20H 32H 30H 48H 20H 33H 33H 48H 20H 33H 30H 48H 20H 34H 38H
6D50H 48H 20H 32H 30H 48H 20H 33H 34H 48H 20H 33H 38H 48H 0DH 0AH 20H
6D60H 36H 44H 33H 30H 48H 20H 34H 38H 48H 20H 32H 30H 48H 20H 34H 38H
6D70H 48H 20H 32H 30H 48H 20H 33H 34H 48H 20H 33H 38H 48H 0DH 0AH 20H
6D80H 36H 44H 37H 30H 48H 20H 34H 38H 48H 20H 32H 30H 48H 20H 33H 32H
6D90H 48H 20H 33H 30H 48H 20H 34H 38H 48H 20H 32H 30H 48H 20H 33H 33H
6DA0H 48H 20H 33H 34H 48H 20H 34H 38H 48H 20H 32H 30H 48H 20H 33H 33H
6DB0H 48H 20H 33H 38H 48H 20H 34H 38H 48H 20H 32H 30H 48H 20H 33H 34H
6DC0H 48H 20H 33H 38H 48H 20H 34H 38H 48H 20H 32H 30H 48H 20H 33H 32H
6DD0H 48H 20H 33H 30H 48H 20H 34H 38H 48H 20H 32H 30H 48H 20H 33H 33H
6DE0H 48H 20H 33H 34H 48H 0DH 0AH 20H 36H 44H 43H 30H 48H 20H 34H 38H
6DF0H 48H 20H 32H 30H 48H 20H 33H 33H 48H 20H 33H 38H 48H 20H 34H 34H

Output queue front pointer is:  6D29H
```

CALL 104 - Print PRT1 Input Buffer and Pointer

Purpose:

Use CALL 104 to print the complete input buffer with address, front pointer, and number of characters in the buffer to the program port screen. No PUSHes or POPs are needed.

Use this information as a troubleshooting aid. It does not affect the contents of the buffer.

Syntax:

CALL 104

Example:

```
>CALL 104
                                PRT1 Input Queue

6C00H 33H 0DH 43H 41H 4CH 4CH 20H 31H 30H 34H 7FH 7FH 7FH 7FH 7FH 7FH
6C10H 7FH 7FH 52H 45H 4DH 20H 45H 58H 41H 4DH 50H 4CH 45H 53H 7FH 7FH
6C20H 7FH 7FH 7FH 7FH 7FH 7FH 7FH 7FH 7FH 7FH 0DH 0DH 0DH 0DH 0DH 0DH
6C30H 0DH 0DH 0DH 45H 58H 41H 4DH 7FH 7FH 7FH 7FH 52H 45H 4DH 20H 45H
6C40H 58H 41H 4DH 50H 4CH 45H 53H 20H 4FH 4EH 20H 50H 41H 47H 45H 20H
6C50H 36H 2DH 37H 0DH 43H 41H 4CH 4CH 20H 31H 30H 34H 0DH 52H 4DH 41H
6C60H 4EH 54H 20H 7FH 7FH 7FH 54H 20H 44H 41H 54H 41H 0DH 52H 45H 4DH
6C70H 20H 54H 48H 45H 52H 45H 20H 49H 53H 20H 4EH 4FH 20H 52H 45H 41H
6C80H 4CH 20H 52H 45H 53H 50H 4FH 4EH 53H 45H 20H 57H 48H 49H 43H 48H
6C90H 20H 57H 49H 4CH 4CH 20H 53H 48H 4FH 57H 20H 55H 50H 20H 49H 4EH
6CA0H 20H 41H 4EH 20H 45H 58H 41H 4DH 50H 4CH 45H 0DH 0DH 0DH 0DH 0DH
6CB0H 52H 45H 4DH 20H 45H 58H 41H 4DH 50H 4CH 45H 53H 20H 4FH 4EH 20H
6CC0H 50H 41H 47H 45H 20H 36H 2DH 36H 0DH 50H 55H 53H 48H 20H 38H 30H
6CD0H 30H 30H 48H 3AH 50H 7FH 7FH 20H 70H 55H 53H 7FH 7FH 7FH 3AH 20H
6CE0H 50H 55H 53H 48H 20H 38H 30H 7FH 30H 34H 46H 48H 20H 3AH 43H 41H
6CF0H 4CH 4CH 20H 31H 30H 31H 0DH 0DH 0DH 43H 41H 4CH 4CH 20H 31H 30H

Input queue front pointer is:  6C5DH
```

CALL 109 - Print Argument Stack

Purpose:

Use CALL 109 to print the top 9 values on the argument stack to the console. No PUSHes or POPs are needed. Use this information as a troubleshooting aid. It does not affect the contents of the argument stack or pointer to the stack.

Syntax:

CALL 109

Example:

```
>CALL 109

 1C9H 00H 00H 00H 00H 00H 00H
 1CFH 00H 00H 00H 00H 00H 00H
 1D5H 00H 00H 00H 00H 00H 00H
 1DBH 41H 67H 50H 00H 00H 7EH
 1E1H 83H 75H 00H 00H 00H 7CH
 1E7H 13H 04H 00H 00H 00H 7CH
 1EDH 32H 84H 70H 00H 00H 85H
 1F3H 32H 76H 80H 00H 00H 85H
 1F9H 00H 00H 00H 00H 00H 00H


Argument stack pointer is:  01FEH
```

CALL 110 - Print PRT2 Output Buffer Pointer

Purpose:

Use CALL 110 to print the complete buffer with addresses, front pointer and the number of characters in the buffer to the console device. No PUSHes or POPs are needed.

Use this information as a troubleshooting aid. It does not affect the contents of the buffer.

Syntax:

CALL 110

Example:

```
>CALL 110
                                PRT2 Output Queue

6F00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6F10H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6F20H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6F30H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6F40H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6F50H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6F60H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6F70H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6F80H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6F90H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6FA0H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6FB0H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6FC0H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6FD0H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6FE0H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6FF0H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H

Output queue front pointer is:  6F00H
```

CALL 111 - Print PRT2 Input Buffer Pointer

Purpose:

Use CALL 111 to print the complete buffer with addresses, front pointer and the number of characters in the buffer to the console device. No PUSHes or POPs are needed.

Use this information as a troubleshooting aid. It does not affect the contents of the buffer.

Syntax:

CALL 111

Example:

```
>CALL 111
                                PRT2 Input Queue

6E00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6E10H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6E20H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6E30H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6E40H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6E50H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6E60H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6E70H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6E80H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6E90H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6EA0H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6EB0H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6EC0H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6ED0H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6EE0H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
6EF0H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H

Input queue front pointer is:  6E00H
```

Chapter 6

Assignment Functions

CLEAR

Purpose:

Use the CLEAR statement to set all variables equal to 0 and reset all BASIC evoked interrupts and stacks. This means that after the CLEAR statement is executed, an ONTIME statement must be executed before the module acknowledges the internal timer interrupts. ERROR trapping with the ONERR statement also does not occur until an ONERR [ln num] statement is executed. The CLEAR statement does not affect the free running clock that is enabled by the CLOCK1 statement. CLEAR also does not reset the memory that has been allocated for strings, so it is not necessary to enter the STRING [expr], [expr] statement to re-allocate memory for strings after the CLEAR statement is executed. In general, CLEAR is used to “erase” all variables.

Syntax:

CLEAR

Example:

```
>CLEAR

>LIST
1  REM EXAMPLE PROGRAM
10 DIM A(4)
20 DATA 10,20,30,40
30 FOR I=0 TO 3
40 READ A(I)
50 NEXT I
60 FOR J=O TO 3
70 PRINT A(J)
80 NEXT J

READY
>PRINT A(1),I,J
 0  0  0

>RUN

 10
 20
 30
 40
```

```
READY
>PRINT A(1),I,J
 20  4  4

>CLEAR

>PRINT A(1),I,J
 0  0  0
```

CLEARI

Purpose:

Use the CLEARI statement to clear all of the BASIC evoked interrupts. The ONTIME interrupt is disabled after the CLEARI statement is executed. CLEARI does not affect the free running clock enabled by the CLOCK1 statement. You can use this statement to selectively DISABLE ONTIME interrupts during specific sections of your BASIC program. You must execute the ONTIME statement again before the specific interrupts are enabled.

Important: When the CLEARI statement is LISTed it appears as CLEARI.

Syntax:

CLEARI

Example:

```
READY
>CLEARI
```

CLEARS

Purpose:

Use the CLEARS statement to reset all of the stacks. The control, argument and internal stacks all reset to their initialization values. You can use this command to reset the stacks if an error occurs in a subroutine.

Important: When the CLEARS statement is LISTed it appears as CLEARS.

Syntax:

CLEARS

Example:

```
READY
>CLEARS
```

DATA

Purpose:

Use the DATA statement to specify the expressions that you can retrieve with a READ statement. If multiple expressions per line are used, you MUST separate them with a comma.

Every time a READ statement is encountered the next consecutive expression in the DATA statement is evaluated and assigned to the variable in the READ statement. You can place DATA statements anywhere within a program. They are not executed and do not cause an error. DATA statements are considered to be chained and appear to be one large DATA statement. If at anytime all the data is read and another READ statement is executed, the program terminates and the message ERROR: NO DATA – IN LINE XX prints to the console device. The module returns to command mode.

Syntax:

DATA

Example:

```
>LIST
1  REM EXAMPLE PROGRAM
10 DIM A(4)
20 DATA 10,ASC(A),ASC(C),35.627
30 FOR I=0 TO 3
40 READ A(I)
50 NEXT I
60 FOR J=O TO 3
70 PRINT A(J)
80 NEXT J


READY
>RUN

 10
 65
 67
 35.627
```

Important: You cannot use the CHR operator in a DATA statement.

DIM

Purpose:

Use the DIM statement to reserve storage for matrices. The storage area is first assumed to be zero. Matrices in the BASIC module may have only one dimension and the size of the dimensioned array may not exceed 254 elements.

Once a variable is dimensioned in a program it may not be re-dimensioned. An attempt to re-dimension an array causes an ARRAY SIZE ERROR that causes the module to enter the command mode.

If an array variable is used that was not dimensioned by a DIM statement, BASIC assigns a default value of 10 to the array size. All arrays are set equal to zero when the RUN command, NEW command or the CLEAR statement is executed.

The number of bytes allocated for an array is six times the array size plus one (6 * (array size + 1)). For example, the array A (100) requires 606 bytes of storage. Memory size usually limits the size of a dimensioned array.

Syntax:

DIM

Examples:

More than one variable can be dimensioned by a single DIM statement.

```
>1  REM EXAMPLE PROGRAM
>10 DIM A(25), C(15), A1(20)
```

Error on attempt to re-dimension array

```
>1  REM EXAMPLE PROGRAM
>10 A(5) = 10 : REM BASIC ASSIGNS DEFAULT OF 10 TO ARRAY A
>20 DIM A(5) : REM ARRAY RE–DIMENSION ERROR
>

READY
>RUN

ERROR: ARRAY SIZE – IN LINE  20

20     DIM A(5) :  REM ARRAY RE–DIMENSION ERROR
---------------X
READY
>
```

LET

Purpose:

Use the LET statement to assign a variable to the value of an expression.

Syntax:

LET [var] = [expr]

Examples:

```
>1  REM EXAMPLE PROGRAM
>10 LET A = 10*SIN(C)/100
```

```
>1  REM EXAMPLE PROGRAM
>10 LET A = A +1
```

Note that the = sign used in the LET statement is not an equality operator. It is a “replacement” operator. The statement should be read A is replaced by A plus one. The word LET is always optional (example: `LET A = 2` is the same as `A = 2`).

When LET is omitted the LET statement is called an IMPLIED LET. We use the word LET to refer to both the LET statement and the IMPLIED LET statement.

Also use the LET statement to assign string variables:

```
LET $(1)=“THIS IS A STRING”
```

or

```
LET $(2)=$(1)
```

Before you can assign strings you must execute the STRING [expr], [expr] statement or else a MEMORY ALLOCATION ERROR occurs that causes the module to enter the command mode. .

RESTORE

Purpose:

Use the RESTORE statement to “reset” the internal read pointer to the beginning of the data so that it may be read again.

Syntax:

RESTORE

Example:

```
>1  REM EXAMPLE PROGRAM
>10 FOR I = 1 TO 3
>20 READ A,C
>30 PRINT A,C
>40 NEXT I
>50 RESTORE
>60 READ A,C
>70 PRINT A,C
>80 DATA 10,20,10/2,20/2,SIN(PI),COS(PI)


READY
>RUN

 10         20
 5          10
 0          -1
 10         20
```

Chapter 7

Control Functions

CLOCK1

Purpose:

Use the CLOCK1 statement to enable the free running clock resident on the BASIC module. The special function operator TIME is incremented once every 5 milliseconds after the CLOCK1 statement is executed. The CLOCK1 statement uses an internal TIMER to generate an interrupt once every 5 milliseconds. Because of this, the special function operator TIME has a resolution of 5 milliseconds. The special function operator TIME counts from 0 to 65535.995 seconds. After reaching a count of 65535.995 seconds TIME overflows back to a count of zero. The interrupts associated with the CLOCK1 statement cause the module programs to run at about 99.6% of normal speed. This means that the interrupt handling for the free running clock uses about 0.4% of the total CPU time.

Important: This does not include additional overhead for ON-TIME user interrupt handling execution.

Syntax:

CLOCK1

Example:

```
>NEW

>1  REM EXAMPLE PROGRAM
>10 TIME = 0
>15 DBY(71) = 0
>20 CLOCK1
>30 ONTIME 2,100
>40 DO
>50 WHILE TIME < 10
>60 END
>100 PRINT “TIMER INTERRUPT AT – ”,TIME,“ SECONDS”
>110 ONTIME TIME+2,100
>120 RETI

READY
>RUN

TIMER INTERRUPT AT –  2.01  SECONDS
TIMER INTERRUPT AT –  4.015  SECONDS
TIMER INTERRUPT AT –  6.01  SECONDS
TIMER INTERRUPT AT –  8.01  SECONDS
TIMER INTERRUPT AT –  10.01  SECONDS
```

CLOCK0

Purpose:

Use the CLOCK0 (zero) statement to disable or "turn off" the free running clock resident on the BASIC module. After CLOCK0 is executed, the special function operator TIME no longer increments. CLOCK0 is the only module statement that can disable the free running clock. CLEAR and CLEARI do NOT disable the free running clock, only its associated ONTIME interrupt.

Important: CLOCK1 and CLOCK0 are independent of the clock/calendar.

Syntax:

CLOCK0

Example:

```
READY
>CLOCK0
```

DO-WHILE

Purpose:

Use the DO-WHILE statement to set up "loop control" within a module program. The operation of this statement is similar to the DO-UNTIL [rel expr]. All statements between the DO and the WHILE [rel expr] are executed as long as the relational expression following the WHILE statement is true. You can nest DO-WHILE statements.

The control stack (C-stack) stores all information associated with loop control (example: DO-WHILE, DO-UNTIL, FOR-NEXT and BASIC subroutines). The control stack is 157 bytes long. DO-WHILE and DO-UNTIL loops and GOSUB commands use 3 bytes of the control stack. FOR-NEXT loops use 17 bytes.

Important: Excessive nesting will exceed the limits of the control stack, generate an error, and cause the module to enter command mode.

Syntax:

DO-WHILE [rel expr]

Examples:

Simple DO-WHILE

```
>NEW

>1  REM EXAMPLE PROGRAM
>10 DO
>20 A = A + 1
>30 PRINT A
>40 WHILE A < 4
>50 PRINT “DONE”
>60 END

READY
>RUN

 1
 2
 3
 4
DONE

READY
>
```

Nested DO-WHILE

```
>NEW

>1  REM EXAMPLE PROGRAM
>10 A=0 : C=0
>20 DO
>30 A=A+1
>40 DO
>45 C=C+1
>50 PRINT A,C,A*C
>60 WHILE C<>3
>70 C=0
>80 WHILE A<4
>90 END

READY
>RUN

 1  1  1
 1  2  2
 1  3  3
 2  1  2
 2  2  4
 2  3  6
 3  1  3
 3  2  6
 3  3  9

READY
>
```

DO-UNTIL

Purpose:

Use the DO-UNTIL statement to set up “loop control” within a module program. All statements between the DO and the UNTIL[rel expr] are executed until the relational expression following the UNTIL statement is TRUE. You can nest DO-UNTIL loops.

The control stack (C-stack) stores all information associated with loop control (example: DO-WHILE, DO-UNTIL, FOR-NEXT and BASIC subroutines). The control stack is 157 bytes long. DO-WHILE and DO-UNTIL loops and GOSUB commands use 3 bytes of the control stack. FOR-NEXT loops use 17 bytes.

Important: Excessive nesting will exceed the limits of the control stack, generate an error, and cause the module to enter command mode.

Syntax:

DO-UNTIL [rel expr]

Examples:

Simple DO-UNTIL

```
>1  REM EXAMPLE PROGRAM
>10 A=0
>20 DO
>30 A=A+1
>40 PRINT A
>50 UNTIL A=4
>60 PRINT “DONE”
>70 END
>RUN
```

Nested DO-UNTIL

```
>1  REM EXAMPLE PROGRAM
>10 DO
>20 A=A+1
>30 DO
>40 C=C+1
>50 PRINT A,C,A*C
>60 UNTIL C=3
>70 C=0
>80 UNTIL A=3
>90 END
RUN
```

END

Purpose:

Use the END statement to terminate program execution. CONT does not operate if the END statement is used to terminate execution. A CAN’T CONTINUE ERROR prints to the console. Always include an END statement to properly terminate a program.

Syntax:

END

Example:

End Statement Termination

```
>1  REM EXAMPLE PROGRAM
>10 FOR I = 1 TO 4
>20 PRINT I,
>30 NEXT I
>40 END

READY
>RUN

 1  2  3  4
READY
>
```

FOR-TO-(STEP)-NEXT

Purpose:

Use the FOR – TO – (STEP) – NEXT statement to set up and control program loops.

The control stack (C-stack) stores all information associated with loop control (example: DO-WHILE, DO-UNTIL, FOR-NEXT and BASIC subroutines). The control stack is 157 bytes long. DO-WHILE and DO-UNTIL loops and GOSUB commands use 3 bytes of the control stack. FOR-NEXT loops use 17 bytes.

Important: Excessive nesting will exceed the limits of the control stack, generate an error, and cause the module to enter command mode.

Syntax:

FOR [expr] TO [expr] STEP [expr]
.
.
.
NEXT [expr]

Examples:

```
>1  REM EXAMPLE PROGRAM
>5  E=0 : C=10 : D=2
>10 FOR A=E TO C STEP D
>20 PRINT A
>30 NEXT A
>40 END
>RUN
```

```
0
2
4
6
8
10

READY

>1  REM EXAMPLE PROGRAM
>10 FOR I = 1 TO 4
>20 PRINT I,
>30 NEXT I
>40 END

READY
>RUN

 1  2  3  4
```

Since E=0, C=10, D=2, and the PRINT statement at line 20 executes 6 times, the values of "A" that are printed are 0, 2, 4, 6, 8 and 10. "A" represents the name of the index or loop counter. The value of "E" is the starting value of the index. The value of "C" is the limit value of the index and the value of "D" is the increment to the index.

If the STEP statement and the value "D" are omitted, the increment value defaults to 1, therefore; STEP is an optional statement. The NEXT statement returns the loop to the beginning of the loop and adds the value of "D" to the current index value. The current index value is then compared to the value of "C", the limit value of the index.

If the index is less than or equal to the limit, control transfers back to the statement after the FOR statement. Stepping "backward" (FOR I = 100 TO 1 STEP–1) is permitted in the BASIC module. The NEXT statement is always followed by the appropriate variable. You may nest FOR–NEXT loops up to 9 times.

```
>1  REM EXAMPLE PROGRAM
>10 FOR I=1 TO 4
>20 PRINT I,
>30 NEXT I
>40 END
>RUN
>1 2 3 4


READY
```

```
>1  REM EXAMPLE PROGRAM
>10 FOR I=0 TO 8 STEP 2
>20 PRINT I
>30 NEXT I
>40 END
>RUN
0
2
4
6
8

READY
```

GOTO

Purpose:

Use the GOTO statement to cause BASIC to transfer control to the line number ([ln num]) specified.

Syntax:

GOTO [ln num]

Example:

```
>1  REM EXAMPLE PROGRAM
>50 GOTO 100
```

If line 100 exists, this statement causes execution of the program to resume at line 100. If line number 100 does not exist the message ERROR: INVALID LINE NUMBER is printed to the console device and the BASIC module enters the command mode.

Unlike the RUN command, the GOTO statement, if executed in the COMMAND mode, does not clear the variable storage space or interrupts. However, if the GOTO statement is executed in the command mode after a line is edited, the module clears the variable storage space and all BASIC evoked interrupts.

IF-THEN-ELSE

Purpose:

Use the IF-THEN-ELSE statement to set up a conditional test.

Syntax:

IF [rel expr] THEN valid statement ELSE valid statement

Examples:

```
>1  REM EXAMPLE PROGRAM
>10 IF A =100 THEN A=0 ELSE A=A+1
```

Upon execution of line 10 IF A is equal to 100, THEN A is assigned a value of 0. IF A does not equal 100, A is assigned a value of A+1. If you want to transfer control to different line numbers using the IF statement, you may omit the GOTO statement. The following examples give the same results:

```
>20 IF INT(A)<10 THEN GOTO 100 ELSE GOTO 200
           or
>20 IF INT(A)<10 THEN 100 ELSE 200
```

You can replace the THEN statement with any valid BASIC module statement as shown below:

```
>30 IF A<>10 THEN PRINT A ELSE 10
>30 IF A<>10 PRINT A ELSE 10
```

You may execute multiple statements following the THEN or ELSE if you use a colon to separate them.

Example:

```
>30 IF A<>10 THEN PRINT A : GOTO 150 ELSE 10
>30 IF A<>10 PRINT A : GOTO 150 ELSE 10
```

In these examples, if A does not equal 10, then both PRINT A and GOTO 150 are executed. If A equals 10, then control passes to 10.

You may omit the ELSE statement. If you omit the ELSE statement control passes to the next statement.

Example:

```
>1  REM EXAMPLE PROGRAM
>20 IF A=10 THEN 40
>30 PRINT A
```

In this example, if A equals 10 then control passes to line number 40. If A does not equal 10, line number 30 is executed.

NEXT

Purpose:

Use the NEXT statement to return the FOR-TO-(STEP)-NEXT loop to the beginning of the loop and add the value of the index increment to the index. The current index value is then compared to the index limit to determine if another loop should be performed.

Syntax:

NEXT

Example:

```
>1  REM EXAMPLE PROGRAM
>5  E=0 : C=10 : D=2
>10 FOR A=E TO C STEP D
>20 PRINT A
>30 NEXT A
>40 END
>RUN

0
2
4
6
8
10

READY
>

>NEW

>1  REM EXAMPLE PROGRAM
>10 FOR I = 0 TO 8 STEP 2
>20 PRINT I
>30 NEXT I
>40 END

>RUN

 0
 2
 4
 6
 8
```

ON-GOTO

Purpose:

Use the ON-GOTO statement to transfer control to the line(s) specified by the GOTO statement when the value of the expression following the ON statement is encountered in the BASIC program.

Syntax:

ON [expr] GOTO [ln num]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 ON Q GOTO 100,200,300
```

Control is transferred to line 100 if Q is equal to 0 and then to line 200 if Q is equal to 1. If Q is equal to 2, control is transferred to line number 300, and so on. All comments that apply to GOTO apply to the ON statement. If Q is less than zero, a BAD ARGUMENT ERROR is generated and the BASIC module enters command mode. If Q is greater than the line number list following the GOTO statement, a BAD SYNTAX ERROR is generated. The ON-GOTO statement provides “conditional branching” options within the BASIC module program.

Chapter 8

Execution Control and Interrupt Support Functions

CALL 70 - ROM to RAM Program Transfer

Purpose:

Use CALL 70 to shift program execution from a running ROM program to the beginning of the RAM program. No arguments are PUSHed or POPped.

Important: The first line of the RAM program is not executed. We recommend that you make it a remark.

Syntax:

CALL 70

Example:

```
READY
>LIST
1   REM EXAMPLE PROGRAM
10  REM SAMPLE ROM PROGRAM FOR CALL 70
20  PRINT “NOW EXECUTING ROM #5”
30  CALL 70 :  REM GO EXECUTE RAM
40  END

READY
>RUN

NOW EXECUTING ROM #5
NOW EXECUTING RAM

READY
>LIST
1   REM EXAMPLE PROGRAM
10  REM SAMPLE RAM PROGRAM FOR CALL 70
20  PRINT “NOW EXECUTING RAM”
30  END

READY
```

CALL 71 - ROM/RAM to ROM Program Transfer

Purpose:

Use CALL 71 to transfer from a running ROM or RAM program to the beginning of any available ROM program. One argument is PUSHed (which ROM program). None are POPped. An invalid program error displays and you enter the command mode if the ROM number does not exist.

Important: The first line of the ROM program is not executed. We recommend that you make it a remark.

Syntax:

PUSH [ROM program number]
CALL 71

Example:

```
>1   REM EXAMPLE PROGRAM
>10  REM THIS ROUTINE WILL CALL AND EXECUTE A ROM ROUTINE
>20  INPUT “ENTER ROM ROUTINE TO EXECUTE”,N
>30  PUSH N
>40  CALL 71
>50  END


>RUN


ENTER ROM ROUTINE TO EXECUTE 4
```

The user is now executing ROM 4, if it exists. If the ROM routine requested does not exist the result is:

```
PROGRAM NOT FOUND.
READY
>
```

CALL 72 - RAM/ROM Return

Purpose:

Use CALL 72 to return to the ROM or RAM routine that called this ROM or RAM routine. Execution begins on the line following the line that CALLed the routine. No arguments are PUSHed or POPped. This routine works one layer deep. Program control reverts to the line following the CALL in the previous program.

Important: There must be a next line in the ROM or RAM routine, otherwise unpredictable events could occur that may destroy the contents of RAM. For this reason, always be sure that at least one END statement exists following a CALL 70 or 71.

Syntax:

CALL 72

Example:

Program in ROM #1:

```
>1   REM EXAMPLE PROGRAM
>10  REM SAMPLE PROG FOR CALL 72
>20  PRINT “NOW EXECUTING ROM #1”
>30  PUSH 3
>40  CALL 71 : REM EXECUTE ROM #3 THEN RETURN
>50  PRINT “EXECUTING ROM #1 AGAIN”
>60  END
```

Program in ROM #3:

```
>1   REM EXAMPLE PROGRAM
>10  PRINT “NOW EXECUTING ROM #3”
>20  CALL 72
>30  END
```

With ROM #1 selected:

```
>RUN

NOW EXECUTING ROM #1
NOW EXECUTING ROM #3
EXECUTING ROM #1 AGAIN

READY
>
```

GOSUB

Purpose:

Use the GOSUB statement to cause the BASIC module to transfer control of the program to the line number [ln num] following the GOSUB statement. In addition, the GOSUB statement saves the location of the statement following GOSUB on the control stack so that you can perform a RETURN statement to return control to the statement following the most recently executed GOSUB statement. You may nest the GOSUB statement up to 9 times.

The control stack (C-stack) stores all information associated with loop control (example: DO-WHILE, DO-UNTIL, FOR-NEXT and BASIC subroutines). The control stack is 157 bytes long. DO-WHILE and DO-UNTIL loops and GOSUB commands use 3 bytes of the control stack. FOR-NEXT loops use 17 bytes.

Important: Excessive nesting will exceed the limits of the control stack, generate an error, and cause the module to enter command mode.

Syntax:

GOSUB [ln num]

Example:

Simple Subroutine

```
READY
>1  REM EXAMPLE PROGRAM
>10 FOR I = 1 TO 5
>20 GOSUB 100
>30 NEXT I
>40 END
>100 PRINT I
>110 RETURN

READY
>RUN

 1
 2
 3
 4
 5

READY
>NEW
```

Nested Subroutine

```
>1  REM EXAMPLE PROGRAM
>10 FOR I = 1 TO 3
>20 GOSUB 100
>30 NEXT I
>40 END
>100 REM USER SUBROUTINE HERE
>105 PRINT I,
>110 GOSUB 200
>120 RETURN
>200 REM START OF NESTED SUBROUTINE
>210 PRINT I*I
 220 RETURN

READY
>RUN

 1  1
 2  4
 3  9

READY
>
```

ONERR

Purpose:

Use the ONERR statement to handle arithmetic errors, if they occur, during program execution. Only ARITH. OVERFLOW, ARITH. UNDERFLOW, DIVIDE BY ZERO and BAD ARGUMENT errors are “trapped” by the ONERR statement. All other errors are not “trapped” and cause the BASIC module to enter the command mode. If an arithmetic error occurs after the ONERR statement is executed, the module interpreter passes control to the line number [ln num] following the ONERR statement. You handle the error condition in a manner suitable to your application. The ONERR command does not trap bad data entered during an input instruction. This yields a “TRY AGAIN” message or “EXTRA IGNORED” message.

After the ONERR statement is executed, you can determine what type of error occurred by examining external memory location 257 (101H).

The error codes are:

- ERROR CODE = 10 – DIVIDE BY ZERO
- ERROR CODE = 20 – ARITH. OVERFLOW orARITH.UNDERFLOW

- ERROR CODE = 40 – BAD ARGUMENT

You can examine this location by using an XBY(257) statement.

Syntax:

ONERR [ln num]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 ONERR 500
>20 FOR I = 5 TO  0 STEP –1
>30 PRINT 1/I
>40 NEXT I
>50 END
>500 PRINT “ERROR CODE WAS ”,XBY(257)
>510 END

READY
>RUN

 .2
 .25
 .33333333
 .5
 1
ERROR CODE WAS  10

READY
>
```

A GOTO statement can replace the END statement in this example to provide a method of user programmed error recovery.

ON-GOSUB

Purpose:

Use the ON-GOSUB statement to transfer control to the line(s) specified by the GOSUB statement when the value of the expression following the ON statement is encountered in the BASIC program.

Syntax:

ON [expr] GOSUB [ln num], [ln num],...[ln num]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 ON Q GOSUB 100,200,300
```

If Q is equal to 0, control is transferred to line number 100. If Q is equal to 1, control is transferred to line number 200. If Q is equal to 2, control is transferred to line number 300, and so on. All comments that apply to GOSUB apply to the ON statement. If Q is less than zero a BAD ARGUMENT ERROR is generated. If Q is greater than the line number list following the GOSUB statement, a BAD SYNTAX ERROR is generated. The ON-GOSUB statement provides "conditional branching" options within the BASIC module program.

ONTIME

Purpose:

Use the ONTIME [expr], [ln num] statement to compensate for the incompatibility between the timer/counters on the microprocessor and the BASIC module. Your BASIC module can process a line in milliseconds while the timer/counters on the microprocessor operate in microseconds. The ONTIME statement generates an interrupt every time the special function operator, TIME, is equal to or greater than the expression following the ONTIME statement.

Only the integer portion of TIME is compared to the integer portion of the expression that gives you seconds. This comparison is performed at the end (CR or :) of each line of BASIC. The interrupt forces a GOSUB to the line number [ln num] following the expression [expr] in the ONTIME statement.

The ONTIME statement does not interrupt an input command or a CALL routine. Since the ONTIME statement uses the special function operator, TIME, you must execute the CLOCK1 statement for ONTIME to operate. If CLOCK1 is not executed the special function operator, TIME, does not increment.

Syntax:

ONTIME [expr], [ln num]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 TIME = 0
>15 DBY(71) = 0
>20 CLOCK1
>30 ONTIME 2,100
>40 DO
>50 WHILE TIME < 10
>60 CLOCK0
>70 END
>100 PRINT “TIMER INTERRUPT AT – ”,TIME “ SECONDS”
>110 ONTIME TIME+2,10
>120 RETI

READY
>RUN

TIMER INTERRUPT AT –  2.01  SECONDS
TIMER INTERRUPT AT –  4.005  SECONDS
TIMER INTERRUPT AT –  6.015  SECONDS
TIMER INTERRUPT AT –  8.01  SECONDS
TIMER INTERRUPT AT –  10.01  SECONDS
```

In the example above, the time printed out is .01 seconds later than the time that was supposed to be printed. This is caused by the terminal used in the example operating at 19200 baud which causes a .01 second delay in printing.

To execute the ONTIME interrupt at a fraction of a second use DBY(71) = X where X = 5 to 999 in 5 millisecond increments.

PUSH

Purpose:

Use the PUSH statement to place the arithmetic expression or expressions in the BASIC module argument stack. This statement evaluates the arithmetic expression, or expressions, following the PUSH statement and then places them in sequence on the argument stack.

This statement with the POP statement provides a simple means of passing parameters to CALL routines. In addition, the PUSH and POP statements are used to pass parameters to BASIC subroutines and to “SWAP” variables. The last value PUSHed onto the argument stack is the first value POPped off the argument stack.

You can push more than one expression onto the argument stack using a single PUSH statement with multiple expressions ([expr], [expr],...[expr]). Each expression must be followed by a comma. The last value PUSHed onto the argument stack is the last expression [expr] encountered in the push statement.

Important: The argument stack can hold up to 33 floating point numbers before overflowing.

Syntax:

PUSH [expr], [expr],.....[expr]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 A = 10
>20 C = 20
>30 PRINT “A = ”,A,“ AND C = ” C
>40 PUSH A,C
>50 POP A,C
>60 PRINT “A = ”,A,“  AND C = ”,C
>70 END

READY
>RUN

A =  10  AND C =  20
A =  20  AND C =  10

READY
>

>NEW

>1  REM EXAMPLE PROGRAM
>10 PUSH 0
>20 CALL 14
>30 POP W
>40 PRINT W
>50 END

READY
>RUN

 0

READY
>
```

POP

Purpose:

Use the POP statement to remove values from the BASIC module argument stack. The value at the top of the argument stack is assigned to the variable following the POP statement and the argument stack is "POPped" (example: incremented by 6). You can place values in the stack using either the PUSH statement.

Important: If a POP statement executes and no number is in the argument stack, an A–STACK ERROR occurs and the BASIC module enters command mode.

You can pop more than one variable off the argument stack using a single POP statement with multiple variables ([var], [var],...[var]). Each expression must be followed by a comma.

Syntax:

POP [var], [var],.....[var]

Example:

See PUSH statement above.

You can use the PUSH and POP statements to minimize GLOBAL variable problems. These are caused by the "main" program and all main program subroutines using the same variable names (example: GLOBAL VARIABLES). If you cannot use the same variables in a subroutine as in the main program, you can re-assign a number of variables (example: A=Q) before a GOSUB statement is executed.

If you reserve some variable names just for subroutines (S1, S2) and pass variables on the stack as shown in the previous example, you can avoid any GLOBAL variable problems in the BASIC module.

The PUSH and POP statements accept dimensioned variables A(4) and S1(12) as well as scalar variables. This is useful when large amounts of data must be PUSHed or POPped when using CALL routines.

Example:

```
>1  REM EXAMPLE PROGRAM
>40 FOR I=1 TO 64
>50 PUSH I
>60 CALL 10
>70 POP A(I)
>80 NEXT I
```

RETI

Purpose:

Use the RETI statement to exit from an ONTIME interrupt that is processed in a BASIC module program. The RETI statement functions the same as the RETURN statement except that it also clears a software interrupt flag so interrupts can again be acknowledged. If you fail to execute the RETI statement in the interrupt procedure, all future interrupts are ignored.

Syntax:

RETI

Example:

```
>1  REM EXAMPLE PROGRAM
>10 TIME=0 : CLOCK1 : ONTIME 2, 100 : DO
>20 WHILE TIME<10 : END
>100 PRINT “TIMER INTERRUPT AT -”, TIME,“ SECONDS”
>110 ONTIME TIME+2, 100 : RETI
>RUN

TIMER INTERRUPT AT - 2.045 SECONDS
TIMER INTERRUPT AT - 4.045 SECONDS
TIMER INTERRUPT AT - 6.045 SECONDS
TIMER INTERRUPT AT - 8.045 SECONDS
TIMER INTERRUPT AT - 10.045 SECONDS

READY
```

RETURN

Purpose:

Use the RETURN statement to “return” control to the statement following the most recently executed GOSUB STATEMENT. Use one return for each GOSUB to avoid overflowing the control–stack. This means that a subroutine called by the GOSUB statement can call another subroutine with another GOSUB statement.

Syntax:

RETURN

Examples:

Simple Subroutine

```
>1  REM EXAMPLE PROGRAM
>10 FOR I = 1 TO 5
>20 GOSUB 100
>30 NEXT I
>40 END
>100 PRINT I
>110 RETURN

READY
>RUN

 1
 2
 3
 4
 5

READY
>
```

Nested Subroutine

```
>1  REM EXAMPLE PROGRAM
>10 FOR I = 1 TO 5
>20 GOSUB 100
>30 NEXT I
>40 END
>100 PRINT I,
>110 GOSUB 200
>120 RETURN
>200 PRINT I*I,
>210 GOSUB 300
>220 RETURN
>300 PRINT I*I*I
>310 RETURN

READY
>RUN

 1  1  1
 2  4  8
 3  9  27
 4  16  64
 5  25  125

READY
>
```

STOP

Purpose:

Use the STOP statement to break program execution at specific points in a program. After a program is STOPped you can display or modify variables. You can resume program execution with a CONTinue command. The purpose of the STOP statement is to allow for easy program “debugging.”

Syntax:

STOP

Example:

```
1
STOP – IN LINE 40


>1  REM EXAMPLE PROGRAM
>10 FOR I = 1 TO 100
>20 PRINT I
>30 STOP
>40 NEXT I


READY
>RUN


 1
STOP – IN LINE  40
READY
>CONT


 2
STOP – IN LINE  40
READY
>CONT


 3
STOP – IN LINE  40
READY
>CONT


 4
STOP – IN LINE  40
READY
>
```

Note that the line number printed out after execution of the STOP statement is the line number following the STOP statement, not the line number that contains the STOP statement.

Chapter 9

Math and Backplane Conversion Functions

CALL 14 - 16 Bit Signed Integer to BASIC Floating-Point

Purpose:

Use CALL 14 to convert an SLC 500 controller 16 bit signed integer number to a BASIC floating-point number. The input argument is the address number (0 to 207) of the word in the BASIC module input buffer to be converted. The output argument is the converted value.

Syntax:

PUSH [word number of BASIC module input buffer]
CALL 14
POP [converted value]

Example:

```
>1  REM EXAMPLE PROGRAM
>20 PUSH 0 : REM CONVERT 1ST WORD OF BASIC INPUT BUFFER
>30 CALL 14 : REM DO 16-BIT SIGNED TO F.P. CONVERSION
>40 POP W : REM GET CONVERTED VALUE
>50 PRINT W
>RUN

 0

READY
>
```

CALL 15 - 16 Bit Unsigned Integer to BASIC Floating-Point

Purpose:

Use CALL 15 to convert an SLC 500 controller 16 bit unsigned integer number to a BASIC module floating-point number. The input argument is the address number (0 to 207) of the word in the BASIC module input buffer to be converted. The output argument is the converted value.

Syntax:

PUSH [word number of BASIC module input buffer]
CALL 15
POP [converted value]

Example:

```
>1  REM EXAMPLE PROGRAM
>50 PUSH 9 : REM CONVERT 10TH WORD OF BASIC INPUT BUFFER
>60 CALL 15 : REM DO 16–BIT UNSIGNED INTEGER TO F.P. CONVERSION
>70 POP W : REM GET CONVERTED VALUE
>80 PRINT W
>RUN

 0
```

CALL 24 - BASIC Floating-Point to 16 Bit Signed Integer

Purpose:

Use CALL 24 to convert a BASIC module floating-point number to a signed 16 bit integer and place the result in the BASIC module output buffer. The first value PUSHed is the data variable. The second value PUSHed is the address number (0 to 207) of the word in the BASIC module output buffer.

Important: If an attempt is made to write to word 200 of the BASIC output buffer, an error message is displayed and the module returns to command mode. The bits of word 200 are predefined.

The fractional part of the BASIC module floating-point value is truncated. If the BASIC module floating-point value is less than -32768, the value placed in the BASIC module output buffer is -32768. If the BASIC module floating-point value is greater than +32767, the value placed in the BASIC module output buffer is +32767. The programmer is responsible for checking the range of the number before conversion.

Syntax:

PUSH [value to be converted]
PUSH [word number of BASIC module output buffer]
CALL 24

Example:

```
>1  REM EXAMPLE PROGRAM
>10 W = 17
>40 PUSH W :REM THE VALUE TO BE CONVERTED
>50 PUSH 0 : REM 1ST WORD OF BASIC OUTPUT BUFFER
>60 CALL 24 : REM DO THE F.P. TO 16–BIT SIGNED CONVERSION

READY
>
```

CALL 25 - BASIC Floating-Point to 16 Bit Binary

Purpose:

Use CALL 25 to convert a BASIC module floating-point value between 0 and 65535 to its 16 bit binary representation. The resulting value is then stored in the BASIC module output buffer. The first value PUSHed is the data to be converted. The second value PUSHed is the address number (0 to 207) of the word in the BASIC module output buffer.

Important: If an attempt is made to write to word 200 of the BASIC output buffer, an error message is displayed and the module returns to command mode. The bits of word 200 are predefined.

The fractional part of the BASIC module floating-point value is truncated. If the BASIC module floating-point value is less than 0, then the value placed in the BASIC module output buffer is 0. If the value is greater than +65535, then the value placed in the BASIC module output buffer is +65535. The programmer is responsible for checking the range of the number before conversion.

Syntax:

PUSH [value to be converted]
PUSH [word number of BASIC module output buffer]
CALL 25

Example:

```
>1  REM EXAMPLE PROGRAM
>40 PUSH 9E+1 : REM THE VALUE TO BE CONVERTED
>50 PUSH 0 : REM 1ST WORD OF BASIC OUTPUT BUFFER
>60 CALL 25 : REM DO F.P. TO 16–BIT BINARY CONVERSION
>

READY
>
```

Chapter 10

Clock/Calendar Functions

CALL 40 - Set Clock/Calendar Time

Purpose:

Use CALL 40 to set the following clock/calendar time functions:

- H = hours (0 to 23; only a 24 hour clock is available)
- M = minutes (0 to 59)
- S = seconds (0 to 59)

Syntax:

PUSH [hours]
PUSH [minutes]
PUSH [seconds]
CALL 40

Example:

Program the wall clock for 1:35 pm (programmed as 13:35; only a 24 hour clock is available)

```
>1  REM EXAMPLE PROGRAM
>10 H = 13 : M = 35 : S = 00
>20 REM HOURS = 13, MINUTES = 35, SECONDS = 00
>30 PUSH H,M,S
>40 CALL 40

READY
>RUN

READY
>
```

CALL 41 - Set Clock/Calendar Date

Purpose:

Use CALL 41 to set the following clock/calendar functions:

- D = day
- M = month
- Y = year

Three values are PUSHed and none are POPped.

Syntax:

PUSH [day]
PUSH [month]
PUSH [year]
CALL 41

Example:

Program the clock/calendar for the 16th day of June 1991.

```
>1  REM EXAMPLE PROGRAM
>5  STRING 100,20
>10 H=13 : M=35 : S=00
>20 REM HOURS = 13, MINUTES = 35, SECONDS = 00
>30 PUSH H,M,S
>40 CALL 40
>60 D=16 : MO=6 : Y=91
>70 PUSH D,MO,Y
>80 CALL 41
>90 PUSH 3
>100 CALL 42
>120 PUSH 0
>130 CALL 43
>140 PRINT $(0)

READY
>RUN

16-JUN-91 13:35:00
```

CALL 42 - Set Day of Week

Purpose:

Use CALL 42 to set the day of the week. Sunday is day 1. Saturday is day 7.

Syntax:

PUSH [day of week]
CALL 42

Example:

```
>1  REM EXAMPLE PROGRAM
>10 PUSH 3: CALL 42:REM DAY IS TUESDAY.
```

CALL 43 - Retrieve Date/Time String

Purpose:

Use CALL 43 to return the current date and time as a string. PUSH the number of the string to receive the date/time (dd-mmm-yy HH:MM:SS). You must allocate a minimum of 18 characters for the string. This requires you to set the maximum length for all strings to at least 18 characters.

Syntax:

PUSH [string number]
CALL 43

Example:

```
>1  REM EXAMPLE PROGRAM
>10 STRING 100,20
>20 PUSH 1: CALL 43: REM PUT DATE/TIME IN STRING 1
>30 PRINT $(1)
>40 END

READY
>RUN

16-JUN-91 13:35:00

READY
>
```

CALL 52 - Retrieve Date String

Purpose:

Use CALL 52 to return the current date in a string (dd-mmm-yy). PUSH the number of the string to receive the date. You must allocate a minimum of 9 characters for the string.

Syntax:

PUSH [string number]
CALL 52

Example:

```
>1  REM EXAMPLE PROGRAM
>10 STRING 100,20
>20 PUSH 1 : CALL 52 : REM PUT DATE IN STRING 1
>30 PRINT $(1)
>40 END

READY
>RUN

16-JUN-91

READY
>
```

CALL 44 - Retrieve Date Numeric

Purpose:

Use CALL 44 to return the current date on the argument stack as three numbers. There is no input argument to this routine and three variables are returned. The date is POPped in day, month and year order.

Syntax:

CALL 44
POP [day]
POP [month]
POP [year]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 REM DATE RETRIEVE – NUMERIC EXAMPLE
>20 CALL 44 : REM INVOKE THE UTILITY ROUTINE
>30 POP D,M,Y : REM GET THE DATA FROM THE ARGUMENT STACK
>40 PRINT “CURRENT DATE IS ”,Y,M,D
>50 END

READY
>RUN

CURRENT DATE IS  91  6  19

READY

>
```

CALL 45 - Retrieve Time String

Purpose:

Use CALL 45 to return the current time in a string (HH:MM:SS). PUSH the number of the string to receive the time. You must allocate a minimum of 8 characters for the string.

Syntax:

PUSH [string number]
CALL 45

Example:

```
>1  REM EXAMPLE PROGRAM
>10 STRING 100,20
>20 PUSH 1 : CALL 45 : REM PUT TIME IN STRING 1
>30 PRINT $(1)
>40 END

READY
>RUN

06:40:49

READY
>
```

CALL 46 - Retrieve Time Numeric

Purpose:

Use CALL 46 to return the time of day in numeric form. Retrieve the time of day in numeric form by executing CALL 46 and POPping the three variables off of the argument stack on return. There are no input arguments. The time is POPped in hour, minute and second order.

Syntax:

CALL 46
POP [hour]
POP [minute]
POP [second]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 REM TIME IN VARIABLES EXAMPLE : REM GET THE WALL CLOCK TIME
>20 CALL 46
>30 POP H,M,S
>40 PRINT  “CURRENT TIME IS ”,H,M,S
>50 END

READY
>RUN

CURRENT TIME IS  6  43  7

READY
>
```

CALL 47 - Retrieve Day of Week String

Purpose:

Use CALL 47 to return the current day of week as a three character string. PUSH the number of the string to receive the day of week. You must allocate a minimum of 3 characters per string. Strings returned are SUN, MON, TUE, WED, THU, FRI, SAT.

Syntax:

PUSH [string number]
CALL 47

Example:

```
>1  REM EXAMPLE PROGRAM
>10 STRING 100,20
>20 PUSH 0 :CALL 47
>30 PRINT “TODAY IS ”,$(0)

READY
>RUN

TODAY IS FRI

READY
>
```

CALL 48 - Retrieve Day of Week Numeric

Purpose:

Use CALL 48 to return the current day of week on the argument stack as a number (example: `Sunday=1, Saturday=7`). This number can be POPped into a variable.

Syntax:

CALL 48
POP [day of week]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 REM DAY OF WEEK RETRIEVE – NUMERIC EXAMPLE
>20 CALL 48 : REM INVOKE UTILITY TO GET D.O.W.
>30 POP D
>40 PRINT D
>50 END

READY
>RUN

 5

READY
>
```

Chapter 11

Status Functions

CALL 36 - Get Number of Characters in PRT2 Buffers

Purpose:

Use CALL 36 to retrieve the number of characters in the chosen buffer of port PRT2.

- PUSH 1 for the input buffer
- PUSH 0 for the output buffer

You must PUSH the buffer to be examined. One POP is required to get the number of characters.

Syntax:

PUSH [buffer selection]
CALL 36
POP [number of characters]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 PUSH 0 : REM EXAMINES THE OUTPUT BUFFER
>20 CALL 36
>30 POP X : REM GET THE NUMBER OF CHARACTERS
>40 PRINT “NUMBER OF CHARACTERS IN OUTPUT BUFFER IS”,X
>50 END

READY
>RUN

NUMBER OF CHARACTERS IN OUTPUT BUFFER IS 0

READY
>
```

CALL 51 - Check CPU Output Image Buffer

Purpose:

Use CALL 51 to determine if the SLC 500 controller output image buffer located in the BASIC module has been updated since the last time it was checked. This routine has no input arguments and one output argument.

The output argument is equal to:

- "0" if the Logic Processor has not written to the output image buffer since the last time this CALL was executed or since the BASIC module was powered up, whichever occurred last
- "1" if the Logic Processor has written to the output image buffer since the last time this CALL was executed or since the BASIC module was powered up, whichever occurred last
- "2" if the Logic Processor does not support this capability (as with the SLC 5/01 processor)

Syntax:

CALL 51
POP [output image buffer status]

Example:

```
>1   REM EXAMPLE PROGRAM
>120 CALL 51 : REM WAIT ON SLC
>130 POP S
>140 IF (S = 2) THEN PRINT "THE SLC DOES NOT SUPPORT THIS FUNCTION"
>150 IF (S = 2) THEN STOP
>160 IF (S = 0) THEN GOTO 120
>170 PRINT "OUTPUT IMAGE HAS BEEN UPDATED"

READY
>RUN

THE SLC DOES NOT SUPPORT THIS FUNCTION
STOP – IN LINE  160
READY
```

CALL 55 - Check CPU Input Image Buffer

Purpose:

Use CALL 55 to determine if the CPU INPUT IMAGE BUFFER located in the BASIC module has been read by the Logic Processor since the last time it was checked. This routine has no input arguments and one output argument.

The output argument is equal to:

- "0" if the Logic Processor has not read from the CPU INPUT IMAGE BUFFER since the last time the CALL was executed or since the BASIC module was powered up, whichever occurred last
- "1" if the Logic Processor has read from the CPU INPUT IMAGE BUFFER since the last time this CALL was executed or since the BASIC module was powered up, whichever occurred last
- "2" if a Logic Processor does not support this capability (as with the SLC 5/01 processor)

Syntax:

CALL 55
POP [input image buffer status]

Example:

```
>1   REM EXAMPLE PROGRAM
>120 CALL 55 :  REM WAIT ON SLC
>130 POP S
>140 IF (S=2) THEN  PRINT "THE SLC DOES NOT SUPPORT THIS FUNCTION"
>150 IF (S=2) THEN  STOP
>160 IF (S=0) THEN  GOTO 120
>170 PRINT "INPUT IMAGE HAS BEEN READ"

READY
>RUN

INPUT IMAGE HAS BEEN READ

READY
>
```

CALL 58 - Check M0 File

Purpose:

Use CALL 58 to determine if the Module File M0 located in the BASIC module has been updated since the last time it was checked. This routine has no input argument and one output argument.

The output argument is equal to:

- “0” if the Logic Processor has not written to the Module File M0 since the last time this CALL was executed or since the BASIC module was powered up
- “1” if the Logic Processor has written to the Module File M0 since the last time this CALL was executed or since the BASIC module was powered up, whichever occurred last
- “2” if the Logic Processor does not support this capability (as with the SLC 5/01 processor)

Syntax:

CALL 58
POP[module file M0 write status]

Example:

```
>1   REM EXAMPLE PROGRAM
>120 CALL 58 : REM START WAITING ON M0 UPDATE
>130 POP S
>140 IF (S = 2) THEN PRINT “PROCESSOR DOES NOT SUPPORT THIS FUNCTION”
>150 IF (S = 2) THEN STOP
>160 IF (S = 0) THEN GOTO 120
>170 PUSH 64
>180 CALL 56
>190 POP A
>200 PRINT “CALL 56 OUTPUT IS ”,A


READY
>RUN

CALL 56 OUTPUT IS  0
```

CALL 59 - Check M1 File

Purpose:

Use CALL 59 to determine if the Module File M1 located in the BASIC module has been read by the Logic Processor since the last time it was checked. This routine has no input arguments and one output argument.

The output argument is equal to:

- "0" if the Logic Processor has not read from the Module File M1 since the last time this CALL was executed or since the BASIC module was powered up, whichever occurred last.
- "1" if the Logic Processor has read from the Module File M1 since the last time this CALL was executed or since the BASIC module was powered up, whichever occurred last.
- "2" if the Logic Processor does not support this capability (as with the SLC 5/01 processor)

Syntax:

CALL 59
POP [module file M1 read status]

Example:

```
>1   REM EXAMPLE PROGRAM
>100 PUSH 64
>110 CALL 56 :  REM COPY BASIC OUTPUT BUFFER TO M1
>120 POP A
>130 IF (A=2) THEN  PRINT "SLC DOES NOT SUPPORT THIS FUNCTION"
>140 IF (A=2) THEN  STOP
>150 IF (A<>0)THE GOTO 110
>160 CALL 59 :  REM START WAITING NOW
>170 POP S
>180 IF (S=2) THEN  PRINT "SLC DOES NOT SUPPORT THIS FUNCTION"
>190 IF (S=2)THE STOP
>200 IF (S=0) THEN  GOTO 170
>210 PRINT "CALL 59 OUTPUT IS ",S

READY
>RUN

CALL 59 OUTPUT IS  1
```

CALL 75 - Check SLC 500 Controller CPU Status

Purpose:

Use CALL 75 to check the mode (Run/Program/Test) of the SLC 500 controller CPU. No PUSHes are required. One POP is required.

In SLC 5/01 mode of operation, the POPped values are:

- 0 = SLC processor in RUN mode
- 1 = SLC processor not in RUN mode

In SLC 5/02 mode of operation, the POPped values are:

- 0 = SLC processor in RUN mode
- 1 = SLC processor in program mode
- 2 = SLC processor in TEST mode

Syntax:

CALL 75
POP [processor mode]

Example:

```
>1   REM EXAMPLE PROGRAM
>100 CALL 75
>110 POP S
>120 IF (S=0) THEN PRINT “SLC IS IN RUN MODE”
>130 IF (S=1) THEN PRINT “SLC IS NOT IN RUN MODE”
>140 IS (S=2) THEN PRINT “SLC IS IN TEST MODE”

READY
>RUN

SLC IS IN RUN MODE

READY
>
```

CALL 80 - Check Battery Condition

Purpose:

Use CALL 80 to check the BASIC module battery condition. If a 0 is POPped after a CALL 80, battery is okay. If a 1 is POPped a low battery condition exists.

Syntax:

CALL 80
POP [battery status]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 CALL 80
>20 POP C
>30 IF (C<>0) THEN PRINT ”BATTERY LOW!”
>40 END

READY
>RUN

BATTERY LOW!
```

CALL 86 - Check DH485 Interface File Remote Write Status

Purpose:

Use CALL 86 to determine if the DH485 Common Interface File located in the BASIC module has been updated since the last time it was checked. This routine has no input arguments and one output argument.

The output argument is equal to:

- “0” if a device on the DH485 Serial Communications Link has not written to the DH485 Serial Common Interface File since the last time this CALL was executed or since the BASIC module was powered up, whichever occurred last
- “1” if a device on the DH485 Serial Communications Link has written to the DH485 Common Interface File since the last time this CALL was executed or since the BASIC module was powered up, whichever occurred last

Syntax:

CALL 86
POP [DH-485 interface file remote write status]

Example:

```
>1   REM EXAMPLE PROGRAM
>100 CALL 86 : REM CHECK FILE STATUS
>110 POP X : REM GET THE STATUS
>120 IF(X<>1) THEN GOTO 100 : REM WAIT ON THE DATA

READY
>
```

CALL 87 - Check DH485 Interface File Remote Read Status

Purpose:

Use CALL 87 to determine if the DH485 Common Interface File located in the BASIC module has been read by a device on the DH485 Serial Communications Link since the last time it was checked. This routine has no input argument and one output argument.

The output argument is equal to:

- "0" if a device has not read from the DH485 Common Interface File since the last time this CALL was executed or since the BASIC module was powered up, whichever occurred last
- "1" if a device on the DH485 Serial Communications Link has read the DH485 Common Interface File since this CALL was executed or since the BASIC module was powered up, whichever occurred last

Syntax:

CALL 87
POP [DH-485 interface file remote read status]

Example:

```
>1   REM EXAMPLE PROGRAM
>100 CALL 87 : REM CHECK FILE STATUS
>110 POP X : REM GET THE STATUS
>120 IF (X<>1) GOTO 100: REM WAIT ON DATA TO BE READ

READY
>
```

CALL 95 - Get Number of Characters in PRT1 Buffers

Purpose:

Use CALL 95 to retrieve the number of characters in the chosen buffer of port PRT1.

- PUSH 1 for the input buffer
- PUSH 0 for the output buffer

You must PUSH which buffer is to be examined. One POP is required to get the number of characters.

Syntax:

PUSH [buffer selection]
CALL 95
POP [number of characters]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 PUSH 0 : REM EXAMINES THE OUTPUT BUFFER
>20 CALL 95
>30 POP X : REM GET THE NUMBER OF CHARACTERS
>40 PRINT “NUMBER OF CHARACTERS IN PRT1 OUTPUT BUFFER IS ”,X
>50 END

READY
>RUN

NUMBER OF CHARACTERS IN PRT1 OUTPUT BUFFER IS  0

READY
>
```

CALL 97 - Enable Port PRT2 DTR Signal

Purpose:

Use CALL 97 to enable the Data Terminal Ready (DTR) signal from port PRT2. The DTR signal on PTR2 is enabled by default on power-up. This call will re-enable the DTR signal if it has been disabled by CALL 98.

Syntax:

CALL 97

Example:

```
>10 REM EXAMPLE PROGRAM
>20 CALL 97 : REM ENABLE DTR SIGNAL

READY
>
```

CALL 98 - Disable Port PRT2 DTR Signal

Purpose:

Use CALL 98 to disable the Data Terminal Ready (DTR) signal from port PRT2.

Syntax:

CALL 98

Example:

```
>10 REM EXAMPLE PROGRAM
>20 CALL 98 : REM DISABLE DTR SIGNAL

READY
>
```

CALL 108 - Enable DF1 Driver Communications

Purpose:

Use CALL 108 to enable DF1 driver communications via port PRT2.

Important: DF1 can only be enabled if jumper JW4 on the BASIC module is in the correct position. Refer to the SLC 500 BASIC Module Design and Integration Manual (Catalog Number 1746-ND005) for additional information.

This routine has six input arguments and no output arguments. The first input argument specifies the operational code selection that indicates the mode of operation for the DF1 driver. The operational code specifies the following DF1 parameters:

- full-duplex or half-duplex slave operation
- duplicate packet detection (DPD) selection
- BCC or CRC error checking selection
- enable embedded responses (ER) or auto-detect embedded responses (ADER). If ADER is selected, enabled responses will not be transmitted until an embedded response is received. It is assumed that if the device being communicated with can send an ER it can also receive them. (Only applies to full-duplex operation)
- modem handshaking selection:
  - half-duplex options:

    No HandShaking (NHS)
    Half-Duplex Modem without Continuous Carrier (HDMwoCC)
    Half-Duplex Modem with Continuous Carrier (HDMwCC)

  - full-duplex options:

    No HandShaking (NHS)
    Full-Duplex Modem (FDM)

Legal values for the operational code are 0 to 11 for half-duplex mode and 16 to 31 for full-duplex mode. Table 11.A lists the legal values for the half-duplex operational codes and their corresponding mode of operation. Table 11.B lists the legal values for the full-duplex operational codes and their corresponding mode of operation.

A special range of operational codes (32 -43) are also accepted. These codes are identical to codes 0 -11 except that the end of transmission (EOT) packets are suppressed. This operation is a deviation from the standard DF1 protocol and should only be used where transmissions from a slave module need to be minimized. When using one of these selections, the DF1 driver will not respond to ENQuires from a DF1 master unless there is a data packet to be transmitted.

Important: Other port parameters, such as baud rate, number of stop bits, and parity are selected using the MODE command before DF1 is enabled. The modem handshaking selection made here overrides the handshaking parameter of the MODE command until DF1 is disabled.

Table 11.A
DF1 Half-duplex Operational Codes

Operational Code	Corresponding Mode of Operation	Special Operational Code (Same as 0 - 11 except EOT is suppressed)
0	NHS, Disable DPD, BCC Error Checking	32
1	NHS, Enable DPD, BCC Error Checking	33
2	NHS, Disable DPD, CRC Error Checking	34
3	NHS, Enable DPD, CRC Error Checking	35
4	HDMwoCC, Disable DPD, BCC Error Checking	36
5	HDMwoCC, Enable DPD, BCC Error Checking	37
6	HDMwoCC, Disable DPD, CRC Error Checking	38
7	HDMwoCC, Enable DPD, CRC Error Checking	39
8	HDMwCC, Disable DPD, BCC Error Checking	40
9	HDMwCC, Enable DPD, BCC Error Checking	41
10	HDMwCC, Disable DPD, CRC Error Checking	42
11	HDMwCC, Enable DPD, CRC Error Checking	43

Table 11.B
DF1 Full-duplex Operational Codes

Operational Code	Corresponding Mode of Operation
16	NHS, ER, Disable DPD, BCC Error Checking
17	NHS, ER, Enable DPD, BCC Error Checking
18	NHS, ER, Disable DPD, CRC Error Checking
19	NHS, ER, Enable DPD, CRC Error Checking
20	NHS, ADER, Disable DPD, BCC Error Checking
21	NHS, ADER, Enable DPD, BCC Error Checking
22	NHS, ADER, Disable DPD, CRC Error Checking
23	NHS, ADER, Enable DPD, CRC Error Checking
24	FDM, ER, Disable DPD, BCC Error Checking
25	FDM, ER, Enable DPD, BCC Error Checking
26	FDM, ER, Disable DPD, CRC Error Checking
27	FDM, ER, Enable DPD, CRC Error Checking
28	FDM, ADER, Disable DPD, BCC Error Checking
29	FDM, ADER, Enable DPD, BCC Error Checking
30	FDM, ADER, Disable DPD, CRC Error Checking
31	FDM, ADER, Enable DPD, CRC Error Checking

Half-Duplex no handshaking modem control, selected by operational codes 0 through 3, has the following characteristics:

- The RTS output line will be activated during transmission, but no RTS On Delay or RTS Off Delay will be performed
- The DTR output line will not be manipulated by the DF1 driver. It is recommended that you activate DTR in your BASIC program while DF1 communications are taking place
- The CTS and DSR input lines will NOT be monitored or have any affect on transmissions or receptions
- A transmission monitor guarantees that transmitter interrupts are being generated in a timely manner. If a time-out occurs, the DF1_Status will be set to code value “5” if a data packet was being transmitted. Also, RTS will immediately be dropped when this time-out occurs.

Half-Duplex without continuous carrier modem control, selected by operational codes 4 through 7, has the following characteristics:

Important: For proper operation, the Data Carrier Detect (DCD) line from the modem must be connected to the DSR input of port PRT2.

- The RTS output line will be activated only during transmissions. The actual packet transmission will start after the delay specified by the RTS On Delay parameter, assuming the CTS input is active by then. When the transmission is complete and the delay time period specified by the RTS Off Delay parameter has timed out, RTS is deactivated.

- An actual transmission will not start until the CTS input is active. A transmission guarantees that transmitter interrupts are being generated in a timely manner. If a time out occurs, then the DF1_Status will be set to code value “5” if the data packet was being transmitted. RTS will be dropped immediately when this occurs.

- If not already active, the DTR line will be raised when the DF1 Driver is enabled. Even after the DF1 Driver is disabled, it WILL remain active; the user may deactivate it via a Basic Call.

- Characters that are received will only be accepted if the DCD line is active. A packet reception will be aborted if DCD goes inactive during the byte-to-byte reception of that packet.

  There will be no constant monitoring of DCD even between packets as there will be with the constant carrier selection. Therefore, the DTR line will never be deactivated.

Half-Duplex with continuous carrier modem control, selected by operational codes 8 through 11, has the following characteristics:

Important: For proper operation, the Data Carrier Detect (DCD) line from the modem must be connected to the DSR input of port PRT2.

- The RTS output line will be activated only during transmissions. The actual packet transmission will start after the delay specified by the RTS On Delay parameter, assuming the CTS input is active by then. When the transmission is complete and the delay time period specified by the RTS Off Delay parameter has timed out, RTS is deactivated.

- An actual transmission will not start until the CTS input is active. A transmission guarantees that transmitter interrupts are being generated in a timely manner. If a time out occurs, then the DF1_Status will be set to code value “5” if the data packet was being transmitted. RTS will be dropped immediately when this occurs.

- If not already active, the DTR line will be raised when the DF1 Driver is enabled. It will be dropped only when DCD is lost as described in the next paragraph. Even after the DF1 Driver is disabled or remains active, the user may deactivate it via a Basic CALL.

- For packet reception, the DCD signal will be monitored (via the DSR input line). If DCD is not already active when the DF1 Driver is enabled, then it is immediately detected when it does go active. At this point, the DCD is checked every 5 mS to make sure it remains active. If DCD goes inactive, the driver will wait 10 seconds for it to go active again. If DCD does not go active again in this amount of time, then the DTR output line will be dropped for a period of time ranging from 5 to 10 mS in length.

  Also, characters that are received will only be accepted if the DCD line is active. A packet reception will be aborted if DCD goes inactive during the byte-to-byte reception of a packet.

Full-Duplex with no handshaking, selected by operation codes 16 through 23, has the following characteristics:

- The RTS output line will be activated when the DF1 Driver is enabled and will remain so until the DF1 Driver is disabled.

- The DTR output line will not be manipulated by the DF1 Driver. It is recommended that the user activate DTR in his/her Basic program while the DF1 communications is taking place.

- The CTS and DSR input lines will NOT be monitored or have any effect on transmissions.

- A transmission monitor guarantees that transmitter interrupts are being generated in a timely manner. If a time-out occurs, then the DF1_Status will be sent to code value “5” if the packet in the process of being transmitted was a data packet. RTS will NOT be deactivated when this time-out occurs.

Full-Duplex Modem, selected by operation codes 24 through 31, has the following characteristics:

- The RTS output line will be activated when the DF1 Driver is enabled and will remain so until the DF1 Driver is disabled.
- An actual transmission will not start until the CTS input is active. A transmission monitor guarantees that transmitter interrupts are being generated in a timely manner. If a time-out occurs, then the DF1_Status will be sent to code value “5” if the packet in the progress of being transmitted was a data packet. RTS will NOT be deactivated when this occurs.
- If DTR is not already active when the DF1 Driver is enabled, it is immediately activated. It will become active only if DCD is lost as described in the next paragraph. Even after the DF1 Driver is disabled, DTR remains active; the user may deactivate it via a Basic CALL.
- For packet receptions, the DCD signal will be monitored via the DSR input line. If DCD is not already active when the DF1 Driver is enabled, then it will immediately be detected when it does go active. At this point, DCD is checked every 5 mS to make sure it remains active. If it goes inactive, the driver will wait 10 seconds for DCD to go active again. If DCD does not go active again in this amount of time, then the DTR output line will be deactivated for a period of time ranging from 5 to 10 mS in length.

  Also, characters that are received will only be accepted if the DCD line is active. A packet reception is aborted if DCD goes inactive during the byte-to-byte reception of that packet.

The second input argument specifies the Poll Time-out period when in half-duplex mode or the ACKnowledge Time-out period when in full-duplex mode. Poll Time-out specifies in 5 msec. increments how long to wait to be polled by the DF1 master before a transmission request is ignored. PUSHing 0 indicates no Poll Time-out period. ACKnowledge Time-out specifies in 5 msec. increments how long to wait for an ACK/NAK before transmitting an ENQuiry. The valid range for the ACKnowledge Time-out is 2 to 65535.

The third input argument specifies the number of message retries when in half-duplex mode or the number of ENQuiry Retries to perform when in full-duplex mode. Message retries specifies the number of message transmission retry attempts to be made before giving up and flagging the transmission as failed. PUSHing 0 indicates only the initial attempt will be made and if not acknowledged by the master the attempt is flagged as failed. ENQuiry Retries specifies the number of ENQ's to transmit before a packet transmission is flagged as failed. The valid range for both is 0 to 254.

The fourth input argument specifies the RTS On Delay time period when in half-duplex mode or the number of NAK Received Retries to perform when in full-duplex mode. RTS On Delay specifies in 5 msec. increments the delay between when a Request-To-Send (RTS) is activated and a transmission is initiated. Only used if HDMwCC or HDMwoCC is selected through the first input argument. The valid range for the RTS On Delay is 0 to 65535. NAK Received Retries specifies the number of packet retries to transmit due to receiving NAK responses. The valid range for NAK Received Retries is 0 to 254.

The fifth input argument specifies the RTS Off Delay time period. RTS Off Delay specifies in 5 msec. increments the delay between when a transmission is completed and a Request-To-Send (RTS) is deactivated. The valid range for the RTS Off Delay is 0 to 65499. This argument is only used if HDMwCC or HDMwoCC is selected through the first input argument. This input argument is only used for half-duplex mode. When full-duplex mode is selected a NULL value must be PUSHed.

The sixth input argument specifies slave address that the DF1 driver responds to when receiving enquires from a DF1 master. Legal values are 0 to 254. This input argument is only used for half-duplex mode. When full-duplex mode is selected a NULL value must be PUSHed.

Syntax:

PUSH [operational code]
PUSH [poll time-out or ACKnowledge time-out]
PUSH [message retries or ENQuiry retries]
PUSH [RTS On delay or NAK received retries]
PUSH [RTS Off delay or NULL value]
PUSH [slave address or NULL value]
CALL 108

Example:

```
>1  REM EXAMPLE PROGRAM
>10 PUSH 5 : REM HDMWOCC, ENABLE DPD, BCC ERROR CHECKING
>20 PUSH 200 : REM WAIT 1 SECOND TO BE POLLED BY MASTER
>30 PUSH 2 : REM PERFORM 2 RETRIES
>40 PUSH 4 : REM 20 MS RTS ON DELAY
>50 PUSH 4 : REM 20 MS RTS OFF DELAY
>60 PUSH 10 : REM SLAVE ADDRESS OF 10
>70 CALL 108
>80 END
```

CALL 113 - Disable DF1 Driver Communications

Purpose:

Use CALL 113 to disable DF1 driver communications. This routine has no input arguments and no output arguments. This call terminates DF1 communication immediately, even if the serial transmission of a data packet is in progress. You should write your user program so that it completes any transmission before performing CALL 113. This call clears the PRT2 transmission and receive buffers.

Syntax:

CALL 113

Example:

```
>1  REM EXAMPLE PROGRAM
>10 CALL 113
>20 END
```

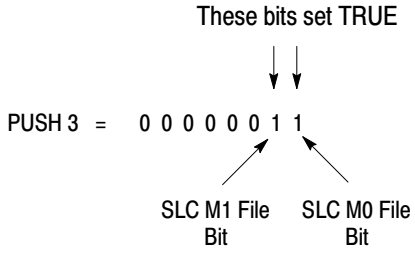
CALL 120 - Clear BASIC Module Input and Output Buffers

Purpose:

Use CALL 120 to clear the BASIC module input and output buffers. This routine has one input argument and no output arguments. The input argument is an 8-bit word that corresponds to the BASIC module input and output buffers, as shown below:

Bit	Decimal Equivalent	BASIC Module Input and Output Buffer Areas
0	1	SLC M0 File
1	2	SLC M1 File
2	4	SLC Output Image Table
3	8	SLC Input Image Table
4	16	Common Interface Input File
5	32	Common Interface Output File
6		Not Used
7		Not Used

You must PUSH the decimal equivalent of the areas of the input and output buffers that you want to clear. For example to clear the SLC M0 and SLC M1 files you would PUSH the value “3”. The BASIC module sets bits 0 and 1 true and clears the SLC M0 and M1 file areas of the input and output buffers.



Syntax:

PUSH [decimal equivalent]
CALL 120

Example:

```
>1  REM EXAMPLE PROGRAM
>10 PUSH 3 : REM CLEAR SLC MO AND M1 FILES
>20 CALL 120
>30 END
```

CALL 121 - Get SLC Processor Program ID Number

Purpose:

Use CALL 121 to get the ID number of the active SLC processor program. This routine has no input arguments and one output argument. The output argument is an integer value between 0 and 65536 that corresponds to the program ID number of the active program on the SLC processor.

Syntax:

CALL 121
POP [program ID number]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 CALL 121
>20 POP X : REM GET SLC PROCESSOR PROGRAM ID
>30 END
```

Chapter 12

Output Functions

CALL 31 - Display Current PRT2 Port Setup

Purpose:

Use CALL 31 to display the current PRT2 port configuration on the program port terminal screen. No arguments are PUSHed or POPped.

Syntax:

CALL 31

Example:

```
>CALL 31

1200 Baud
Hardware Handshaking OFF
1 Stop Bit(s)
No Parity
8 Bits/Char
Xon/Xoff

READY
>
```

CALL 37 - Clear PRT2 Input/Output Buffers

Purpose:

Use CALL 37 to clear the peripheral port input and/or output buffers. Use the following PUSHes to clear the corresponding buffer:

- PUSH 0 to clear the output buffer.
- PUSH 1 to clear the input buffer.
- PUSH 2 to clear both buffers.

Syntax:

PUSH [buffer selection]
CALL 37

Example:

```
>1  REM EXAMPLE PROGRAM
>10 PUSH 0 : REM CLEARS THE OUTPUT BUFFER
>20 CALL 37
>30 END

READY
>RUN

READY
>
```

CALL 54 - Transfer BASIC Output Buffer to CPU Input Image

Purpose:

Use CALL 54 to transfer words 200 to 207 of the BASIC module output buffer to words 0 to 7 of the CPU input image table. This routine has no input arguments and one output argument. The output argument is the status of the logic processor.

Word 200 in the BASIC output buffer is reserved and can not be modified. This word provides module status information to the SLC 500 processor. Bit 15 is the module mode bit. It can have one of the following values:

- 0 = SLC Processor is in the RUN mode
- 1 = SLC Processor is not in the RUN mode

Bit 14 of word 200 is the EEPROM checksum bit. It can have one of the following values:

- 0 = EEPROM checksum is correct
- 1 = EEPROM checksum is incorrect

Bit 13 of word 200 is the battery status bit. It can have one of the following values:

- 0 = Battery is good
- 1 = Battery is low

Word integrity is guaranteed during this transfer. File integrity is not. Handshaking bits can be used in your application program to provide file integrity.

Syntax:

CALL 54
POP [processor mode]

Example:

```
>1  REM EXAMPLE PROGRAM
>30 CALL 54  : REM XFER BASIC OUTPUT BUFFER TO CPU INPUT IMAGE TABLE
>40 POP X : REM LOGIC PROCESSOR STATUS IS IN X
>50 IF X<>0 THEN PRINT “PROCESSOR NOT IN RUN MODE”

READY
>RUN

READY
>
```

CALL 57 - Transfer BASIC Output Buffer to CPU M1 File

Purpose:

Use CALL 57 to transfer up to 64 words starting at word 100 from the BASIC module output buffer to the CPU M1 file starting at word 0. This routine has one input argument and one output argument. The input argument is the number of words to be transferred (1 to 64). If the number is not within the range 1 to 64, no transfer occurs and the output argument is set to 10.

If the input argument is a valid number, then the output argument is the status of the Logic Processor. It can have one of the following values:

- 0 = Successful Transfer, SLC Processor in RUN mode
- 1 = Successful Transfer, SLC Processor in PROGRAM mode
- 2 = Successful Transfer, SLC Processor in TEST mode
- 10 = Illegal length specified
- 11 = SLC Processor does not support this capability

Word integrity is guaranteed during this transfer. File integrity is not. Handshaking bits can be used in your application program to provide file integrity.

Syntax:

PUSH [words to transfer]
CALL 57
POP [processor mode]

Example:

```
>1  REM EXAMPLE PROGRAM
>50 PUSH 64 : REM TRANSFER LENGTH IS 64 WORDS
>60 CALL 57 : REM TRANSFER BASIC OUTPUT BUFFER TO M1
>70 POP X : REM LOGIC PROCESSOR STATUS IS IN X
>80 PRINT X

READY
>RUN 0

READY
>
```

CALL 85 - Transfer BASIC Output Buffer to DH-485 Common Interface File

Purpose:

Use CALL 85 to transfer up to 40 words starting at the designated word offset of the BASIC module output buffer to the DH-485 Common Interface File starting at the same designated offset from word 0.

This routine has two input arguments and one output argument. The first input argument is the starting word offset into the DH-485 Common Interface File and the BASIC module output buffer (0 to 39). If the number of words to be transferred is not within the range 0 to 39, then the output argument equals 1. If the number is not within the range 0 to 39, then the output argument equals 2. If the maximum word offset of 39 is exceeded, then the transfer does not take place. The second input argument is the number of words to be transferred (1-40).

The output argument specifies the transfer status. It can have one of the following values:

- 0 = Successful transfer
- 1 = Illegal starting offset
- 2 = Illegal length

Word integrity is guaranteed during this transfer. File integrity is not.

Syntax:

PUSH [DH-485 common interface file word offset]
PUSH [number of words to be transferred]
CALL 85
POP [transfer status]

Example:

```
>1  REM EXAMPLE PROGRAM
>40 PUSH 31 : REM OFFSET ADDRESS = 31
>50 PUSH 3 : REM WORD LENGTH = 3
>60 CALL 85 : REM TRANSFER DATA TO DH-485 COMMON INTERFACE FILE
>70 POP R
>80 IF R<>0 PRINT "TRANSFER ERROR CODE = ",R : REM PRINT ERROR

READY
>RUN

READY
>
```

CALL 91 - Write BASIC Output Buffer to Remote DH-485 Data File

Purpose:

Use CALL 91 to write up to 40 words starting at word 0 of the BASIC module output buffer to the remote DH-485 data file at the designated node address, file #, file type, and element offset. This routine has six input arguments and one output argument.

The first input argument is the Node Address of the remote device (0 to 31). If the number is not within the range 0 to 31, then the output argument equals 10, and the Write Message does not take place.

The second input argument is the File # on the remote device (0 to 255). If the number is not within the range 0 to 255, then the output argument equals 11, and the Write Message does not take place.

The third input argument is the File Type to be written to the remote device. Valid type codes are ASC(N), ASC(S), ASC(C), ASC(T), ASC(B), and ASC(R). If the file type is not one of these valid types, then the output argument equals 241, and the Write Message does not take place.

File Type	File Type Code	Words/Element
Integer File	ASC(N)	1 word/element
Status File	ASC(S)	1 word/element
Counter File	ASC(C)	3 words/element
Timer File	ASC(T)	3 words/element
Bit File	ASC(B)	1 word/element
Control File	ASC(R)	3 words/element

The fourth input argument is the starting element offset within the file on the remote device (0 to 32767). If the number is not within the range (0 to 32767), then the output argument equals 12, and the transfer does not take place.

The fifth input argument is the number of elements to be transferred. If the number is not within the range specified below, then the output argument equals 13, and the transfer does not take place.

File Type	Valid Length Range
ASC(N)	1 to 40
ASC(S)	1 to 40
ASC(C)	1 to 13
ASC(T)	1 to 13
ASC(B)	1 to 40
ASC(R)	1 to 13

The sixth input argument is the Message Time Out value. This value is the number of hundreds of milliseconds that are allowed to receive the read response (1 to 50 = 0.1 to 5.0 seconds). If the read response is not received within this time, the message aborts with the output argument equal to 55. If the number is not within the range 1 to 50, the output argument equals 14, and the transfer does not take place.

The Write data from the BASIC module output buffer written to the Remote Device starting at word 0 and filling as many words as specified by the element length of the message.

The output argument specifies the status of the message instruction. Upon return from the CALL, the output argument has the following definition.

Decimal Output	Hexadecimal Output	Description
0	00	Successful Completion.
2	02	Target Node cannot accept the message at this time.
3	03	Target Node cannot respond because message is too large.
4	04	Target Node cannot respond because it does not understand the command parameters.
5	05	BASIC module is off–line (not on link).
6	06	Target Node cannot respond because requested function is not available.
7	07	Target Node does not respond.

Decimal Output	Hexadecimal Output	Description
10	0A	BASIC module detects illegal target node address.
11	0B	BASIC module detects illegal File#.
12	0C	BASIC module detects illegal target file element offset.
13	0D	BASIC module detects illegal target file length.
14	0E	BASIC module detects illegal time out value.
16	10	Target Node cannot respond because of incorrect command parameters or unsupported command.
55	37	Message timed out (time out value exceeded).
80	50	Target Node is out of memory.
96	60	Target Node cannot respond because file is protected.
231	E7	Target Node cannot respond because length requested is too large.
235	EB	Target Node cannot respond because target node denies access.
236	EC	Target Node cannot respond because requested function is currently unavailable.
241	F1	BASIC module detects illegal target file type.
250	FA	Target node cannot respond because another node is file owner (has sole file access).
251	FB	Target Node cannot respond because another node is program owner (has sole access to all files).

This CALL is implemented as a Protected Typed Logical Write with two address fields.

Syntax:

PUSH [remote device node address]
PUSH [remote device file number]
PUSH [remote device file type]
PUSH [starting element offset of remote device file]
PUSH [number of elements to be transferred]
PUSH [message timeout value]
CALL 91
POP [status of message instruction]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 PUSH 1 : REM REMOTE NODE ADDRESS = 1
>20 PUSH 7 : REM REMOTE FILE # 7
>30 PUSH ASC(N) : REM FILE TYPE = INTEGER
>40 PUSH 0 : REM OFFSET = 0
>50 PUSH 10: REM WORD LENGTH = 10
>60 PUSH 5 : REM THE TIME OUT VALUE = 0.5 SECOND
>70 CALL 91 : REM WRITE DATA FROM OUTPUT BUFFER
>80 POP R : REM GET THE OUTPUT ARGUMENT
>90 IF R<>0 PRINT "READ ERROR CODE =",R

READY
>RUN

READ ERROR CODE =  5

READY
>
```

CALL 93 - Write Output Buffer to Remote DH-485 Common Interface File

Purpose:

Use CALL 93 to write up to 40 words starting at word 0 of the BASIC module output buffer to the remote DH-485 Common Interface File at the designated node address, starting at the designated word offset . This routine has four input arguments and one output argument.

The first input argument is the Node Address of the remote device (0 to 31). If the number is not within the range 0 to 31, then the output argument equals 10, and the Write Message does not take place.

The second input argument is the starting word offset within the file on the remote device (0 to 32767). If the number is not within the range (0 to 32767), then the output argument equals 12, and the transfer does not take place.

The third input argument is the number of words to be transferred. If the number is not within the range specified below, then the output argument equals 13, and the transfer does not take place.

The fourth input argument is the Message Time Out value. This value is the number of hundreds of milliseconds that are allowed to receive the read response (1 to 50 = 0.1 to 5.0 seconds). If the read response is not received within this time, the message aborts with the output argument equal to 55. If the number is not within the range 1 to 50, the output argument equals 14, and the transfer does not take place.

The data from the BASIC module output buffer starting at word 0 is written to the Remote Common Interface File starting at the specified word, and filling as many words as specified by the word length of the message.

The output argument specifies the status of the message instruction. Upon return from the CALL, the output argument has the following definition.

Decimal Output	Hexadecimal Output	Description
0	00	Successful Completion.
2	02	Target Node cannot accept the message at this time.
3	03	Target Node cannot respond because message is too large.
4	04	Target Node cannot respond because it does not understand the command parameters.
5	04	BASIC module is off–line (not on link).
6	04	Target Node cannot respond because requested function is not available.
7	07	Target Node does not respond.
10	0A	BASIC module detects illegal target node address.
11	0B	BASIC module detects illegal File#.
12	0C	BASIC module detects illegal target file element offset.
13	0D	BASIC module detects illegal target file length.
14	0E	BASIC module detects illegal time out value.
16	10	Target Node cannot respond because of incorrect command parameters or unsupported command.
55	37	Message timed out (time out value exceeded).
80	50	Target Node is out of memory.
96	60	Target Node cannot respond because file is protected.
231	E7	Target Node cannot respond because length requested is too large.
235	EB	Target Node cannot respond because target node denies access.
236	EC	Target Node cannot respond because requested function is currently unavailable.
241	F1	BASIC module detects illegal target file type.

Decimal Output	Hexadecimal Output	Description
250	FA	Target node cannot respond because another node is file owner (has sole file access).
251	FB	Target Node cannot respond because another node is program owner (has sole access to all files).

Syntax:

PUSH [remote device node address]
PUSH [starting word offset of remote device file]
PUSH [number of words to be transferred]
PUSH [message timeout value]
CALL 93
POP [status of message instruction]

Example:

```
>1  REM EXAMPLE PROGRAM
>30 PUSH 1 : REM REMOTE NODE ADDRESS = 1
>40 PUSH 0 : REM OFFSET = 0
>50 PUSH 10 : REM WORDLENGTH = 10
>60 PUSH 5 : REM THE TIMEOUT VALUE = 0.5 SECONDS
>70 CALL 93 : REM WRITE DATA FROM BASIC OUTPUT BUFFER
>80 POP R : REM GET THE OUTPUT ARGUMENT
>90 IF R<>0 THEN PRINT  READ ERROR CODE = ",R


READY
>RUN

READ ERROR CODE =  5

READY
>
```

CALL 94 - Display Current PRT1 Port Setup

Purpose:

Use CALL 94 to display the current PRT1 port configuration on the program port terminal screen. No arguments are PUSHed or POPped.

Syntax:

CALL 94

Example:

```
>CALL 94

19200 Baud
Hardware Handshaking OFF
1 Stop Bit(s)
No Parity
8 Bits/Char
Xon/Xoff
```

CALL 96 - Clear PRT1 Input/Output Buffers

Purpose:

Use CALL 96 to clear port PRT1 input and output buffers. Use the following PUSHes to clear the corresponding buffer:

- PUSH 0 to clear the output buffer
- PUSH 1 to clear the input buffer
- PUSH 2 to clear both buffers

Important: If port PRT1 is configured for DH-485 protocol, this call has no effect.

Syntax:

PUSH [buffer selection]
CALL 96

Example:

```
>1  REM EXAMPLE PROGRAM
>10 PUSH 0 : CALL 96 : REM CLEAR PRT1 OUTPUT BUFFER

READY
>
```

CALL 112 - User LED Control

Purpose:

Use CALL 112 to activate or de-activate the user LEDs (LED1 and LED2). Two inputs arguments are required and no output arguments. The first input argument activates or de-activates LED1. The second input argument activates or de-activates LED2. An input argument of one (1) activates the LED. An input argument of zero (0) deactivates the LED. Any other value has no effect on that particular LED.

Syntax:

PUSH [LED1 state]
PUSH [LED2 state]
CALL 112

Example:

```
>1   REM EXAMPLE PROGRAM
>100 PUSH 1 : REM TURN ON LED1
>110 PUSH 0 : REM TURN OFF LED2
>120 CALL 112 : REM SET THE LEDS

READY
>RUN

READY
>
```

CALL 114 - Transmit DF1 Packet

Purpose:

Use CALL 114 to transmit the DF1 data packet. This routine has no input arguments and no output arguments. When CALL 114 is performed, the DF1 data is “posted” for the DF1 driver to transmit as a single message packet. If half-duplex slave operation is selected, the message packet is transmitted the next time an ENQuiry is received from the DF1 master. If full-duplex operation is selected, the message packet is transmitted immediately.

Use one or more PRINT#, PH0.#, or PH1.# statements to construct the desired data in the transmit buffer of port PRT2. After constructing the data in the transmit buffer, use CALL 114 to initiate transmission of the data inside a DF1 message packet.

Caution must be exercised when building DF1 data packets. If an attempt is made to transmit five or less bytes of data (minimum is six bytes), the error message “ERROR - DF1 Data Packet To Transmit Is Too Small” is sent to the program port and the BASIC module enters command mode.

If an attempt is made to place more than 256 bytes of data into the transmit buffer, the error message “BUFFER OVERFLOW” is sent to the program port and the BASIC module enters command mode.

The user’s program must wait for one transmission to complete before construction of another data packet may be performed. Use CALL 115 to check the DF1 transmission status to determine when a transmission is complete.

Syntax:

CALL 114

Example:

```
>1  REM EXAMPLE PROGRAM
>10 CALL 114
>20 END
```

CALL 115 - Check DF1 XMIT Status

Purpose:

Use CALL 115 to check the DF1 transmit status. This routine has no input arguments and one output argument. The output argument returns a value that represents the DF1 transmit status. The possible DF1 transmit status values are shown below:

- 0 = No transmit result pending
- 1 = Transmit result pending
- 2 = Transmission successful
- 3 = Transmission failed
- 4 = Enquiry timeout, no transmission
- 5 = If modem handshaking is selected, either a loss of CTS signal while transmitting or a fatal transmitter failure has occurred.
  If no handshaking is selected, a fatal transmitter failure has occurred
- 6 = If modem handshaking with constant carrier has been selected for either half-duplex or full-duplex modes, this error indicates transmission failure due to modem disconnection (DCD signal loss for more than 10 seconds)
- 7 = DF1 driver is not enabled

Important: Transmit status value “4” should never be returned if full-duplex mode is selected.

Syntax:

CALL 115
POP [DF1 transmit status]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 CALL 113
>20 POP X
>30 END
```

PRINT

Purpose:

Use the PRINT statement to direct the BASIC module to output a value to the console device. You may print the value of expressions, strings, literal values, variables or text strings. You may combine the various forms in the print list by separating them with commas. If the list is terminated with a comma, the carriage return/line feed is suppressed. P. is a “shorthand” notation for PRINT.

Important: The BASIC Intepretor terminates the printing of a string if it encounters a NULL (0), or CR (13) character. If you want to print strings containing these value, print the characters individually inside of a loop construct. Remember, to suppress the CR LF on the loop Print instruction use a trailing comma.

Syntax:

PRINT

Example:

```
>PRINT 10*10,3*3
 100  9

>PRINT ”1746-BAS”
1746-BAS

>PRINT 5,1E3
 5  1000
```

Important: Values are printed next to one another with two intervening blanks. A PRINT statement with no arguments sends a carriage return/line feed sequence to the console device.

The symbols @ and # can be used to direct the print output to ports PRT1 and PRT2 respectively.

Use the PRINT CR expression to output a carriage return without a line feed.

```
>1  REM EXAMPLE PROGRAM
>10 PRINT “A”, CR
>20 PRINT “B”

READY
>RUN

B

READY
>
```

The “A” was printed and then overwritten by the “B”.

Use the PRINT SPC( ) expression to output a specified number of spaces.

```
>1  REM EXAMPLE PROGRAM
>10 PRINT “A”, SPC(10), “B”

READY
>RUN

A          B
```

Use the PRINT TAB( ) expression to output a specified number of tab characters.

```
>1  REM EXAMPLE PROGRAM
>10 PRINT “A”, TAB(1), “B”

READY
>RUN

A    B
```

Use the PRINT USING(Fx) expression to output all numeric values in scientific notation. The x represents the total number of digits of the mantissa that are displayed. One digit is displayed before the decimal point. The value of x is a minimum of three and a maximum of eight. The value displayed will be adjusted according to these limits.

```
>1  REM EXAMPLE PROGRAM
>10 PRINT USING(F4), 123.45678

READY
>RUN

1.234 E+2
```

Use the PRINT USING(#.#) expression to output all numeric values in decimal notation according to the format specified by the instruction.

```
>1  REM EXAMPLE PROGRAM
>10 PRINT USING(###.##),
>20 PRINT 4.67890, 123.456
>30 PRINT .0123, .234
>40 PRINT 123.456, 2.1

READY
>RUN

4.97       123.45
0.01       0.23
123.45     2.10

READY
>
```

PH0., PH1.

Purpose:

Use the PH0. and PH1. statements to direct the BASIC module to output a hexadecimal value to the console device. These statements function the same as the PRINT statement except that the values are printed out in a hexadecimal format. The PH0. statement suppresses two leading zeros if the number printed is less than 255 (0FFH). The PH1. statement always prints out four hexadecimal digits.

The character “H” is always printed after the number when PH0. or PH1. is used to direct an output. The values printed are always truncated integers. If the number printed is not within the range of valid integer (example: is between 0 and 65535 (0FFFFH) inclusive), the BASIC module defaults to the normal mode of print. If this happens no “H” prints out after the value. Since integers are entered in either decimal or hexadecimal form, the statements PRINT, PH0., and PH1. can be used to perform decimal to hexadecimal and hexadecimal to decimal conversion. All comments that apply to the PRINT statement apply to the PH0. and PH1. statements.

Syntax:

PH0., PH1.

Example:

```
>PH0.2*2
 04H

>PH1.2*2
 0004H

>PH0. 100
 64H

>PH0. 1000
 3E8H

>PH1. 1000
 03E8H

>PH1. 3E8
 3.0 E+8

>PH0. PI
 03H

>
```

ST@

Purpose:

Use the ST@ statement to store BASIC module floating point numbers to a specified address. The expression [expr] following the ST@ statement specifies the address where the number is to be stored in RAM. The ST@ statement takes the value on the top of the argument stack and stores it in RAM at the address location specified by [expr].

This statement can be used with CALL 77 to store variables to a protected area of memory. This protected area is not zeroed on power-up or when the RUN command is issued.

Syntax:

ST@ [expr]

Example:

```
>P. MTOP
 24515

P. MTOP 10*6
 24455

>PUSH 24455 : CALL 77

>P. MTOP
 24455
```

```
>1  REM EXAMPLE PROGRAM
>5  DIM A(10),B(10)
>10 REM *** ARRAY SAVE ***
>20 FOR I = 0 TO 9
>30 A(I) = I+20
>40 PUSH A(I)
>50 ST@ 5FFFH-I*6
>60 NEXT I
>70 REM *** GET ARRAY ***
>80 FOR I = 0 TO 9
>90 LD@ 5FFFH-I*6
>100 POP B(I)
>110 PRINT B(I)
>120 NEXT I

READY
>RUN

 20
 21
 22
 23
 24
 25
 26
 27
 28
 29

READY
>PUSH 5FFFH : CALL 77

>P. MTOP
 24575
```

Chapter 13

Input Functions

CALL 35 - Get Numeric Input Character from PRT2

Purpose:

Use CALL 35 to retrieve the current character in the 255 character input buffer of port PRT2. It returns the decimal representation of the characters received as its output argument. Port PRT2 receives data transmitted by your device and stores it in this buffer. If there is no character, the output argument is a 0 (null). If there is a character, the output argument is the ASCII value of that character. There is no input argument for this routine.

Syntax:

CALL 35
POP [ASCII value of character]

Examples:

```
>1  REM EXAMPLE PROGRAM
>10 CALL 35
>20 POP X
>30 IF X=0 THEN GOTO 10
>40 PRINT CHR(X)

READY
>RUN

STOP – IN LINE  30
READY
>
```

Important: A 0 (null) is a valid character in some communication protocols. Use CALL 36 to determine the actual number of characters in the buffer.

Important: Purge the buffer before storing data to ensure data validity.

```
>1   REM EXAMPLE PROGRAM
>10  REM PERIPHERAL PORT INPUT USING CALL 35
>20  STRING 200,20
>30  DIM D(254)
>40  CALL 35 : POP X
>50  IF X <>2 GOTO 40
>55  REM WAIT FOR DEVICE TO SEND START OF TEXT
>60  REM
>70  DO
>80  I=I+1
>90  CALL 35 : POP D(I): REM STORE DATA IN ARRAY
>100 UNTIL D(I)=3 : REM WAIT FOR DEVICE TO SEND END OF TEXT
>120 REM
>130 REM FORMAT AND PRINT DATA TYPES
>140 PRINT “RAW DATA=”
>150 FOR J=1 TO I : PRINT D(J),: NEXT J
>155 REM PRINT RAW DECIMAL DATA
>160 PRINT: PRINT: PRINT
>170 PRINT “ASCII DATA=”
>180 FOR J=1 TO I : PRINT CHR(D(J)),:NEXT J
>185 REM PRINT ASCII DATA
>190 PRINT: PRINT: PRINT
>200 PRINT “$(1)=”
>210 FOR J=1 TO I: ASC($(1),J)=D(J): NEXT J
>215 REM STORE DATA IN STRING
>220 PRINT $(1)
>230 PRINT: PRINT: PRINT
>240 I=0
>250 REM
>260 GOTO 40

READY
>RUN


RAW DATA=
 65 66 67 68 69 70 71 49 50 51 52 53 54 55 56 57 3


ASCII DATA=
 ABCDEFG123456789


$(1)=
 ABCDEFG123456789
```

CALL 53 - Transfer CPU Output Image to BASIC Input Buffer

Purpose:

Use CALL 53 to transfer words 0 to 7 of the CPU output image table to words 200 to 207 of the BASIC module input buffer. This routine has no input arguments and one output argument. The output argument is the status of the logic processor. It can have one of the following values:

- 0 = Logic processor is in the RUN mode
- 1 = Logic processor is not in the RUN mode

Word integrity is guaranteed during this transfer. File integrity is not. Handshaking bits can be used in your application program to provide file integrity.

All data transferred to the BASIC module from the SLC 500 processor must be routed through the BASIC module input buffer. Table 14.A lists the definition of the addresses in the BASIC module input buffer.

Table 14.A
BASIC Module Input Buffer Addresses

Address	Definition
0 to 39	Data transferred from the DH-485 common interface file.
40 to 99	Reserved
100 to 163	Data transferred from the SLC 500 CPU module M0 file.
164 to 199	Reserved
200 to 207	Data transferred from the SLC 500 CPU output image table.

Syntax:

CALL 53
POP [processor status]

Example:

```
>1  REM EXAMPLE PROGRAM
>30 CALL 53 : REM XFER CPU OUTPUT IMAGE TO BASIC INPUT BUFFER
>40 POP X : REM LOGIC PROCESSOR STATUS
>50 IF (X<>0) THEN PRINT “PROCESSOR NOT IN RUN MODE”

READY
>RUN
```

CALL 56 - Transfer CPU M0 File to BASIC Input Buffer

Purpose:

Use CALL 56 to transfer up to 64 words starting at word 0 of the CPU M0 file to the BASIC module input buffer starting at word 100. This routine has one input argument and one output argument. The input argument is the number of words to be transferred (0 to 64). If the number is not within the range 0 to 64, no transfer occurs, and the output argument sets to 10. If the input argument is valid number, the output argument is the status of the Logic Processor. It can have one of the following values:

- 0 = Successful Transfer, Logic Processor in RUN mode
- 1 = Successful Transfer, Logic Processor in PROGRAM mode
- 2 = Successful Transfer, Logic Processor in TEST mode
- 10 = Illegal length specified
- 11 = Logic Processor does not support this capability

Word integrity is guaranteed during this transfer. File integrity is not. Handshaking bits can be used in your application program to provide file integrity.

Syntax:

PUSH [number of words to be transferred]
CALL 56
POP [processor status]

Examples:

```
>1  REM EXAMPLE PROGRAM
>30 PUSH 64 :  REM TRANSFER 64 WORDS
>40 CALL 56 :  REM TRANSFER M0 TO BASIC INPUT BUFFER
>50 POP X :  REM LOGIC PROCESSOR STATUS IS IN X
>60 IF (X=10) PRINT “ILLEGAL INPUT ARGUMENT”
>70 IF (X<>0).AND.(X<>10) THEN  PRINT “PROCESSOR NOT IN RUN MODE”

READY
>RUN
```

CALL 84 - Transfer DH-485 Interface File to BASIC Input Buffer

Purpose:

Use CALL 84 to transfer up to 40 words starting at the designated offset of the DH-485 Common Interface File to the BASIC module input buffer starting at the same designated offset from word 0. This routine has two input arguments and one output argument. The first input argument is the starting offset in the DH-485 Common Interface File and the BASIC module input buffer (0 to 39). If the number is not within the range 0 to 39, the output argument equals 1, and the transfer does not take place. The second input argument is the length in words to be transferred (1 to 40). If the number of words is not within the range 1 to 40, the output argument equals 2, and the transfer does not take place.

- 0 = Successful transfer
- 1 = Illegal starting offset
- 2 = Illegal length

Word integrity is guaranteed during this transfer. File integrity is not. Handshaking bits can be used in your application program to provide file integrity.

Syntax:

PUSH [starting word offset in DH-485 interface file]
PUSH [number of words to be transferred]
CALL 84
POP [transfer status]

Example:

```
>1  REM EXAMPLE PROGRAM
>40 PUSH 0 : REM OFFSET ADDRESS = 0
>50 PUSH 32 : REM WORD OFFSET = 32
>60 CALL 84 : REM TRANSFER THE DATA TO THE BASIC INPUT BUFFER
>70 POP R : REM GET THE OUTPUT ARGUMENT
>80 IF (R<>0) THEN PRINT “TRANSFER ERROR CODE = ”,R : REM PRINT ERROR

READY
>RUN

READY
>
```

CALL 90 - Read Remote DH-485 Data File to BASIC Input Buffer

Purpose:

Use CALL 90 to read up to 40 words from the designated node address, file #, file type, and element offset of a remote DH-485 data file to the BASIC module input buffer starting at word 0. This routine has six input arguments and one output argument.

The first input argument is the Node Address of the remote device (0 to 31). If the number is not within the range 0 to 31, then the output argument equals 10, and the Read Message does not take place.

The second input argument is the File # on the remote device (0 to 255). If the number is not within the range 0 to 255, then the output argument equals 11, and the Read Message does not take place.

The third input argument is the File Type to be read from the remote device. Valid file type codes are ASC(N), ASC(S), ASC(C), ASC(T), ASC(B), and ASC(R). If the file type is not one of these valid types, then the output argument equals 241, and the Read Message does not take place.

File Type	File Type Code	Words/Element
Integer File	ASC(N)	1 word/element
Status File	ASC(S)	1 word/element
Counter File	ASC(C)	3 words/element
Timer File	ASC(T)	3 words/element
Bit File	ASC(B)	1 word/element
Control File	ASC(R)	3 words/element

The fourth input argument is the starting element offset within the file on the remote device (0 to 32767). If the number is not within the range 0 to 32767, then the output argument equals 12, and the transfer does not take place.

The fifth input argument is the number of elements to be transferred. If the number is not within the valid length range specified below, then the output argument equals 13, and the transfer does not take place.

File Type Code	Valid Length Range
ASC(N)	1 to 40
ASC(S)	1 to 40
ASC(C)	1 to 13
ASC(T)	1 to 13

File Type Code	Valid Length Range
ASC(B)	1 to 40
ASC(R)	1 to 13

The sixth input argument is the Message Time Out value. This value is the number of hundreds of milliseconds that are allowed to receive the read response (1 to 50 = 0.1 to 5.0 seconds). If the read response is not received within this time, the message aborts with the output argument equal to 55. If the number is not within the range 1 to 50, the output argument equals 14, and the transfer does not take place.

The read data from the remote device is read into the BASIC module input buffers starting at word 0 and filling as many words as specified by the element length of the message.

The output argument specifies the status of the message instruction. Upon return from the CALL, the output argument has the following definition.

Decimal Output	Hexadecimal Output	Description
0	00	Successful Completion.
2	02	Target Node cannot accept the message at this time.
3	03	Target Node cannot respond because message is too large.
4	04	Target Node cannot respond because it does not understand the command parameters.
5	05	BASIC module is off–line (not on link).
6	06	Target Node cannot respond because requested function is not available.
7	07	Target Node does not respond.
10	0A	BASIC module detects illegal target node address.
11	0B	BASIC module detects illegal File#.
12	0C	BASIC module detects illegal target file element offset.
13	0D	BASIC module detects illegal target file length.
14	0E	BASIC module detects illegal time out value.
16	10	Target Node cannot respond because of incorrect command parameters or unsupported command.
55	37	Message timed out (time out value exceeded).
80	50	Target Node is out of memory.
96	60	Target Node cannot respond because file is protected.

Decimal Output	Hexadecimal Output	Description
231	E7	Target Node cannot respond because length requested is too large.
235	EB	Target Node cannot respond because target node denies access.
236	EC	Target Node cannot respond because requested function is currently unavailable.
241	F1	BASIC module detects illegal target file type.
250	FA	Target node cannot respond because another node is file owner (has sole file access).
251	FB	Target Node cannot respond because another node is program owner (has sole access to all files).

Syntax:

PUSH [remote device node address]
PUSH [remote device file number]
PUSH [remote device file type]
PUSH [starting element offset of remote device file]
PUSH [number of elements to be transferred]
PUSH [message timeout value]
CALL 90
POP [status of message instruction]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 PUSH 1 : REM REMOTE NODE ADDRESS = 1
>20 PUSH 5 : REM REMOTE FILE   5
>30 PUSH ASC(C) : REM FILE TYPE = COUNTER
>40 PUSH 0 : REM OFFSET = 0
>50 PUSH 10 : REM ELEMENT LENGTH = 10 = 30 WORDS
>60 PUSH 5 : REM TIMEOUT = 0.5 SECONDS
>70 CALL 90
>80 POP R : REM GET THE OUTPUT ARGUMENT
>90 IF (R<>0) THEN PRINT “READ ERROR CODE =”,R

READY
>RUN

READ ERROR CODE = 5
```

CALL 92 - Read Remote DH-485 Common Interface File to BASIC Input Buffer

Purpose:

Use CALL 92 to read up to 40 words from the remote DH-485 Common Interface File of the designated node address, starting at the designated word offset to the BASIC module input buffer starting at word 0. This routine has four input arguments and one output argument.

The first input argument is the Node Address of the remote device (0 to 31). If the number is not within the range 0 to 31, then the output argument equals 10, and the Read Message does not take place.

The second input argument is the starting word offset within the file on the remote device (0 to 32767). If the number is not within the range 0 to 32767, then the output argument equals 12, and the transfer does not take place.

The third input argument is the number of words to be transferred. If the number is not within the range (1 to 40), then the output argument equals 13, and the transfer does not take place.

The fourth input argument is the Message Time Out value. This value is the number of hundreds of milliseconds that are allowed to receive the read response (1 to 50 = 0.1 to 5.0 seconds). If the read response is not received within this time, the message aborts with the output argument equal to 55. If the number is not within the range 1 to 50, the output argument equals 14, and the transfer does not take place.

The read data from the remote device is read into the BASIC module input buffer starting at word 0 and filling as many words as specified by the word length of the message.

The output argument specifies the status of the message instruction. Upon return from the CALL, the output argument has the following definition.

Decimal Output	Hexadecimal Output	Description
0	00	Successful Completion.
2	02	Target Node cannot accept the message at this time.
3	03	Target Node cannot respond because message is too large.
4	04	Target Node cannot respond because it does not understand the command parameters.
5	04	BASIC module is off–line (not on link).

Decimal Output	Hexadecimal Output	Description
6	04	Target Node cannot respond because requested function is not available.
7	07	Target Node does not respond.
10	0A	BASIC module detects illegal target node address.
11	0B	BASIC module detects illegal File#.
12	0C	BASIC module detects illegal target file element offset.
13	0D	BASIC module detects illegal target file length.
14	0E	BASIC module detects illegal time out value.
16	10	Target Node cannot respond because of incorrect command parameters or unsupported command.
55	37	Message timed out (time out value exceeded).
80	50	Target Node is out of memory.
96	60	Target Node cannot respond because file is protected.
231	E7	Target Node cannot respond because length requested is too large.
235	EB	Target Node cannot respond because target node denies access.
236	EC	Target Node cannot respond because requested function is currently unavailable.
241	F1	BASIC module detects illegal target file type.
250	FA	Target node cannot respond because another node is file owner (has sole file access).
251	FB	Target Node cannot respond because another node is program owner (has sole access to all files).

Syntax:

PUSH [remote device node address]
PUSH [starting word offset of remote device file]
PUSH [number of words to be transferred]
PUSH [message timeout value]
CALL 92
POP [status of message instruction]

Example:

```
>1  REM EXAMPLE PROGRAM
>30 PUSH 1 : REM REMOTE NODE ADDRESS = 1
>40 PUSH 0 : REM OFFSET = 0
>50 PUSH 10 : REM WORD LENGTH = 10
>60 PUSH 5   REM TIMEOUT VALUE = 0.5 SECONDS
>70 CALL 92
>80 POP R : REM GET THE OUTPUT ARGUMENT
>90 IF (R<>0) THEN PRINT “READ ERROR CODE IS”,R : REM PRINT ERROR

READY
>RUN

READ ERROR CODE IS 5
```

CALL 117 - Get DF1 Packet Length

Purpose:

Use CALL 117 to get the length of the DF1 data packet. This routine has no input arguments and one output argument. The output argument returns the length of the oldest DF1 packet queued up in the DF1 receive buffer.

Important: If the receive buffer is found to be empty, then 0000 is returned to the argument stack.

When CALL 117 is read in a program, the BASIC module checks to see if DF1 communications has been enabled through CALL 108. If DF1 communications have not been enabled, an error message is printed to the console device and the BASIC module enters command mode.

After the length of the DF1 packet has been retrieved, it must be used in conjunction with the GET statement to retrieve the data in the received DF1 packet.

Syntax:

CALL 117
POP [length of DF1 packet]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 CALL 117
>20 POP X
>30 END
```

GET

Purpose:

Use the GET operator in the RUN mode. It returns a result of zero in the command mode. The GET operator reads the console input device. If a character is available from the console device, the value of the character is assigned to GET. After GET is read in the program, it is assigned the value of zero until another character is sent from the console device.

Use the GET# operator to read port PRT2 and the GET@ operator to read port PRT1. The following example prints the decimal representation of any character sent from the console device.

Syntax:

GET

Example:

```
>1  REM EXAMPLE PROGRAM
>10 A = GET
>20 IF (A<>0) THEN PRINT A : REM ZERO MEANS NO ENTRY
>30 GOTO 10
>RUN

 65 [A]
 49 [1]
 24 [^X]
 50 [2]

STOP – IN LINE  30
READY
>
```

The GET operator is read only once before it is assigned a value of zero. This guarantees that the first character entered is always read, independent of where the GET operator is placed in the program. There is no buffering of characters on the program port.

INPL

Purpose:

Use the INPL statement to read an entire line (up to 254 characters) from program port buffer. The line must be stored in a string variable. The INPL statement reads all characters from the program port until a carriage return or the 254 character limit is reached, whichever comes first. INPL does not echo characters read from the program port.

Use the INPL# statement to read an entire string of characters from the PRT2 port buffer. Use the INPL@ statement to read an entire string of characters from the PRT1 port buffer. Both these statements function like the INPL statement.

Syntax:

INPL string_variable

Example:

```
>1  REM EXAMPLE PROGRAM
>10 STRING 270,254 : REM ONE STRING OF < 254 BYTES
>20 INPL $(0) : REM READ LINE FROM PROGRAM PORT
>30 PRINT# $(0) : REM ECHO STRING TO PORT PRT2
```

INPS

Purpose:

Use the INPS statement to read an entire string of characters from the program port buffer. No characters are echoed. The INPS statement is preferred over INPUT or INPL for communications because all ASCII characters may be significant. INPUT is least desirable because input stops when a comma or a carriage return is seen. INPL terminates when a carriage return is seen.

Use the INPS# statement to read an entire string of characters from the PRT2 port buffer. Use the INPS@ statement to read an entire string of characters from the PRT1 port buffer. Both these statements function like the INPS statement.

Syntax:

INPS string_variable, number_of_characters

Example:

```
>1   REM EXAMPLE PROGRAM
>100 PRINT, “TYPE P TO PROCEED OR S TO STOP”
>110 REM READ SINGLE CHARACTER FROM PROGRAM PORT
>120 INPS $(0),1
>130 IF ASC($(0),0)= ASC(P) GOTO 500
>140 IF ASC($(0),0)= ASC(S) GOTO 700
>150 GOTO 100
```

INPUT

Purpose:

Use the INPUT statement to enter data from the console device during program execution. You may assign data to one or more variables with a single input statement. You must separate the variables with a comma.

Use the INPUT# statement to input data from port PRT2. Use the INPUT@ statement to input data from port PRT1. Both these statements function like the INPUT statement.

Syntax:

INPUT

Examples:

```
>INPUT A,C
```

>INPUT A,C causes a question mark (?) to print on the console device. This prompts you to input two numbers separated by a comma. If you do not enter enough data, the module prints TRY AGAIN on the console device.

```
>1  REM EXAMPLE PROGRAM
>10 INPUT A,C
>20 PRINT A,C
>RUN

?1

TRY AGAIN

?1,2
l      2

READY
```

You can write the INPUT statement so that a descriptive prompt tells you what to enter. The message to be printed is placed in quotes after the INPUT statement. If a comma appears before the first variable on the input list, the question mark prompt character is not displayed.

```
>1  REM EXAMPLE PROGRAM
>10 INPUT “ENTER A NUMBER”A
>20 PRINT SQR(A)
>30 END


READY
>RUN


ENTER A NUMBER
?4
 2

READY
>


>NEW

>1  REM EXAMPLE PROGRAM
>10 INPUT “ENTER A NUMBER – ”,A
>20 PRINT SQR(A)
>30 END

>RUN


ENTER A NUMBER – 25
 5

READY
>
```

You can also assign strings with an INPUT statement. Strings are always terminated with a carriage return (cr). If more than one string input is requested with a single INPUT statement, the module prompts you with a question mark.

```
>1  REM EXAMPLE PROGRAM
>10 STRING 100,20
>20 INPUT “NAME(CR),AGE – ”,$(1),A
>30 PRINT “HELLO ”,$(1), “YOU ARE ”,A,“ YEARS OLD.”
>40 END

READY
>RUN

NAME(CR),AGE – PAM
?29
HELLO PAM YOU ARE 29 YEARS OLD.

READY
>
```

You can assign strings and variables with a single INPUT statement.

```
>1  REM EXAMPLE PROGRAM
>10 STRING 100,10
>20 INPUT “NAME(CR), AGE – ”,$(1),A
>30 PRINT “HELLO ”,$(1),“, YOU ARE ”, A,“ YEARS OLD”
>40 END
>RUN


NAME(CR),AGE – FRED
?15
HELLO FRED, YOU ARE 15 YEARS OLD


READY
>
```

LD@

Important: This instruction is not associated with any port designation.

Purpose:

Use the LD@ statement to retrieve floating point numbers that were stored with a ST@ statement. The expression [expr] following the LD@ statement specifies the address where the number is to be stored after executing the LD@. The LD@ statement places the number on the ARGUMENT STACK at the address location specified by [expr].

This statement can be used with CALL 77 to retrieve variables from a protected area of memory. This protected area is not zeroed on power–up or when the RUN command is issued.

Important: LD@ is not used with any port designation.

Syntax:

LD@ [expr]

Example:

```
>P. MTOP
 24515

P. MTOP 10*6
 24455

>PUSH 24455 : CALL 77

>1  REM EXAMPLE PROGRAM
>5  DIM A(10),B(10)
>10 REM *** ARRAY SAVE ***
>20 FOR I = 0 TO 9
>30 A(I) = I+20
>40 PUSH A(I)
>50 ST@ 5FFFH-I*6
>60 NEXT I
>70 REM *** GET ARRAY ***
>80 FOR I = 0 TO 9
>90 LD@ 5FFFH-I*6
>100 POP B(I)
>110 PRINT B(I)
>120 NEXT I0

READY
>RUN

 20
 21
 22
 23
 24
 25
 26
 27
 28
 29

READY
>PUSH 5FFFH : CALL 77

>P. MTOP
 24575
```

READ

Purpose:

Use the READ statement to retrieve the expressions that are specified in the DATA statement and assign the value of the expression to the variable in the READ statement. The READ statement is always followed by one or more variables. If more than one variable follows a READ statement, they are separated by a comma.

Syntax:

READ

Example:

```
>1  REM EXAMPLE PROGRAM
>10 FOR I = 1 TO 3
>20 READ A,C
>30 PRINT A,C
>40 NEXT I
>50 RESTORE
>60 READ A,C
>70 PRINT A,C
>80 DATA 10,20,10/2,20/2,SIN(PI),COS(PI)

READY
>RUN

 10  20
 5  10
 0 -1
 10  20

READY
>
```

Every time a READ statement is encountered the next consecutive expression in the DATA statement is evaluated and assigned to the variable in the READ statement. You can place DATA statements anywhere within a program. They are not executed and do not cause an error. DATA statements are considered to be chained together and appear to be one large DATA statement. If at anytime all the data is read and another READ statement is executed, the program terminates and the message ERROR: NO DATA – IN LINE XX prints to the console device.

Chapter 14

Setup Functions

CALL 30 - Set PRT2 Port Parameters

Purpose:

Use CALL 30 to set the port parameters for port PRT2. Table 14.A lists the PRT2 port parameters and their selections in the order they are PUSHed on the stack before executing the CALL.

Table 14.A
PRT2 Port Parameters

PRT2 Port Parameters	Selections
Bits per Word	5, 6, 7, 8
Parity Enable	0 = None, 1 = Odd, 2 = Even
Number of Stop Bits	1 = 1 Stop bit , 2 = 2 Stop bits , 3 = 1.5 Stop bits
Software Handshaking	0 = None, 1 = XON-XOF
Hardware Handshaking	0 = Disabled DCD, 1 = Enabled DCD

Syntax:

PUSH [bits per word]
PUSH [parity enable]
PUSH [number of stop bits]
PUSH [software handshaking enable/disable]
PUSH [hardware handshaking enable/disable]
CALL 30

Example:

```
>1  REM EXAMPLE PROGRAM
>10 REM CALL 30 INPUT PARAMETERS:
>20 REM FIRST PUSH : 5, 6, 7, OR 8 (BITS/CHARACTER)
>30 REM SECOND PUSH : 0, 1, OR 2 (NO PARITY, ODD, OR EVEN)
>40 REM THIRD PUSH : 1, 2, OR 3 (1, 2, OR 1.5 STOP BITS)
>50 REM FOURTH PUSH: 0 OR 1 (SOFTWARE HANDSHAKING DISABLE, ENABLED)
>60 REM FIFTH PUSH : 0 OR 1 (HARDWARE HANDSHAKING DISABLED, ENABLED)
>70 REM PRT2 DEFAULT CONFIGURATION IS:
>80 REM 1200 BAUD, 8 BITS/CHAR, NO PARITY, 1 STOP BIT, AND
>90 REM SOFTWARE HANDSHAKING ENABLED
>100 PUSH 8 0,1,1,0 : CALL 30
>110 CALL 31

19200 Baud
Hardware Handshaking OFF
1 Stop Bit(s)
No Parity
8 Bits/Char
Xon/Xoff
>
```

CALL 78 - Set Program Port Baud Rate

Purpose:

Use CALL 78 to change the program port baud rate from its default value (1200 baud) to one of the following: 300, 600, 1200, 2400, 4800, 9600 or 19200 baud. The default baud rate for the program port is 1200 baud if port PRT1 is configured as the program port or 19200 baud if port DH485 is configured as the program port. PUSH the desired baud rate and CALL 78. The program port remains at this baud rate unless CALL 73 is invoked or the following conditions are met:

- the battery is dead or has been removed
- the battery-backup capacitor is discharged
- the EEPROM is removed or not programmed
- and power is cycled

If this happens the baud rate defaults to 1200 baud.

Syntax:

PUSH [baud rate]
CALL 78

Example:

```
>1  REM EXAMPLE PROGRAM
>10 PUSH 4800
>20 CALL 78
```

CALL 99 - Reset Print Head Pointer

Purpose:

Use CALL 99 to reset the internal print head character counter of your printer when printing out wide forms. This call prevents the automatic CR/LF at character 79. You must keep track of the charatcters in each line.

Syntax:

CALL 99

Example:

```
>10 REM EXAMPLE PROGRAM
>20 REM THIS PRINTS TIME BEYOND 80TH COLUMN
>30 PRINT TAB(79)
>40 CALL 99
>50 PRINT TAB(41), “TIME -”,
>60 PRINT H,“:”,M,“:”,S
>70 END
```

CALL 105 - Reset PRT1 to Default Settings

Purpose:

Use CALL 105 to reset the port parameters of port PRT1 to their default settings. Table 14.B lists the default parameters for port PRT1.

Table 14.B
PRT1 Port Parameter Default Settings

PRT1 Port Parameters	Default Setting
Baud rate	1200 baud
Number of data bits	8 bits
Number of stop bits	1 bit
Parity	No parity
Handshaking	Software handshaking

Syntax:

CALL 105

Example:

```
>1  REM EXAMPLE PROGRAM
>10 CALL 105
```

CALL 119 - Reset PRT2 to Default Settings

Purpose:

Use CALL 119 to reset port parameters of PRT2 to their default settings. Table 14.C lists the default port parameter settings for port PRT2.

Table 14.C
PRT2 Port Parameter Default Settings

PRT2 Port Parameters	Default Setting
Baud rate	1200 baud
Number of data bits	8 bits
Number of stop bits	1 bit
Parity	No parity
Handshaking	Software handshaking

Syntax:

CALL 119

Example:

```
>1  REM EXAMPLE PROGRAM
>10 CALL 119
```

MODE

Purpose:

Use the MODE command to set the port parameters of ports PRT1, PRT2, and DH485.

Table 14.D lists the port parameters for ports PRT1 or PRT2.

Table 14.D
PRT1 and PRT2 Port Parameters

Port Parameters	Selections	Default Settings
baud rate	300, 600, 1200, 2400, 4800, 9600, 19200	1200
arg1 (parity)	None (N), Even (E), Odd (O)	N
arg2 (number of data bits)	7 or 8	8
arg3 (number of stop bits)	1 or 2	1
arg4 (handshaking)	No handshaking (N) Software handshaking (S) Hardware handshaking (H) Hardware and software handshaking (B)	S
arg5 (storage type)	Store information in user ROM and RAM (E) Store information in battery backed RAM (R)	R

Important: If any argument (other than port name and baud rate) is left blank, then that argument defaults to the previously specified value for that argument.

Table 14.E lists the port parameters for port DH485.

Table 14.E
DH485 Port Parameters

Port Parameters	Selections	Default Settings
baud rate	300, 600, 1200, 2400, 4800, 9600, 19200	19200 baud
arg1 (host node address)	0 to 31	0
arg2 (module node address)	1 to 31	1
arg3 (maximum node address)	1 to 31	31
arg4 (not used)		
arg5 (storage type)	Store information in user ROM and RAM (E) Store information in battery backed RAM (R)	R

Important: The E storage type option cannot be used if MODE is used as a statement.

Syntax:

MODE(port name, baud rate, arg1, arg2, arg3, arg4, arg5)

Example:

```
>1  REM EXAMPLE PROGRAM
>10 MODE(DH485,19200,0,1,2,,R)
>.
.
>25 MODE(PRT1,1200,N,8,,,)
```

Chapter 15

String Functions

CALL 60 - String Repeat

Purpose:

Use CALL 60 to repeat a character and place it in a string. You can use the String Repeat when designing output formats. First PUSH the number of times to repeat the character, then PUSH the number of the string containing the character to be repeated. No arguments are POPped. You cannot repeat more characters than the string's maximum length.

Syntax:

PUSH [number of times to repeat character]
PUSH [base string number
CALL 60

Example:

```
>1  REM EXAMPLE PROGRAM
>10 REM STRING REPEAT EXAMPLE PROGRAM
>20 STRING 200,48
>30 $(1) = “*”
>40 PUSH 40 : REM THE NUMBER OF TIMES TO REPEAT CHARACTER
>50 PUSH 1 : REM BASE STRING NUMBER
>60 CALL 60
>70 PRINT $(1)
>80 END

READY
>RUN

****************************************

READY
>
```

CALL 61 - String Append

Purpose:

Use CALL 61 to append one string to the end of another string. This call expects two string arguments. The first is the string number of the string to be appended and the second is the string number of the base string. If the resulting string is longer than the maximum string length, the append characters are lost. There are no output arguments. This is a string concatenation assignment. (example: `$(1)=$(1)+$(2)`).

Important: If the new string length exceeds the length allocated by the string command, an error message is printed on the console device and the BASIC module enters command mode.

Syntax:

PUSH [string number to be appended]
PUSH [base string number]
CALL 61

Example:

```
>1  REM EXAMPLE PROGRAM
>10 STRING 200,20
>20 $(1) = “How are ”
>30 $(2) = “you?”
>40 PRINT “BEFORE ”
>50 PRINT “$(1) = ”,$(1)
>60 PRINT “$(2) = ”,$(2)
>70 PUSH 2 : REM STRING NUMBER TO BE APPENDED
>80 PUSH 1 : REM BASE STRING NUMBER
>90 CALL 61 : REM INVOKE STRING APPEND ROUTINE
>100 PRINT “AFTER:”
>110 PRINT “$(1) = ”,$(1)
>120 PRINT “$(2) = ”,$(2)
>130 END

READY
>RUN

BEFORE:
$(1) = How are
$(2) = you?
AFTER:
$(1) = How are you?
$(2) = you?

READY
>
```

CALL 62 - Number to String Conversion

Purpose:

Use CALL 62 to convert a number or numeric variable into a string. You must allocate a minimum of 14 characters for the string. If the exponent of the value to be converted is anticipated to be 100 or greater, you must allocate 15 characters. Error checking traps string allocation of less than 14 characters only.

Syntax:

PUSH [number to convert to string]
PUSH [string number to receive the value]
CALL 62

Example:

```
>1  REM EXAMPLE PROGRAM
>10 STRING 100,14
>20 INPUT “ENTER A NUMBER TO CONVERT TO A STRING ”,N
>30 PUSH N : REM NUMBER TO CONVERT TO STRING
>40 PUSH 1 : REM CONVERT NUMBER TO STRING 1
>50 CALL 62 : REM DO THE CONVERSION
>60 PRINT $(1)
>70 END

READY
>RUN

ENTER A NUMBER TO CONVERT TO A STRING 2E3
2000

READY
>RUN

ENTER A NUMBER TO CONVERT TO A STRING 1.233
1.233

READY
>
```

CALL 63 - String to Number Conversion

Purpose:

Use CALL 63 to convert the first decimal number found in the specified string to a number on the argument stack. Valid numbers and associated characters are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., E, +, –. The comma is not a valid number character and terminates the conversion.

If the string does not contain a legal value, a zero is returned. A valid value is between 1 and 255. PUSH the number of the string to convert. Two POPs are required. First POP the validity of the value, then POP the actual value. If a string contains a number followed by an E and a letter or non-numeric character, it is assumed that no number was found since the letter is not a valid exponent (CALL 63 returns a zero in the first argument popped indicating that no valid number was in the string).

Syntax:

PUSH [string number to convert]
CALL 63
POP [validity of the value]
POP [actual value]

Example:

```
>1  REM EXAMPLE PROGRAM
>20 INPUT “INPUT A STRING TO CONVERT ”,$(1)
>30 PUSH 1 : REM CONVERT STRING 1
>40 CALL 63
>50 POP V,N
>60 IF (V<>0) THEN PRINT $(1),“ ” N:GOTO 80
>70 PRINT “INVALID OR NO VALUE FOUND”
>80 END

READY
>RUN

INPUT A STRING TO CONVERT 123ABC
123ABC  123

READY
>RUN

INPUT A STRING TO CONVERT 1.2E–7
1.2E–7  1.2 E–7

READY
>RUN

INPUT A STRING TO CONVERT 1.3.6
INVALID OR NO VALUE FOUND

READY
>
```

CALL 64 - Find a String in a String

Purpose:

Use CALL 64 to find a string within a string. It locates the first occurrence (position) of this string. This call expects two input arguments. The first is the string to be found, the second is the string to be searched for a match. One return argument is required. If the number is not zero then a match was located at the position indicated by the value of the return argument. This routine is similar to the BASIC INSTR$(findstr$,str$). (example: `L=INSTR$($(1),$(2))`)

Syntax:

PUSH [string number to be found]
PUSH [base string number]
CALL 64
POP [match position]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 REM SAMPLE FIND STRING IN STRING ROUTINE
>20 STRING 100,20
>30 $(1) = “456”
>40 $(2) = “12345678”
>50 PUSH 1 : REM STRING NUMBER OF STRING TO BE FOUND
>60 PUSH 2 : REM BASE STRING NUMBER
>70 CALL 64 : REM GET THE LOCATION OF FIRST CHARACTER
>80 POP L
>90 IF (L=0) THEN PRINT “NOT FOUND”
>100 IF(L>0) THEN PRINT “FOUND AT LOCATION”,L
>110 END

READY
>RUN

FOUND AT LOCATION 4

READY
>
```

CALL 65 - Replace a String in a String

Purpose:

Use CALL 65 to replace a string within a string. Three arguments are expected. The first argument is the number of the string which replaces the string identified by the second argument string number. The third argument is the base string number. There are no return arguments.

Important: If the new string length exceeds the length allocated by the string command, an error message is printed on the console device and the BASIC module enters command mode.

Syntax:

PUSH [new string number]
PUSH [old string number to be replaced]
PUSH [base string number]
CALL 65

Example:

```
>1  REM EXAMPLE PROGRAM
>10 REM SAMPLE OF REPLACE STRING IN STRING
>20 STRING 100,20
>30 $(0) = “RED–LINES”
>40 $(1) = “RED”
>50 $(2) = “BLUE”
>60 PRINT “BEFORE:”
>70 PRINT “$(0) = ”,$(0)
>80 PUSH 2 : REM STRING NUMBER OF THE STRING TO BE REPLACED WITH
>90 PUSH 1 : REM STRING NUMBER OF THE STRING TO BE REPLACED
>100 PUSH 0 : REM BASE STRING NUMBER
>110 CALL 65 : REM INVOKE REPLACE STRING IN STRING
>120 PRINT “AFTER:”
>130 PRINT “$(0) = ”,$(0)
>140 END

READY
>RUN

BEFORE:
$(0) = RED–LINES
AFTER:
$(0) = BLUE–LINES

READY
>
```

CALL 66 - Insert a String in a String

Purpose:

Use CALL 66 to insert a string within another string. The call expects three arguments. The first argument is the position at which to begin the insert. The second argument is the string number of the characters inserted into the base string. The third argument is the number of the base string. This routine has no return arguments.

Important: If the new string length exceeds the length allocated by the string command, an error message is printed on the console device and the BASIC module enters command mode.

Syntax:

PUSH [insert position]
PUSH [string number of inserted character]
PUSH [base string number]
CALL 66

Example:

```
>1  REM EXAMPLE PROGRAM
>10 REM SAMPLE ROUTINE TO INSERT A STRING IN A STRING
>20 STRING 100,15
>30 $(0) = “1234590”
>40 $(1) = “67890”
>50 PRINT “BEFORE:”
>60 PRINT “$(0) = ”,$(0)
>70 PUSH 6 : REM POSITION TO START THE INSERT
>80 PUSH 1 : REM STRING NUMBER TO BE INSERTED
>85 PUSH 0 : REM BASE STRING NUMBER
>90 CALL 66 : REM INVOKE INSERT A STRING IN A STRING
>100 PRINT “$(0) = ”, (0)
>110 END

READY
>RUN

BEFORE:
$(0) = 1234590
$(0) = 123456789090

READY
>
```

CALL 67 - Delete a String in a String

Purpose:

Use CALL 67 to delete a string from within another string. The call expects two arguments. The first argument is the base string number. The second is the number of the string to be deleted from the base string. This routine has no return arguments.

Important: This routine deletes only the first occurrence of the string.

Syntax:

PUSH [base string number]
PUSH [deleted string number]
CALL 67

Example:

```
>1  REM EXAMPLE PROGRAM
>10 REM ROUTINE TO DELETE A STRING IN A STRING
>20 STRING 200,14
>30 $(1) = “123456789012”
>40 $(2) = “12”
>50 PRINT “BEFORE:”
>60 PRINT “$(1) = ”,$(1)
>70 PUSH 1 : REM BASE STRING NUMBER
>80 PUSH 2 : REM STRING NUMBER OF THE STRING TO BE DELETED
>90 CALL 67 : REM INVOKE STRING DELETE ROUTINE
>100 PRINT “AFTER:”
>110 PRINT “$(1) = ”,$(1)
>120 END

READY
>RUN

BEFORE:
$(1) = 123456789012
AFTER:
$(1) = 3456789012

READY
>
```

CALL 68 - Find the Length of a String

Purpose:

Use CALL 68 to determine the length of a string. One input argument is expected. This is the string number on which the routine acts. One output argument is required. It is the actual number of non–carriage return (CR) characters in this string. This is similar to the BASIC command LEN(str$). (example: `L=LEN($l)`).

Syntax:

PUSH [string number]
CALL 68
POP [number of characters]

Example:

```
>1  REM EXAMPLE PROGRAM
>10 REM SAMPLE OF STRING LENGTH
>20 STRING 1 0,10
>30 $(1) = "1234567"
>40 PUSH 1 : REM BASE STRING
>50 CALL 68 : REM INVOKE STRING LENGTH ROUTINE
>60 POP L : REM GET LENGTH OF BASE STRING
>70 PRINT "THE LENGTH OF ",$(1)," IS",L
>80 END

READY
>RUN

THE LENGTH OF 1234567 IS 7

READY
>
```

STRING

Purpose:

Use the STRING statement to allocate memory for strings. Initially, no memory is allocated for strings. If you attempt to define a string with a statement such as LET $(1)="HELLO" before memory is allocated for strings, a MEMORY ALLOCATION ERROR is generated. The first expression ([expr]) in the STRING statement is the total number of bytes you want to allocate for string storage. The second expression ([expr]) gives the maximum number of bytes in each string. The second value should not be larger than 254. These two numbers determine the total number of defined string variables.

The BASIC module requires one additional byte for each string, plus one additional byte overall. The additional character for each string is allocated for the carriage return character that terminates the string. This means that the statement STRING 100,10 allocates enough memory for 9 string variables, ranging from $(0) to $(8) and all of the 100 allocated bytes are used. Note that $(0) is a valid string in the BASIC module.

Important: If an ASCII null character is used within the string it acts as a marker denoting the end of a string.

Important: After memory is allocated for string storage, commands (example: NEW) and statements (example: CLEAR) cannot "de–allocate" this memory. Cycling power also cannot de-allocate this memory unless battery backup is disabled. You can de-allocate memory by executing a STRING 0,0 statement. STRING 0,0 allocates no memory to string variables.

Important: The BASIC module executes the equivalent of a CLEAR statement every time the STRING [expr],[expr] statement executes. This is necessary because string variables and numeric variables occupy the same external memory space. After the STRING statement executes, all variables are "wiped–out." Because of this, you should perform string memory allocation early in a program (during the first statement if possible). If you re-allocate string memory you destroy all defined variables.

Syntax:

STRING [expr], [expr]

Examples:

```
>1  REM EXAMPLE PROGRAM
>10 STRING 100,30
>20 $(0) = "-----MONTHLY REPORT-----"
>30 PRINT $(0)

READY
>RUN

-----MONTHLY REPORT-----

READY
>
```

Appendix **A**

Decimal/Hexadecimal/Octal/ASCII Conversion Table

Column 1				Column 2				Column 3				Column 4			
DEC	HEX	OCT	ASC	DEC	HEX	OCT	ASC	DEC	HEX	OCT	ASC	DEC	HEX	OCT	ASC
00	00	000	NUL	32	20	040	SP	64	40	100	@	96	60	140	\
01	01	001	SOH	33	21	041	!	65	41	101	A	97	61	141	a
02	02	002	STX	34	22	042	”	66	42	102	B	98	62	142	b
03	03	003	ETX	35	23	043	#	67	43	103	C	99	63	143	c
04	04	004	EOT	36	24	044	$	68	44	104	D	100	64	144	e
05	05	005	ENQ	37	25	045	%	69	45	105	E	101	65	145	d
06	06	006	ACK	38	26	046	&	70	46	106	F	102	66	146	e
07	07	007	BEL	39	27	047	’	71	47	107	G	103	67	147	f
08	08	010	BS	40	28	050	(	72	48	110	H	104	68	150	g
09	09	011	HT	41	29	051	)	73	49	111	I	105	69	151	h
10	0A	012	LF	42	2A	052	*	74	4A	112	J	106	6A	152	i
11	0B	013	VT	43	2B	053	+	75	4B	113	K	107	6B	153	j
12	0C	014	FF	44	2C	054	,	76	4C	114	L	108	6C	154	k
13	0D	015	CR	45	2D	055	_	77	4D	115	M	109	6D	155	l
14	0E	016	S0	46	2E	056	.	78	4E	116	N	110	6E	156	m
15	0F	017	S1	47	2F	057	/	79	4F	117	O	111	6F	157	n
16	10	020	DLE	48	30	060	0	80	50	120	P	112	70	160	o
17	11	021	DC1	49	31	061	1	81	51	121	Q	113	71	161	p
18	12	022	DC2	50	32	062	2	82	52	122	R	114	72	162	q
19	13	023	DC3	51	33	063	3	83	53	123	S	115	73	163	r
20	14	024	DC4	52	34	064	4	84	54	124	T	116	74	164	s
21	15	025	NAK	53	35	065	5	85	55	125	U	117	75	165	t
22	16	026	SYN	54	36	066	6	86	56	126	V	118	76	166	u
23	17	027	ETB	55	37	067	7	87	57	127	W	119	77	167	v
24	18	030	CAN	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	EM	57	39	071	9	89	59	131	Y	121	79	171	y
26	1A	032	SUB	58	3A	072	:	90	5A	132	Z	122	7A	172	z
27	1B	033	ESC	59	3B	073	;	91	5B	133	[	123	7B	173	{
28	1C	034	FS	60	3C	074	<	92	5C	134	\	124	7C	174	.
29	1D	035	GS	61	3D	075	=	93	5D	135	]	125	7D	175	}
30	1E	036	RS	62	3E	076	>	94	5E	135	^	126	7E	176	~
31	1F	037	US	63	3F	077	?	95	5F	137	–	127	7F	177	DEL

Appendix B

BASIC Command, Statement, and CALL Quick Reference Guide

Mnemonic	Required PUSHes or POPs	Description	Active In Command Mode Only	Page
ABS( )		Absolute Value		3-9
ASC( )		Returns Integer Value Of ASCII Character		3-11
ATN( )		Returns Arctangent Of Argument		3-8
BRKPNT		Set Program Break Point	*	4-1
CALL 14	PUSH [word number of BASIC input buffer] POP [converted value]	16 Bit Signed Integer to BASIC Floating Point		9-1
CALL 15	PUSH [word number of BASIC input buffer] POP [converted value]	16 Bit Unsigned Integer to BASIC Floating Point		9-1
CALL 18	None	Re-enable the Control C Break Function		4-5
CALL 19	None	Disable the Control C Break Function		4-5
CALL 24	PUSH [value to be converted] PUSH [word number of BASIC output buffer]	BASIC Floating Point to 16 Bit Signed Integer		9-2
CALL 25	PUSH [value to be converted] PUSH [word number of BASIC output buffer]	BASIC Floating Point to 16 Bit Binary		9-3
CALL 30	PUSH [bits per word] PUSH [parity enable] PUSH [number of stop bits] PUSH [software handshaking enable/disable] PUSH [hardware handshaking enable/disable]	Set PRT2 Port Parameters		14-1
CALL 31	None	Display Current PRT2 Port Setup		12-1
CALL 35	POP [ASCII value of character]	Get Numeric Input Character From Port PRT2		13-1
CALL 36	PUSH [buffer selection] POP [number of character]	Get Number of Characters in PRT2 Buffers		11-1
CALL 37	PUSH [buffer selection]	Clear Port PRT2 Input and Output Buffers		12-1
CALL 40	PUSH [hours] PUSH [minutes] PUSH [seconds]	Set Clock/Calendar Time (Hour, Minute, Second)		10-1
CALL 41	PUSH [day] PUSH [month] PUSH [year]	Set Clock/Calendar Date (Day, Month, Year)		10-2

Mnemonic	Required PUSHes or POPs	Description	Active In Command Mode Only	Page
CALL 42	PUSH [day of week]	Set Clock/Calendar – Day of Week		10-3
CALL 43	PUSH [string number]	Retrieve Date/Time String		10-3
CALL 44	POP [day] POP [month] POP [year]	Retrieve Date Numeric (Day, Month, Year)		10-4
CALL 45	PUSH [string number]	Retrieve Time String		10-5
CALL 46	POP [hours] POP [minutes] POP [seconds]	Retrieve Time Numeric		10-6
CALL 47	PUSH [string number]	Retrieve Day of Week String		10-6
CALL 48	POP [day of week]	Retrieve Day of Week Numeric		10-7
CALL 51	POP [output image buffer status]	Check CPU Output Image Buffer		11-2
CALL 52	PUSH [string number]	Retrieve Date String		10-4
CALL 53	POP [processor status]	Transfer CPU Output Image Buffer to BASIC Module Input Buffer		13-3
CALL 54	POP [processor mode]	Transfer BASIC Module Output Buffer to CPU Input Image Buffer		12-2
CALL 55	POP [input image buffer status]	Check CPU Input Image Buffer		11-3
CALL 56	PUSH [number of words to be transferred] POP [processor status]	Transfer CPU M0 File to BASIC Module Input Buffer		13-4
CALL 57	PUSH [words to transfer] POP [processor mode]	Transfer BASIC Module Output Buffer to CPU M1 File		12-3
CALL 58	POP [module file M0 write status]	Check M0 File Status		11-4
CALL 59	POP [module file M1 read status]	Check M1 File Status		11-5
CALL 60	PUSH [number ot times to repeat string] PUSH [base string number]	String Repeat		15-1
CALL 61	PUSH [string number to be appended] PUSH [base string number]	String Append (Concatenation)		15-2
CALL 62	PUSH [value to be converted] PUSH [string number to receive the value]	Number to String Conversion		15-3
CALL 63	PUSH [string number to convert] POP [validity of the value] POP [actual value]	String to Number Conversion		15-4
CALL 64	PUSH [string number to be found] PUSH [base string number] POP [match position]	Find a String in a String		15-5

Mnemonic	Required PUSHes or POPs	Description	Active In Command Mode Only	Page
CALL 65	PUSH [new string number] PUSH [old string number to be replaced] PUSH [base string number]	Replace a String in a String		15-6
CALL 66	PUSH [insert position] PUSH [string number of inserted characters] PUSH [base string number]	Insert String in a String		15-7
CALL 67	PUSH [base string number] PUSH [deleted string number]	Delete String from a String		15-8
CALL 68	PUSH [string number] POP [number of characters]	Determine Length of a String		15-9
CALL 70	None	ROM to RAM Program Transfer		8-1
CALL 71	PUSH [ROM program number]	ROM/RAM to ROM Program Transfer		8-2
CALL 72	None	RAM/ROM Return		8-3
CALL 73	None	Battery–backed RAM Disable		5-1
CALL 74	None	Battery–backed RAM Enable		5-1
CALL 75	POP [processor mode]	Check SLC 500 Controller CPU Status		11-6
CALL 77	PUSH [new MTOP address]	Protected Variable Storage		5-2
CALL 78	PUSH [baud rate]	Set Program Port Baud Rate		14-2
CALL 80	POP [battery status]	Check Battery Condition		11-7
CALL 81	None	User Memory Module Check and Description	*	5-3
CALL 82	None	Check User Memory Module Map	*	5-4
CALL 84	PUSH [DH485 interface file starting word offset] PUSH [number of words to be transferred] POP [transfer status]	Transfer DH485 Interface File to BASIC Module Input Buffer		13-5
CALL 85	PUSH [DH485 interface file starting word offset] PUSH [number of words to be transferred] POP [transfer status]	Transfer BASIC Module Output Buffer to DH485 Interface File		12-4
CALL 86	POP [DH485 interface file remote write status]	Check DH485 Interface File Remote Write Status		11-7
CALL 87	POP [DH485 interface file remote read status]	Check DH485 Interface File Remote Read Status		11-8
CALL 90	PUSH [remote device node address] PUSH [remote device file number] PUSH [remote device file type] PUSH [starting element offset of remote device file] PUSH [number of elements to be transferred] PUSH [message timeout value] POP [status of message instruction]	Read Remote DH485 Data File to BASIC Input Buffer		13-6

Mnemonic	Required PUSHes or POPs	Description	Active In Command Mode Only	Page
CALL 91	PUSH [remote device node address] PUSH [remote device file number] PUSH [remote device file type] PUSH [starting element offset of remote device file] PUSH [number of elements to be transferred] PUSH [message timeout value] POP [status of message instruction]	Write BASIC Module Output Buffer to Remote DH485 Data File		12-5
CALL 92	PUSH [remote device node address] PUSH [starting element offset of remote device file] PUSH [number of words to be transferred] PUSH [message timeout value] POP [status of message instruction]	Read Remote DH485 Interface File to BASIC Module Input Buffer		13-9
CALL 93	PUSH [remote device node address] PUSH [starting element offset of remote device file] PUSH [number of words to be transferred] PUSH [message timeout value] POP [status of message instruction]	Write BASIC Module Output Buffer to Remote DH485 Interface File		12-8
CALL 94	None	Print Current PRT1 Port Setup		12-11
CALL 95	PUSH [buffer selection] POP [number of characters]	Get Number of Characters in PRT1 Buffers		11-9
CALL 96	PUSH [buffer selection]	Clear Port PRT1 Input and Output Buffers		12-11
CALL 97	None	Enable Port PRT2 DTR Signal		11-10
CALL 98	None	Disable Port PRT2 DTR Signal		11-10
CALL 99	None	Reset Print Head Pointer		14-3
CALL 101	PUSH [starting address] PUSH [ending address]	Upload User Memory Module Code to Host		5-4
CALL 103	None	Print Port PRT1 Output Buffer and Pointer		5-5
CALL 104	None	Print Port PRT1 Input Buffer and Pointer		5-6
CALL 105	None	Reset Port PRT1 to Default Settings		14-3
CALL 108	PUSH [operational code] PUSH [poll time-out or ACKnowledge time-out] PUSH [message retries or ENQuiry retries] PUSH [RTS On delay or NAK received retries] PUSH [RTS Off delay or NULL value] PUSH [slave address or NULL value]	Enable DF1 Driver Communications		11-11
CALL 109	None	Print the Argument Stack		5-7
CALL 110	None	Print Port PRT2 Output Buffer and Pointer		5-8
CALL 111	None	Print Port PRT2 Input Buffer and Pointer		5-9
CALL 112	PUSH [LED1 state] PUSH [LED2 state]	User LED Control		12-12
CALL 113	None	Disable DF1 Driver Communications		11-18

Mnemonic	Required PUSHes or POPs	Description	Active In Command Mode Only	Page
CALL 114	None	Transmit DF1 Packet		12-12
CALL 115	POP [DF1 transmit status]	Check DF1 XMIT Status		12-13
CALL 117	POP [DF1 packet length]	Get DF1 Packet Length		13-11
CALL 119	None	Reset Port PRT2 to Default Settings		14-4
CALL 120	PUSH [decimal equivalent of buffer areas]	Clear BASIC Module Input and Output Buffers		11-19
CALL 121	POP [program ID number]	Get SLC Processor Program ID Number		11-20
CBY( )		Retrieve Data from Specified Memory Address		3-16
CHR( )		Counts Value Converted ASCII Character		3-14
CLEAR		Clears Variables, Interrupts & Strings		6-1
CLEARI		Clear Interrupts		6-2
CLEARS		Clear All Stacks		6-2
CLOCK0		Disable Real Time Clock		7-2
CLOCK1		Enable Real Time Clock		7-1
CONT		Continue After A Stop or Control-C	*	4-3
CONTROL C		Stop Execution & Return to Command Mode		4-4
CONTROL Q		Restart A List After Control S		4-7
CONTROL S		Interrupt A List Command		4-6
COS( )		Returns the Cosine of Argument		3-7
DATA		Data to be read by Read Statement		6-3
DBY( )		Retrieve or Assign Data To or From the Internal Data Memory of the BASIC Module		3-17
DIM		Allocate Memory for Array Variables		6-4
DO-UNTIL		Setup Conditional Do-Loop		7-4
DO-WHILE		Setup a Conditional Do-Loop		7-2
EDIT		Edit a Line of the BASIC Program	*	4-8
END		Terminate Program Execution		7-4
EOF		Test for Empty Input Buffer		3-15
ERASE		Delete the Last BASIC Program Stored In ROM by a PROG Command	*	4-9
EXP( )		“e” (2.7182818) TO THE X		3-11
FOR-TO-(STEP)-NEXT		Set Up For-Next Loop		7-5

Mnemonic	Required PUSHes or POPs	Description	Active In Command Mode Only	Page
FREE		Test for Number of Free Bytes of RAM Memory		3-15
GET		Read Console Input Device		13-12
GET#		Read Console Input Device Connected to PRT2		13-12
GET@		Read Console Input Device Connected to PRT1		13-12
GOSUB		Execute Subroutine		8-4
GOTO		Go To Program Line Number		7-7
IDLE		Force BASIC Module to Enter “Wait Until Interrupt Mode”		4-9
IF–THEN–ELSE		Conditional Test		7-8
INPL		Read Line of Characters from the Program Port Buffer		13-13
INPL#		Read Line of Characters from Port PRT2 Buffer		13-13
INPL@		Read Line of Characters from Port PRT1 Buffer		13-13
INPS		Read String of Characters from the Program Port Buffer		13-13
INPS#		Read String of Characters from Port PRT2 Buffer		13-13
INPS@		Read String of Characters from Port PRT1 Buffer		13-13
INPUT		Input a String or Variable		13-14
INPUT#		Input a String or Variable from Port PRT2		13-14
INPUT@		Input a String or Variable from Port PRT1		13-14
INT( )		Integer		3-9
LD@		Load Variable		13-16
LEN		Read the Number of Bytes of Memory in the Current Selected Program		3-16
LET		Assign a Variable or String a Value (LET is optional)		6-5
LIST		List Program to the Console Devise	*	4-10
LIST#		List Program to Serial Printer	*	4-11
LIST@		List Program to Device Connected to Port PRT1	*	4-11
LOG( )		Natural Log		3-11
MODE		Set Port Parameters of Ports PRT1, PRT2, and DH485		4-12, 14-4
MTOP		Read the Last Valid Memory Address		3-16
NEW		Erase the Program Stored in RAM	*	4-13

Mnemonic	Required PUSHes or POPs	Description	Active In Command Mode Only	Page
NEXT		Test For-Next Loop Condition		7-9
NOT( )		One's Complement		3-9
NULL		Set Null Count After Carriage Return-line Feed	*	4-13
ONERR		ONERR or Go To Line Number		8-5
ON-GOSUB		Conditional GOSUB		8-6
ON-GOTO		Conditional GOTO		7-10
ONTIME		Generate An Interrupt When TIME is Equal to or Greater Than ONTIME Argument-line Number		8-7
PH0.		Print Hex Value with Zero Suppression to Console Device		12-16
PH0.#		Print Hex Value with Zero Suppression to PRT2		12-16
PH0.@		Print Hex Value with Zero Suppression to PRT1		12-16
PH1.		Print Hex Value with No Zero Suppression to Console Device		12-16
PH1.#		Print Hex Value with No Zero Suppression to PRT2		12-16
PH1.@		Print Hex Value with No Zero Suppression to PRT1		12-16
PI		PI-3.1415926		3-9
POP		POP Argument Stack to Variables		8-10
PRINT		Print Variables, Strings or Literals to Console Device; P. is Shorthand for Print		12-14
PRINT#		Print to Port PRT2		12-15
PRINT@		Print to Port PRT1		12-15
PRINT CR		Print Carriage Return		12-15
PRINT SPC( )		Print Spaces		12-15
PRINT TAB( )		Print Tabs		12-15
PRINT USING(Fx)		Print Numeric Values in Scientific Notation		12-15
PRINT USING(#.#)		Print Numeric Values in Decimal Notation		12-15
PROG		Save the Current Program in EPROM	*	4-14
PROG1		Save Baud Rate Information in EPROM	*	4-15
PROG2		Save Baud Rate Information in EPROM and Execute Program after Reset	*	4-16
PUSH		PUSH Expressions on Argument Stack		8-8
RAM		Evoke RAM Mode	*	4-18

Mnemonic	Required PUSHes or POPs	Description	Active In Command Mode Only	Page
READ		READ Data in Data Statement		13-18
REM		Specifies a Remark or Comment Line		4-18
REN		Renumber BASIC Program		4-19
RESTORE		RESTORE Read Point		6-6
RETI		Return From Interrupt		8-11
RETURN		RETURN from Subroutine		8-11
RND		Random Number		3-10
ROM		Select ROM Mode	*	4-20
RROM		Select ROM Mode and Execute the Selected Program	*	4-21
RUN		Execute a Program	*	4-22
SGN		Sign		3-10
SIN( )		Returns the Sine of Argument		3-7
SNGLSTP		Initiate Single-step Program Execution	*	4-23
SQR( )		Square Root		3-10
ST@		Store Variable		12-17
STOP		Break Program Execution		8-13
STRING		Allocate Memory for Strings		15-9
TAN( )		Returns The Tangent of the Argument		3-7
TIME		Retrieve And/Or Assign Free Running Clock Value		3-18
VER		Verify BASIC Module Firmware Version		4-24
XBY( )		Retrieve or Assign Data To or From the External Data Memory of the BASIC Module		3-17
XFER		Transfer a Program From ROM to RAM	*	4-25
+		Addition		3-3
/		Division		3-3
**		Exponentiation		3-3
*		Multiplication		3-3
–		Subtraction		3-4
.AND.		Logical AND		3-5
.OR.		Logical OR		3-5

Mnemonic	Required PUSHes or POPs	Description	Active In Command Mode Only	Page
.XOR.		Logical Exclusive OR		3-5
@		Direct Communications to Port PRT1		3-15
#		Direct Communications to Port PRT2		3-15

Symbols

.AND., 3-5
.OR., 3-5
.XOR., 3-5
and @, 3-15

Numbers

16 Bit Signed Integer to BASIC Floating Point–CALL 14, 9-1
16 Bit Unsigned Integer to BASIC Floating Point–CALL 15, 9-1

A

ABS([expr]), 3-9
Add (+), 3-3
Argument Stack, 2-1
Arithmetic Operators, 3-3
ASC([expr]), 3-11
ASCII Conversion Table, A-1
ATN([expr]), 3-8

B

Backplane Conversion Requirements, 2-4
BASIC Floating Point to 16 Bit Binary–CALL 25, 9-3
BASIC Floating Point to 16 Bit Signed Integer–CALL 24, 9-2
BASIC Line Length, 1-2
BASIC line numbers, 1-1
BASIC Module Documentation Set, P-5
BASIC program line, 1-1
BASIC Statements, Commands, and Calls, 1-2
Battery Backed RAM Disable-CALL 73, 5-1
Battery Backed RAM Enable-CALL 74, 5-1
Battery Condition Check–CALL 80, 11-7
BRKPNT, 4-1

C

CALL 101-Upload User Memory Module Code to Host, 5-4
CALL 103-Print PRT1 Output Buffer and Pointer, 5-5
CALL 104-Print PRT1 Input Buffer and Pointer, 5-6
CALL 105–Reset PRT1 to Default Settings, 14-3
CALL 108–Enable DF1 Driver Communications, 11-11
CALL 109-Print Argument Stack, 5-7
CALL 110-Print PRT2 Output Buffer Pointer, 5-8
CALL 111-Print PRT2 Input Buffer Pointer, 5-9
CALL 112–User LED Control, 12-12
CALL 113–Disable DF1 Driver Communications, 11-18
CALL 114–Transmit DF1 Packet, 12-12
CALL 115–Check DF1 XMIT Status, 12-13
CALL 117-Get DF1 Packet Length, 13-11
CALL 119–Reset PRT2 to Default Settings, 14-4
CALL 120–Clear BASIC Module Input and Output Buffers, 11-19
CALL 121–Get SLC Processor Program ID Number, 11-20
CALL 14–16 Bit Signed Integer to BASIC Floating Point, 9-1
CALL 15–16 Bit Unsigned Integer to BASIC Floating Point, 9-1
CALL 18-Enable Control C, 4-5
CALL 19-Disable Control C, 4-5
CALL 24–BASIC Floating Point to 16 Bit Signed Integer, 9-2
CALL 25–BASIC Floating Point to 16 Bit Binary, 9-3
CALL 30–Set PRT2 Port Parameters, 14-1
CALL 31–Display Current PRT2 Port Setup, 12-1

CALL 35-Get Numeric Input Character from PRT2, 13-1
CALL 36–Get Number of Characters in PRT2 Buffers, 11-1
CALL 37–Clear PRT2 Input/Output Buffers, 12-1
CALL 40–Set Clock/Calendar Time, 10-1
CALL 41–Set Clock/Calendar Date, 10-2
CALL 42–Set Day of Week, 10-3
CALL 43–Retrieve Date/Time String, 10-3
CALL 44–Retrieve Date Numeric, 10-4
CALL 45–Retrieve Time String, 10-5
CALL 46–Retrieve Time Numeric, 10-6
CALL 47–Retrieve Day of Week String, 10-6
CALL 48–Retrieve Day of Week Numeric, 10-7
CALL 51–Check CPU Output Image Buffer, 11-2
CALL 52–Retrieve Date String, 10-4
CALL 53-Transfer CPU Output Image to BASIC Input Buffer, 13-3
CALL 54–Transfer BASIC Output Buffer to CPU Input Image, 12-2
CALL 55–Check CPU Input Image Buffer, 11-3
CALL 56-Transfer M0 File to BASIC Input Buffer, 13-4
CALL 57–Transfer BASIC Output Buffer to CPU M1 File, 12-3
CALL 58–Check M0 File, 11-4
CALL 59–Check M1 File, 11-5
CALL 60–String Repeat, 15-1
CALL 61–String Append, 15-2
CALL 62–Number to String Conversion, 15-3
CALL 63–String to Number Conversion, 15-4
CALL 64–Find a String in a String, 15-5
CALL 65–Replace a String in a String, 15-6
CALL 66–Insert a String in a String, 15-7
CALL 67–Delete a String in a String, 15-8
CALL 68–Find the Length of a String, 15-9
CALL 70-ROM to RAM Program Transfer, 8-1
CALL 71-ROM/RAM to ROM Program Transfer, 8-2
CALL 72-RAM/ROM Return, 8-3
CALL 73-Battery Backed RAM Disable, 5-1
CALL 74-Battery Backed RAM Enable, 5-1
CALL 75–Check SLC 500 Controller CPU Status, 11-6
CALL 77-Protected Variable Storage, 5-2
CALL 78–Set Program Port Baud Rate, 14-2
CALL 80–Check Battery Condition, 11-7
CALL 81-User Memory Module Check and Description, 5-3
CALL 82-Check User PROM Map, 5-4
CALL 84-Transfer DH-485 Interface File to BASIC Input Buffer, 13-5
CALL 85–Transfer Output Buffer to DH–485 Interface File, 12-4
CALL 86–Check DH–485 Interface File Remote Write Status, 11-7
CALL 87–Check DH–485 Interface File Remote Read Status, 11-8
CALL 90-Read Remote DH-485 Data to BASIC Input Buffer, 13-6
CALL 91–Write Output Buffer to Remote DH–485 Data File, 12-5
CALL 92-Read DH-485 Interface File to BASIC Input Buffer, 13-9
CALL 93–Write Output Buffer to Remote DH–485 Interface File, 12-8
CALL 94–Display Current PRT1 Port Setup, 12-11
CALL 95–Get Number of Characters in PRT1 Buffers, 11-9
CALL 96–Clear PRT1 Input/Output Buffers, 12-11
CALL 97–Enable Port PRT2 DTR Signal, 11-10
CALL 98–Disable Port PRT2 DTR Signal, 11-10
CALL 99–Reset Print Head Pointer, 14-3
CBY([expr]), 3-16
Character Set, 1-1
CHR([expr]), 3-14
CLEAR, 6-1
Clear BASIC Module Input and Output Buffers–CALL 120, 11-19
Clear PRT1 Input/Output Buffers–CALL 96, 12-11

Clear PRT2 Input/Output Buffers–CALL 37, 12-1
CLEARI, 6-2
CLEARS, 6-2
CLOCK0, 7-2
CLOCK1, 7-1
CONT, 4-3
Control C, 4-4
Control C Disable-CALL 19, 4-5
Control C Enable-CALL 18, 4-5
Control Q, 4-7
Control S, 4-6
Control Stack, 7-2
Conventions, P-3
Conversion Table, A-1
COS([expr]), 3-7
CPU Input Image Buffer Check–CALL 55, 11-3
CPU Output Image Buffer Check–CALL 51, 11-2

D

DATA, 6-3
Data Types, 2-1
DBY([expr]), 3-17
Delete a String in a String–CALL 67, 15-8
DF1 Driver Communications Disable–CALL 113, 11-18
DF1 Driver Communications Enable–CALL 108, 11-11
DF1 Packet Transmission–CALL 114, 12-12
DF1 XMIT Status Check–CALL 115, 12-13
DH–485 Interface File Remote Read Status Check–CALL 87, 11-8
DH–485 Interface File Remote Write Status Check–CALL 86, 11-7
DIM, 6-4
Disable Port PRT2 DTR Signal–CALL 98, 11-10
Display Current PRT1 Port Setup–CALL 94, 12-11
Display Current PRT2 Port Setup–CALL 31, 12-1
Divide (/), 3-3
DO–UNTIL, 7-4
DO–WHILE, 7-2

E

EDIT, 4-8
Enable Port PRT2 DTR Signal–CALL 97, 11-10
END, 7-4
EOF, 3-15
ERASE, 4-9
EXP([expr]), 3-11
Exponentiation (**), 3-3
Expressions, 3-1
Expressions and Operaters, 3-1

F

Find a String in a String–CALL 64, 15-5
Find the Length of a String–CALL 68, 15-9
Floating–Point Numbers, 2-3
FOR¿TO–(STEP)–NEXT, 7-5
FREE, 3-15
Functional Operators, 3-9
ABS([expr]), 3-9
INT([expr]), 3-9
NOT([expr]), 3-9
PI, 3-9
RND, 3-10
SGN([expr]), 3-10
SQR([expr]), 3-10

G

GET, 13-12
Get DF1 Packet Length-CALL 117, 13-11
Get Number of Characters in PRT1 Buffers–CALL 95, 11-9
Get Number of Characters in PRT2 Buffers–CALL 36, 11-1
Get Numeric Input Character from PRT2-CALL 35, 13-1
Get SLC Processor Program ID Number–CALL 121, 11-20
GOSUB, 8-4
GOTO, 7-7

H

Hierarchy of Operations, 3-2

I

IDLE, 4-9
IF–THEN–ELSE, 7-8
INPL, 13-13
INPS, 13-13
INPUT, 13-14
Insert a String in a String–CALL 66, 15-7
INT([expr]), 3-9
Integer Numbers, 2-3

L

LD@, 13-16
LEN, 3-16
LET, 6-5
LIST, 4-10
LIST #, 4-11
LIST @, 4-11
LOG([expr]), 3-11
Logarithmic Operators, 3-11
  EXP([expr]), 3-11
  LOG([expr]), 3-11
Logical Operations, 3-5
  .AND., 3-5
  .OR., 3-5
  .XOR., 3-5

M

M0 File Check–CALL 58, 11-4
M1 File Check–CALL 59, 11-5
Manual Overview, P-1
MODE, 4-12, 14-4
MTOP, 3-16
Multiply (*), 3-3

N

Negation (–), 3-4
NEW, 4-13
NEXT, 7-9
NOT([expr]), 3-9
NULL, 4-13
Number to String Conversion–CALL 62, 15-3

O

ON–GOTO, 7-10
ON-GOSUB, 8-6
ONERR, 8-5
ONTIME, 8-7
Operators, 3-1
Overflow and Division by Zero, 3-4

P

PHO.,PH1., 12-16
PI, 3-9
POP, 8-10
PRINT, 12-14
Print Argument Stack-CALL 109, 5-7
PRINT CR, 12-15
Print PRT1 Input Buffer and Pointer-CALL 104, 5-6
Print PRT1 Output Buffer and Pointer-CALL 103, 5-5
Print PRT2 Input Buffer Pointer-CALL 111, 5-9
Print PRT2 Output Buffer Pointer-CALL 110, 5-8
PRINT SPC( ), 12-15
PRINT TAB( ), 12-15
PRINT USING(#.#), 12-16
PRINT USING(Fx), 12-15
PROG, 4-14
PROG 1, 4-15
PROG 2, 4-16
Protected Variable Storage-CALL 77, 5-2
PUSH, 8-8
Pushes and Pops, 2-1

R

RAM, 4-18
RAM/ROM Return-CALL 72, 8-3
READ, 13-18
Read DH-485 Interface File to BASIC Input Buffer-CALL 92, 13-9

Read Remote DH-485 Data to BASIC Input Buffer-CALL 90, 13-6
Reference Page Format, P-4
Relational Operations, 3-6
REM, 4-18
REN, 4-19
Replace a String in a String–CALL 65, 15-6
Reset Print Head Pointer–CALL 99, 14-3
Reset PRT1 to Default Settings–CALL 105, 14-3
Reset PRT2 to Default Settings–CALL 119, 14-4
RESTORE, 6-6
RETI, 8-11
Retrieve Date Numeric–CALL 44, 10-4
Retrieve Date String–CALL 52, 10-4
Retrieve Date/Time String–CALL 43, 10-3
Retrieve Day of Week Numeric–CALL 48, 10-7
Retrieve Day of Week String–CALL 47, 10-6
Retrieve Time Numeric–CALL 46, 10-6
Retrieve Time String–CALL 45, 10-5
RETURN, 8-11
RND, 3-10
ROM, 4-20
ROM to RAM Program Transfer-CALL 70, 8-1
ROM/RAM to ROM Program Transfer-CALL 71, 8-2
RROM, 4-21
RUN, 4-22

S

Set Clock/Calendar Date–CALL 41, 10-2
Set Clock/Calendar Time–CALL 40, 10-1
Set Day of Week–CALL 42, 10-3
Set Program Port Baud Rate–CALL 78, 14-2
Set PRT2 Port Parameters–CALL 30, 14-1
SGN([expr]), 3-10
SIN([expr]), 3-7
SLC 500 Controller CPU Status Check–CALL 75, 11-6
SNGLSTP, 4-23
Special Function Operators, 3-15
  # and @, 3-15
  CBY([expr]), 3-16
  DBY([expr]), 3-17
  EOF, 3-15
  FREE, 3-15
  LEN, 3-16
  MTOP, 3-16
  TIME, 3-18
  XBY([expr]), 3-17
SQR([expr]), 3-10
ST@, 12-17
STOP, 8-13
STRING, 15-9
String and Numeric Elementary Data Types, 2-1
String Append–CALL 61, 15-2
String Operators, 3-11
  ASC([expr]), 3-11
  CHR([expr]), 3-14
String Repeat–CALL 60, 15-1
String to Number Conversion–CALL 63, 15-4
Subtract (–), 3-4

T

TAN([expr]), 3-8
TIME, 3-18
Transfer BASIC Output Buffer to CPU Input Image–CALL 54, 12-2
Transfer BASIC Output Buffer to CPU M1 File–CALL 57, 12-3
Transfer CPU Output Image to BASIC Input Buffer-CALL 53, 13-3
Transfer DH-485 Interface File to BASIC Input Buffer-CALL 84, 13-5
Transfer M0 File to BASIC Input Buffer-CALL 56, 13-4
Transfer Output Buffer to DH–485 Interface File–CALL 85, 12-4
Trigonometric Operators, 3-7
  ATN([expr]), 3-8
  COS([expr]), 3-7
  SIN([expr]), 3-7
  TAN([expr]), 3-8

U

UNTIL, 8-4

Upload User Memory Module Code to Host-CALL 101, 5-4

User LED Control–CALL 112, 12-12

User Memory Module Check and Description-CALL 81, 5-3

User PROM Map Check-CALL 82, 5-4

V

Variable Names, 2-5

Variable Types, 2-5

Variables, 2-4

VER, 4-24

W

WARNINGS, CAUTIONS and Important Information, P-2

WHILE, 8-4

Write Output Buffer to Remote DH–485 Data File–CALL 91, 12-5

Write Output Buffer to Remote DH–485 Interface File–CALL 93, 12-8

X

XBY, 8-6

XBY([expr]), 3-17

XFER, 4-20, 4-21, 4-25

ALLEN-BRADLEY
A ROCKWELL INTERNATIONAL COMPANY

As a subsidiary of Rockwell International, one of the world's largest technology companies — Allen-Bradley meets today's challenges of industrial automation with over 85 years of practical plant-floor experience. More than 13,000 employees throughout the world design, manufacture and apply a wide range of control and automation products and supporting services to help our customers continuously improve quality, productivity and time to market. These products and services not only control individual machines but integrate the manufacturing process, while providing access to vital plant floor data that can be used to support decision-making throughout the enterprise.

With offices in major cities worldwide

WORLD HEADQUARTERS
Allen-Bradley
1201 South Second Street
Milwaukee, WI 53204 USA
Tel: (414) 382-2000
Telex: 43 11 016
FAX: (414) 382-4444

EUROPE/MIDDLE EAST/AFRICA HEADQUARTERS
Allen-Bradley Europa B.V.
Amsterdamseweg 15
1422 AC Uithoorn
The Netherlands
Tel: (31) 2975/60611
Telex: (844) 18042
FAX: (31) 2975/60222

ASIA/PACIFIC HEADQUARTERS
Allen-Bradley (Hong Kong) Limited
Room 1006, Block B, Sea View Estate
28 Watson Road
Hong Kong
Tel: (852) 887-4788
Telex: (780) 64347
FAX: (852) 510-9436

CANADA HEADQUARTERS
Allen-Bradley Canada Limited
135 Dundas Street
Cambridge, Ontario N1R 5X1
Canada
Tel: (519) 623-1810
FAX: (519) 623-8930

LATIN AMERICA HEADQUARTERS
Allen-Bradley
1201 South Second Street
Milwaukee, WI 53204 USA
Tel: (414) 382-2000
Telex: 43 11 016
FAX: (414) 382-2400

Catalog Number 1746-NM002 Series A – September 1991

P/N 40063-089-01(A)
Printed in USA