Supplement for
HE800ASC100

SmartStack™ ASCII BASIC Module

First Edition
14 May 1999

SUP0275-01

PREFACE

This manual explains how to use the Horner APG SmartStack™ ASCII BASIC Module.

For user manual updates, contact Horner Advanced Products Group, Technical Support Division, at (317) 916-4274 or visit our website at www.heapg.com.

LIMITED WARRANTY AND LIMITATION OF LIABILITY

Horner APG. ("HE-APG") warrants to the original purchaser that SmartStack™ ASCII BASIC Module manufactured by HE is free from defects in material and workmanship under normal use and service. The obligation of HE-APG under this warranty shall be limited to the repair or exchange of any part or parts which may prove defective under normal use and service within two (2) years from the date of manufacture or eighteen (18) months from the date of installation by the original purchaser whichever occurs first, such defect to be disclosed to the satisfaction of HE-APG after examination by HE-APG of the allegedly defective part or parts. THIS WARRANTY IS EXPRESSLY IN LIEU OF ALL OTHER WARRANTIES EXPRESSED OR IMPLIED INCLUDING THE WARRANTIES OF MERCHANTABILITY AND FITNESS FOR USE AND OF ALL OTHER OBLIGATIONS OR LIABILITIES AND HE-APG NEITHER ASSUMES, NOR AUTHORIZES ANY OTHER PERSON TO ASSUME FOR HE-APG, ANY OTHER LIABILITY IN CONNECTION WITH THE SALE OF THIS SmartStack™ ASCII BASIC Module. THIS WARRANTY SHALL NOT APPLY TO THIS SmartStack™ ASCII BASIC Module OR ANY PART THEREOF WHICH HAS BEEN SUBJECT TO ACCIDENT, NEGLIGENCE, ALTERATION, ABUSE, OR MISUSE. HE-APG MAKES NO WARRANTY WHATSOEVER IN RESPECT TO ACCESSORIES OR PARTS NOT SUPPLIED BY HE-APG. THE TERM "ORIGINAL PURCHASER", AS USED IN THIS WARRANTY, SHALL BE DEEMED TO MEAN THAT PERSON FOR WHOM THE SmartStack™ ASCII BASIC Module IS ORIGINALLY INSTALLED. THIS WARRANTY SHALL APPLY ONLY WITHIN THE BOUNDARIES OF THE CONTINENTAL UNITED STATES.

In no event, whether as a result of breach of contract, warranty, tort (including negligence) or otherwise, shall HE or its suppliers be liable of any special, consequential, incidental or penal damages including, but not limited to, loss of profit or revenues, loss of use of the products or any associated equipment, damage to associated equipment, cost of capital, cost of substitute products, facilities, services or replacement power, down time costs, or claims of original purchaser's customers for such damages.

To obtain warranty service, return the product to your distributor with a description of the problem, proof of purchase, post paid, insured and in a suitable package.

ABOUT PROGRAMMING EXAMPLES

Any example programs and program segments in this manual or provided on accompanying diskettes are included solely for illustrative purposes. Due to the many variables and requirements associated with any particular installation, Horner APG cannot assume responsibility or liability for actual use based on the examples and diagrams.

It is the sole responsibility of the system designer utilizing the SmartStack™ ASCII BASIC Module to appropriately design the end system, to appropriately integrate the SmartStack™ ASCII BASIC Module and to make safety provisions for the end equipment as is usual and customary in industrial applications as defined in any codes or standards which apply.

Note: The programming examples shown in this manual are for illustrative purposes only. Proper machine operation is the sole responsibility of the system integrator.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 Scope

The supplement for the SmartStack™ ASCII BASIC Module provides information that is pertinent to the setup and operation of the HE800ASC100 (ASC100) module.

Chapter 1 covers the features and hardware description of the ASC100 module. Installation procedures are covered in Chapter Two. Chapter Three provides a BASIC programming overview. Chapter Four covers BASIC Commands and Statements. Chapters Five-Seven provide information on arithmetic and relational operators, string handling and error handling. Chapter Eight covers the PLC Interface as well as configuration procedures for the ASC100 module using Cscape Software. Chapter Nine provides procedures to enter, edit, store and execute an ASCII BASIC program. Appendices A-E cover various topics such as serial port and modem wiring, reserved word list, configuration jumpers, ASCII character set, memory configuration, and use of Terminal Emulation Software.

Installation and configuration procedures that are common to all SmartStack Modules are contained in the Control Station Hardware Manual (MAN0227).

1.2 ASCII BASIC Module Features

The ASCII BASIC Module is capable of performing powerful functions typically reserved for more expensive, mid-sized Programmable Logic Controllers (PLCs). The ASCII BASIC module allows more flexibility for the system designer in applications where the module is used as a stand-alone microcomputer or where information is passed between the Operator Control Station/Remote Control Station (OCS/RCS) and the module.

1. Programmed via the BASIC programming language - versatile instruction set.

2. Sixty-four 16-bit input registers and sixty-four 16-bit output registers interfacing the ASCII BASIC module to the OCS/RCS.

3. Powerful floating-point math instructions including logarithmic and trigonometric functions.

4. Primary RS-232 communication port for connection to a dumb terminal or host computer for program development.

5. Secondary RS-232/RS-485 communication port for connection to an operator interface terminal, printer, etc.

6. Single slot usage, low power consumption, typically less than 130 mA (180 mA max) at 5VDC.

7. Asynchronous program execution.

1.3 Hardware Description

The ASC100 module utilizes state-of-the-art electronic components on a six-layer copper-clad printed circuit board for electrically quiet operation. Two important precautions must be observed while handling the module:

WARNING

NEVER insert or remove the module into or out of the OCS/RCS unit while power is applied to the backplane. If this practice is repeated, eventually the module WILL BE DAMAGED.

WARNING

ALWAYS observe reasonable static discharge precautions while handling the module. Touch a grounded metal surface to discharge any static buildup before touching the module.

1.3.1 Microprocessor

At the heart of the ASCII BASIC Module lies the Dallas 80C320 microprocessor running at 22.1184 Megahertz. This configuration yields an instruction execution time of slightly more than six million instructions per second (at the assembly level). Internal to this chip are 256 bytes of user memory (most of which are used by the ASCII BASIC firmware). The 80C320 can address up to 64 Kilobytes of external CODE memory (this is where the firmware resides), and up to 64 Kilobytes of external DATA memory (this space is divided between DATA and PROGRAM space for the ASCII BASIC module).

1.3.2 Module Reset Options

The 80C320 microprocessor is equipped with a RESET signal that, when active, inhibits all processing activity. This RESET signal is generated for a short time immediately following power-up of the host OCS/RSC system. Reset can be simulated in software using the RESET command.

1.3.3 Primary Serial Port

The PRIMARY port located on the front of the ASCII BASIC Module incorporates a 9-pin "D" type connector for standard cable interface (See **Appendix A** for wiring diagrams). This port features automatic baud rate detection and is used for program entry, editing and debug. It can also be referenced from within the BASIC program during execution.

There are two LED's (Light Emitting Diodes) located on the end of the module. They are labeled according to the RS-232 signal name to which they are connected. The GREEN LED illuminates whenever data is transmitted from the BASIC while the RED LED illuminates whenever data is received by the BASIC module.

1.3.4 Flexible Memory Configuration

As stated before, the 80C320 can address up to 128 Kilobytes of external memory. This memory is divided among 3 devices, and is configured at the factory (See **Appendix D** for a discussion of the memory map configuration).

1.3.5 Firmware Memory

The firmware site consists of a 64 Kilobyte FLASH EPROM mapped to the 80C320's CODE space. The software in this site is a miniature operating system controlling user program input and execution.

1.3.6 Data Memory

The DATA site is equipped with a 32K static RAM device supported by a battery-backed controller. This socket also contains the real-time clock hardware. The lowest 1536 bytes of this memory are reserved for the ASCII BASIC interpreter. The remaining DATA memory is used for all variable storage and for BASIC program number 0 entry and editing. See **Appendix D** for a more complete discussion of the DATA FILE memory.

1.3.7 Program File Memory

The PROGRAM site is equipped with a 32K EEPROM device. Unlike the DATA site, the PROGRAM site can also be EMPTY. In this case, the DATA site is divided between DATA and PROGRAM FILE memory. See **Appendix D** for a more complete discussion of the PROGRAM FILE memory.

1.3.8 PLC Interface

Proprietary circuitry is used in the interface between the ASCII BASIC Module and the OCS/RCS. This circuitry provides up to 128 WORDS (sixty-four 16-bit AI input and sixty-four 16-bit AQ output) for both the OCS/RCS and the 80C320. Circuitry and software are provided in the firmware to insure data integrity on both sides.

1.3.9 Secondary Serial Port

The ASCII BASIC Module is equipped with a secondary serial port. This port is multiplexed between RS-232 and RS-485. The pin-outs/connections for both ports are shown in **Appendix A**.

The module also has two LED's located on the module's front panel behind the plastic window (on the end of the module) for each auxiliary port. They are labeled according to the RS-232 or RS-485 signal name to which they are connected. The GREEN LED illuminates whenever data is transmitted from the BASIC MODULE while the RED LED illuminates whenever data is received by the ASCII BASIC MODULE. For the modem option, the GREEN LED represents off-hook and the RED LED represents on-hook line activity (ring).

1.4 Specifications

	ASC100			ASC100
Number of communication ports	3 (2 RS232, 1 RS485/422		User Memory for BASIC ***program*** storage	32K bytes
Status LEDs	RXD/TXD for all ports RUN (BASIC program)		User Memory for BASIC ***data*** storage	32K bytes

General Specifications			
Required Power (Steady State)	34.0mA @ 5VDC	IEC Rating	Pending
Required Power (Inrush)	89.6mA @ 5VDC in 10µS	UL	Pending
Relative Humidity	5 to 95% Non-condensing	Terminal Type	9-Pin D-Subs
Operating Temperature	0° to 60° Celsius	Weight	9.5 oz. (270 g)

CHAPTER 2: INSTALLATION

2.1 Module Placement

The ASC100 is one of the many SmartStack Option Modules that are available for use in the Operator Control Station (OCS100/OCS200) and the Remote Control Station (RCS210). Compact and easy to install, up to four Smart Stack Modules can be placed into each device. Although Chapter Two contains pertinent installation information for the ASC100, consult the Control Station Hardware User Manual (GFK-1631) for complete details covering panel box requirements and other key installation issues pertaining to SmartStack Option Modules, OCS, and RCS devices.

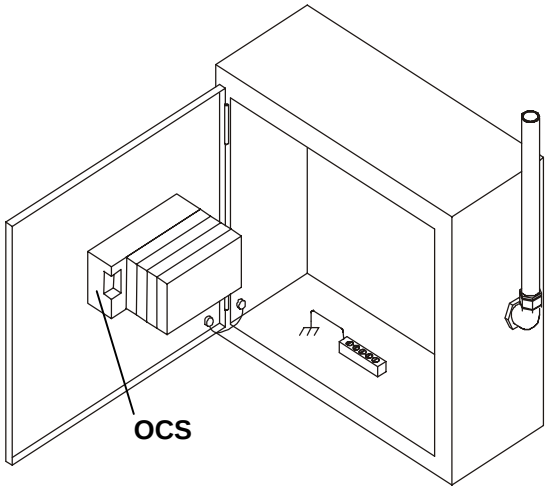

Figure 2.1 - Back View of OCS (Shown with Four SmartStack™ Modules)

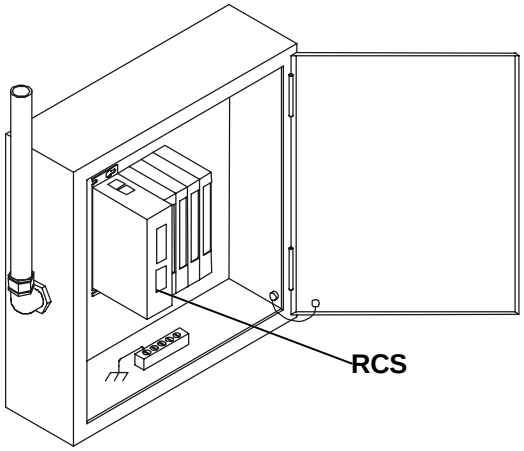

Figure 2.2 - RCS in Panel Box (Shown with Four SmartStack™ Modules)

Caution: Do not install more than four SmartStack Modules per OCS or RCS. Improper operation or damage to the OCS, RCS, and SmartStack Modules could result.

2.2 Ports and Pin-outs

Note: Port 1 and Port 2 share internal communication circuitry. Only one of the ports can be used to transfer data at a time.

Table 2.1 – Port 0/1 Pin-out		
Direction	**Pin**	**ASC100 Port 0/1**
Output	1	**DCD** (Data Carrier Detect)
Output	2	**RXD** (Receive Data)
Input	3	**TXD** (Transmit Data)
Input	4	**DTR** (Data Terminal Ready)
GND	5	**Signal Ground**
Output	6	**DSR** (Data Set Ready)
Input	7	**CTS** (Clear to Send)
Output	8	**RTS** (Request to Send)
Output	9	**RI** (Ring Indicate)

Note: For ports 0 and 1, the signal names reflect the EIA RS232 signal names for a DCE device.

The names do not necessarily reflect the signal direction with respect to the ASC100 module.

Table 2.2 – Port 2 Pin-out		
Direction	**Pin**	**ASC100 Port 2**
Input	1	**RXD-** (Receive Data -)
Output	2	**TXD-** (Transmit Data -)
Output	3	**CTS-** (Clear to Send -)
Input	4	**RTS-** (Request to Send -)
GND	5	**GND** (Signal Ground)
Input	6	**RXD+** (Receive Data +)
Output	7	**TXD+** (Transmit Data +)
Output	8	**CTS+** (Clear to Send +)
Input	9	**RTS+** (Request to Send +)

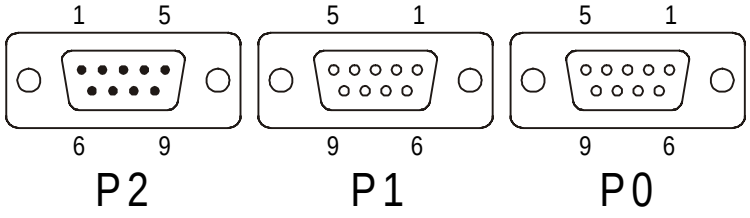

Figure 2.3 – Close-up of Port Connectors

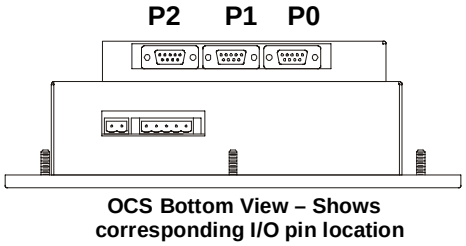

Figure 2.4 – Port Connectors

2.3 Installing and Removing a SmartStack Module (Shown with the OCS)

The following section describes how to install and remove a SmartStack Module.

Caution: To function properly and avoid possible damage, do not install more than four Smart Stack™ Modules per OCS or RCS.

a. Installing SmartStack Modules

1. Hook the tabs. Each SmartStack Module has two tabs that fit into slots located on the OCS. (The slots on the OCS are located on the back cover.)
2. Press the SmartStack Module into the "locked" position, making sure to align the SmartStack Module fasteners with the SmartStack receptacles on the OCS.

b. Removing SmartStack Modules

1. Using a flathead screwdriver, pry up the end of the SmartStack Module (opposite of tabs) and swing the module out.
2. Lift out the tabs of the module.

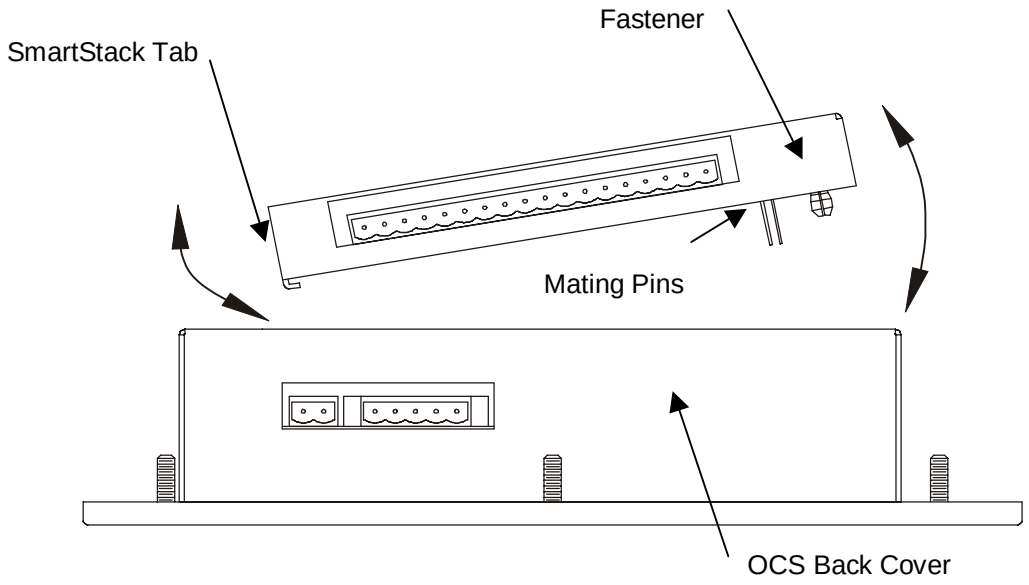

Figure 2.5 – Installing a SmartStack™ Module in an OCS.

2.4 Configuration Procedures Using Cscape Software

2.4.1 Preliminary Configuration Procedures

The SmartStack configuration is accomplished through the Configure Controller Type Dialog.

1. From the Main Menu, select Controller|Configure for the following dialog:

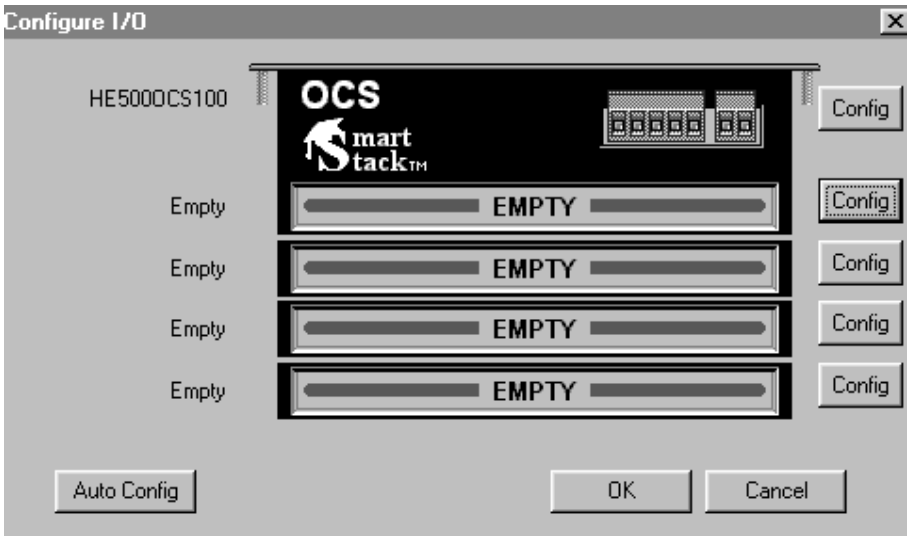

Figure 2.6 – Configure Controller Type Dialog

Note: Ensure that the proper controller is selected. If it is not selected, double-click on the box and select the desired controller from the pull-down menu. Press the OK button .

If the OCS/RCS has one or more SmartStack™ Modules already installed, the user can use AutoConfig to determine and set the I/O automatically. If the controller is not physically attached to Cscape or its SmartStack™ Modules are not available, use the manual configuration.

The user can choose to do the following:

a. Add a SmartStack I/O Module

Caution: To function properly and avoid possible damage, do not install more than four Smart Stack™ Modules per OCS or RCS.

1. To place a SmartStack I/O module into an empty slot, ADD the module. From the CONFIGURE I/O Dialog, click on the CONFIG button to the right of the desired module or double-click on the empty slot. Either method invokes the SmartStack Module Selection Dialog:

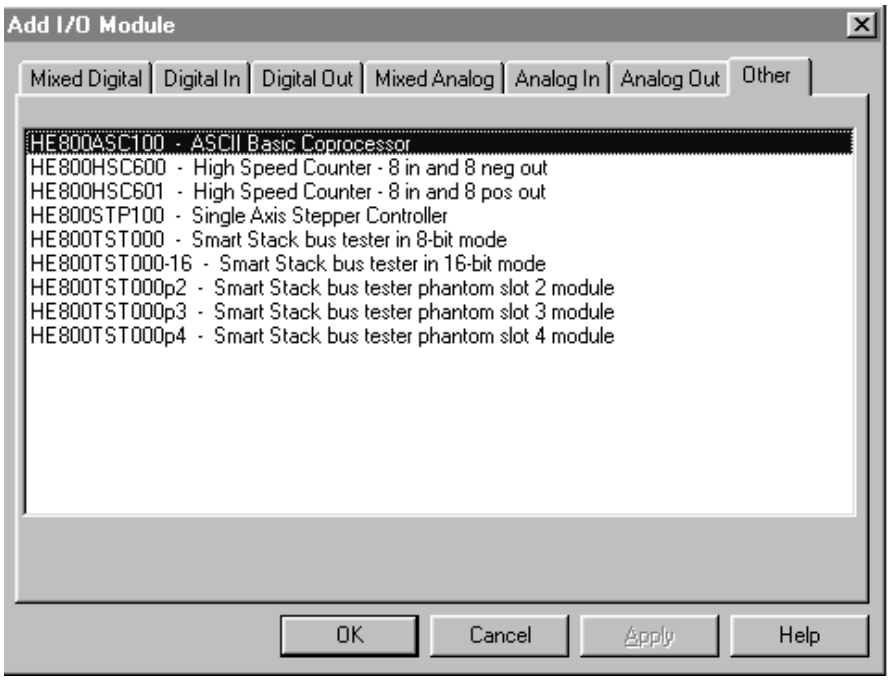

Figure 2.7 – Add I/O Module Screen
("Other" selections are shown.)

2. Use the mouse to select the type of module desired. The ASC100 is a specialty module. Select the **Other** tab. Select the desired module, and click the OK button.

b. Delete a SmartStack Module

If the desired SmartStack "slot" shows a module already installed, the module can be deleted.

1. Right-click on the picture of the configured "slot". A floating menu appears.
2. From the menu, click on DELETE MODULE.

c. Select a Different SmartStack Module

If the desired SmartStack "slot" shows a module already installed, the module can be replaced with a different module.

1. Right-click on the picture of the configured "slot". A floating menu appears.
2. From the menu, click on 1. REPLACE MODULE. This invokes the SmartStack Module Selection Dialog.
3. Use the mouse to select the desired module and then click OK .

2.4.2 Configuration of ASC100 Module

The screen now depicts the controller and ASC100 SmartStack Module that has been chosen by the user. The desired module is ready to be configured.

1. Double-click on the picture of the module or click on the Config button  just to the right of the picture.

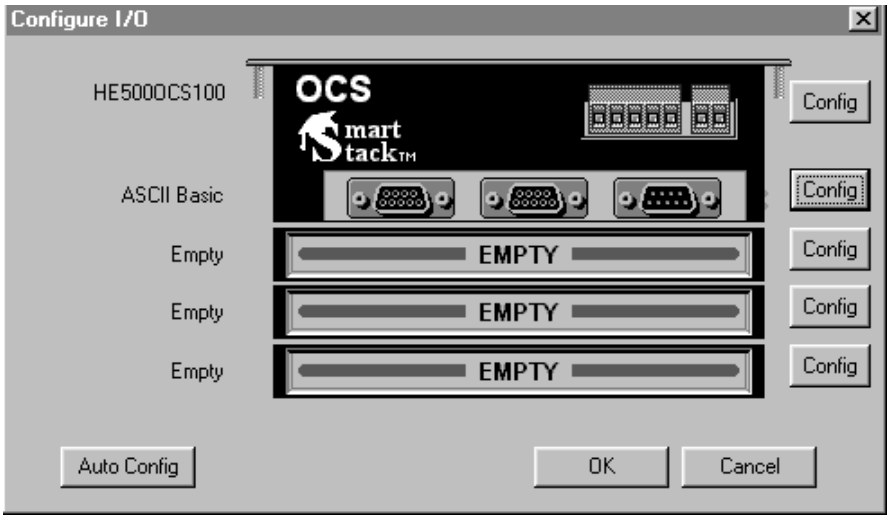

Figure 2.8 – Configure I/O Screen

The following screen appears:

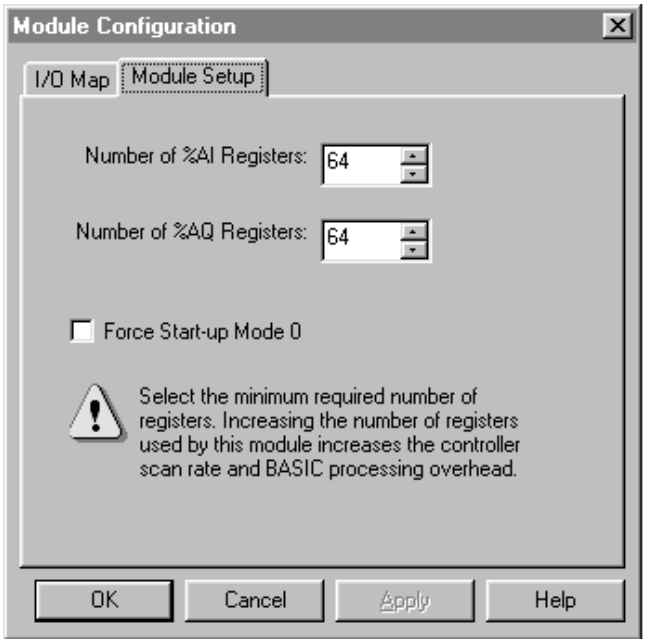

Figure 2.9 – Module Configuration Screen

Select the minimum required number of registers desired. Note that increasing the number of registers used by the module increases the controller scan rate and BASIC processing overhead.

For Cscape configuration purposes, check the box labeled, **Force Startup Mode 0** to force the module into **Startup Mode 0** during the next power-up. (See Chapter Four: Startup Command.)

3. Press OK. The following screen appears

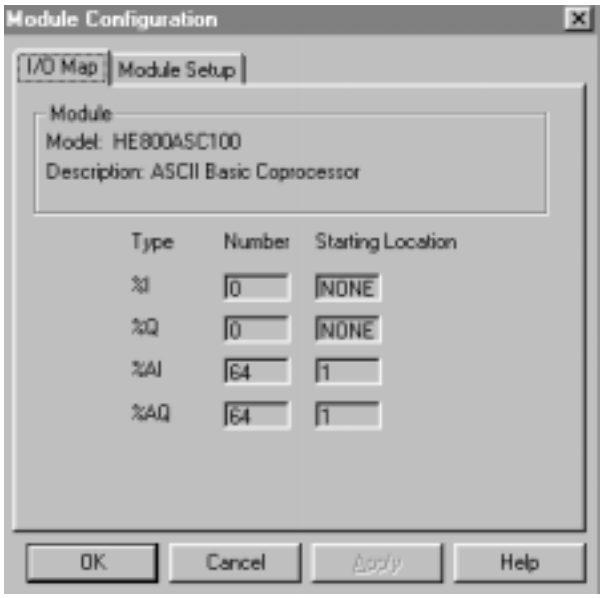

Figure 2.10 – I/O Screen

2.5 The Console Device

To program the ASCII BASIC Module, the user must connect a console device to the primary RS-232 port This device can be either a dumb terminal or a host computer running terminal emulation software Cable wiring diagrams is located in **Appendix A** The console device must be configured to a baud rate of 50 to 38,400 baud with no parity, 8 data bits and one stop-bit Software (or XON/XOFF) handshaking is implemented by the ASCII BASIC Module's primary port upon initial power-up Once connected, follow the steps to initialize communications with the module:

1. Apply power to the OCS/RCS system.

Note: Ensure the OCS/RCS is **NOT** in the **RUN** mode.

2. Press the SPACE bar on the console device. The ASCII BASIC Module automatically determines the baud rate at which the space character (ASCII 32) was received and responds with a full screen of sign-on/status information:

If no response is obtained or if the module responds erratically, recheck the cable wiring and communication parameters. Power cycle and try again.

The prompt characters **0>** are issued by the ASCII BASIC Module to indicate that it is in **Command** mode and is ready to accept commands. The **0** indicates that program number 0 is currently selected.

2.5.1 Using a Host Computer

A host computer can be used as the console device if a terminal emulation software program is available (such as ProComm by DataStorm). There are two important points to be aware of when using such programs:

a. Some terminal emulator programs send out characters when they are invoked. If the ASCII BASIC module receives any character other than a space character (ASCII 32), the baud rate is incorrectly calculated and communications are not possible To avoid this problem, configure and initialize the terminal emulation program before applying power to the ASCII BASIC Module. Then, press the space bar.

b. Some terminal emulator programs do not support handshaking. This means that it is possible for the ASCII BASIC Module to send data to the console device much faster than the host computer can process it This can cause lost data, erroneous display of characters or even computer lockup If software handshaking is not an available option for the terminal emulation software, use a lower baud rate (to allow the terminal emulator program more time to process each character).

2.5.2 TERM - Dumb Terminal Emulation Software

Included on the distribution diskette is a terminal emulation program called TERM.EXE. This program can be loaded and run on most IBM PCs or compatible computers. This program was designed and written specifically for communication with an ASCII BASIC Module and provides the following features:

1. Software (XON/XOFF) and hardware (RTS/CTS) handshaking capability.

2. Capable of communication rates of 110 to 57,600 baud.

3. Complete program UPLOAD and DOWNLOAD capability at any baud rate (Programs created with the user's favorite word processor can be downloaded).

CHAPTER 3: BASIC PROGRAMMING OVERVIEW

3.1 What is BASIC?

BASIC is an acronym for "Beginner's All-purpose Symbolic Instruction Code". It was created in 1964 by two professors at Dartmouth University as a tool to teach the fundamentals of computer programming. It is an interactive "interpreted" language, ideal for this industrial application. Those already familiar with the BASIC language will have little difficulty programming the ASCII BASIC Module.

This manual does not cover how to write programs in BASIC. Commands and statements available in the ASCII BASIC Module are described and demonstrated in examples.

3.2 Operating Modes

The ASCII BASIC Module operates in two states or modes.

3.2.1 COMMAND MODE:

Active whenever the prompt character **>** is present to signify that the module is ready to accept commands and statements from the console device. No BASIC program is currently being executed. The ASCII BASIC Module takes immediate action when a command is entered.

3.2.2 RUN MODE:

Active whenever an ASCII BASIC program is currently being executed. Commands can not be entered until the program is halted.

Some of the commands and statements can only be entered while in COMMAND mode while others can only be entered on BASIC program lines. Some can be used in both modes. The description of each command and statement indicates when it is used.

3.3 BASIC System Elements

3.3.1 Stack Structure

A stack is a dedicated area of memory used to store important information regarding program control and expression evaluation. The ASCII BASIC Module incorporates the use of two software stacks.

3.3.2 Control Stack

The CONTROL STACK is used to store information regarding program control. The FOR-NEXT, DO-WHILE, and GOSUB-RETURN statements stores information on the control stack for use at the bottom of each loop or iteration. If too many of these statements are active or nested at one time, a Control stack error results.

3.3.3 Argument Stack

The ARGUMENT STACK is used to store information while the module evaluates complex expressions. The PUSH and POP statements also make use of the ARGUMENT STACK. If too many values are placed on the ARGUMENT STACK or the POP instruction is executed when no data is on the stack, an Argument stack error is generated.

3.3.4 The Line Editor

An ASCII BASIC command or program line can contain up to 79 characters. If an attempt is made to enter more that 79 characters, the BELL character (ASCII 7) is transmitted from the module and the characters beyond the 79th are ignored.

During line entry, the BACKSPACE character (ASCII 8) can be used to perform a rubout operation. This causes the last character entered to be erased from the line while the cursor is placed at the position of the deleted character. If there are no characters to rubout when the backspace key is pressed, a BELL character (ASCII 7) is sent from the module and the rubout is ignored.

Once a line has been entered (CARRIAGE RETURN has been pressed), the program line can no longer be edited. If any changes are to be made to the program line, the entire line must be reentered.

Blanks or spaces imbedded in statements (except for those in quoted strings and in REM statements) are ignored by the ASCII BASIC Module. However, during the LISTing of programs, the module inserts spaces to improve program readability.

STOP! If a CONTROL-S (ASCII 19) is inadvertently entered while the primary port is configured for XON/XOFF handshaking, the module appears to lockup. This is because the CONTROL-S character is the XOFF signal, which causes the module to cease transmission until a CONTROL-Q (ASCII 17) is received. If this symptom occurs, try pressing CONTROL-Q to resume module transmission.

3.4 BASIC Program Elements

3.4.1 Executable Statements

An ASCII BASIC program is comprised of statements. Every statement begins with a line number, followed by the statement body, and terminated with a CARRIAGE RETURN <CR> (or a colon ":" in the case of multiple statements per line).

3.4.2 Line Numbers

Every ASCII BASIC program line must begin with a line number ranging from 0 to 65535 inclusive. Line numbers are used to order the program sequentially. In any one program, a line number can only be used once. Lines need not be entered in numerical order, because the ASCII BASIC Module automatically orders them in ascending order. For example, if the following program is entered non-sequentially:

```
example     0>10 PRINT "This is line 10"
            0>30 PRINT "This is line 30"
            0>20 PRINT "This is line 20"
            0>RUN

            This is line 10
            This is line 20
            This is line 30
```

Notice that when the program was RUN, it was executed in numeric order and not in the order that the statements were entered.

More than one statement can be placed on a single line if each statement is separated by a colon ":". Only one line number can be used for a single line. For example:

```
example     0>10 PRINT "This is line 10" : PRINT "This is also line 10"
            0>RUN

            This is line 10
            This is also line 10
```

If a line number is entered that already exists, the new line replaces the existing line. Therefore, to remove a line from a program, simply enter the line number of the line to be deleted followed by a CARRIAGE RETURN <CR>.

3.4.3 BASIC Programs

BASIC programs are made up of one or more BASIC statements with each having a unique line number. When in COMMAND mode, the BASIC program lines are entered via the console device. Up to 255 programs can be stored in the ASCII BASIC module's memory. Note, however that only program number 0 can be edited. Program number zero is stored in the DATA memory, all other programs are stored in the PROGRAM FILE memory. PROGRAM FILE programs can be transferred into program 0 using the EDIT command and then re-saved in the PROGRAM FILE. The COMMAND mode prompt specifies which BASIC program is currently SELECTed.

3.4.4 Numeric Values

The ASCII BASIC Module is capable of manipulating numbers in four formats: Decimal integer (1234), hexadecimal integer (89ABH), fractional floating-point (12.34) and exponential floating-point (12.345678 E+5).

3.4.5 Integer Values

Integers require two bytes of memory storage. There are several occasions when integer values are required. In these cases, if a floating point value is used, it is truncated to an integer or an error is generated.

Hexadecimal integers must always begin with a valid digit (0 through 9). For example, A0H is entered as 0A0H.

3.4.6 Floating-point Values

Each floating point value requires six bytes of memory storage. The module rounds all floating point numbers to eight significant digits.

Exponential floating point values can range from +/- 1E-127 to +/- 99999999E+127.

3.4.7 Numeric Constant Values

Some commands and statements require the use of a CONSTANT argument. This means that a variable or expression is not allowed. Constants can be floating point values, but some cases require integers.

3.4.8 Operators

An operator performs a predefined operation on variables and/or constants. Operators require either one or two operands. Typical two operand or DYADIC operators include addition (+), subtraction (-), multiplication (*) and division (/). Operators that require only one operand are often referred to as UNARY operators and include SIN(), COS() and ABS().

3.4.9 Variables

A VARIABLE is an area of memory that is referenced in BASIC statements by a user-defined NAME. Values can be assigned to the variable, and the variable's value can at any time be obtained.

Variable names must start with a letter (A to Z) and can contain up to 8 letters or numbers (including the underscore character "_"). The following are examples of valid variable names:

FRED VOLTAGE1 I3 AIR_CYL

Variables are allocated in a static manner. This means that each time a new variable is defined, BASIC allocates a portion of memory (8 bytes) specifically for that variable. This memory can not be de-allocated on a variable by variable basis. For example, if a statement is executed like "Q = 3", the user can not later tell BASIC that the variable "Q" no longer exists and to free up the 8 bytes that are allocated to Q. The only way to clear the memory that is allocated to a variable is to execute a CLEAR statement. This frees up all memory allocated to ALL variables.

STOP! Three very important anomalies need to be observed when defining variable names:

1. It takes BASIC longer to process variables whose names are greater than two characters in length.

2. Only the first character, the last character and the number of characters in the variable name are significant. This means that the following variable names refer to the same memory space and are in essence the same variable, because they all start with "I", end with "R" and contain 7 characters.

   IN_CHAR ILLFOUR INCDOOR

3. The user CAN NOT USE ANY BASIC KEYWORD as part of a variable name. A BAD SYNTAX error is generated if the user attempts to use a BASIC reserved word as part of a variable name. The following variable names are invalid:

   TABLE (uses TAB) ONES (uses ON) CRABS (uses ABS)

   See **Appendix B** for a list of all BASIC reserved words.

3.4.10 Array Variables

The variables described up to this point are called SCALAR variables. Each variable name refers to only one 8-byte memory entity. Variables can include a ONE DIMENSION subscript expression (ranging from 0 to 254) enclosed in parentheses. This type of variable is referred to as a dimensioned or array variable. For example, an array called MNTH can be used to contain the number of days in each month. The following program segment illustrates:

example

```
0>10 DIM MNTH(13) : REM Tells BASIC how much space to allocate for the MNTH array.

        0>20 MNTH(1) = 31
        0>30 MNTH(2) = 28
        0>40 MNTH(3) = 31
        0>50 MNTH(4) = 30
        0>60 MNTH(5) = 31
        0>70 MNTH(6) = 30
        0>80 MNTH(7) = 31
        0>90 MNTH(8) = 31
        0>100 MNTH(9) = 30
        0>110 MNTH(10) = 31
        0>120 MNTH(11) = 30
        0>130 MNTH(12) = 31
        0>140 FOR X = 1 TO 12
        0>150 PRINT "There are ", MNTH(X), " days in month ", X
        0>160 NEXT X
```

3.4.11 Numeric Expressions

An expression is a logical mathematical formula that involves OPERATORS, CONSTANTS, and/or VARIABLES. Expressions can be simple or quite complex.

```
Example     12 * EXP(A) / 100
            H(1) + 55
            (SIN(A) * SIN(A) + COS(A) * COS(A) ) / 2
```

A stand-alone variable or constant is also considered an expression.

3.4.12 Relational Expressions

Relational expressions involve the operators EQUAL (=), NOT EQUAL (<>), GREATER THAN (>), LESS THAN (<), GREATER THAN OR EQUAL (>=), and LESS THAN OR EQUAL (<=). They are used in control statements to test a condition.

```
Example         10 IF A<100 THEN B=5
```

Relational expressions ALWAYS require two numeric or string expressions.

3.4.13 String Expressions

String expressions are expressions that yield a character string result. Strings are fully discussed in Chapter 6.

3.4.14 Special Function Operators

The special function operators available to the ASCII BASIC programmer are discussed in Chapter 6. These operators are used to assign and/or obtain values of predefined special values.

3.5 Manual Conventions

The following conventions are used in the remainder of this manual:

Stop! Special attention should be paid to the text following this symbol. If caution is not used, irretrievable damage can be done to the module.

COMMAND The command associated with this symbol can be used in COMMAND MODE.

RUN The command associated with this symbol can be used in RUN MODE.

expr Numeric expression, a logical mathematical formula that involves operators, (both unary and dyadic), constants, and/or numeric variables. A "stand-alone" variable or constant is also considered an expression.

integer Numeric integer, Integers used by the ASCII BASIC module are whole numbers that range from 0 to 65535 inclusive.

const Numeric constant, a real number that ranges from +/- 1 E-127 to +/- .99999999 E+127. A constant can be an integer.

line_num BASIC line number, an integer value that refers to or assigns a BASIC program line number.

string_expr String expression, a logical string formula that involves string operators, quoted strings, and/or string variables.

[] Optional parameter, objects appearing in square brackets are optional parameters and can be omitted.

parameter A parameter is an argument required by a BASIC operator or statement. Parameters always appear in italic print and are fully described in the text for the operator or statement.

CHAPTER 4: COMMANDS AND STATEMENTS

4.1 System Commands

The commands described in this section can only be executed while in **Command** mode. Any attempt to use these commands on a BASIC program line causes an Invalid syntax error.

The following commands are discussed in this section:

AUTORUN	**EDIT**	**NULL**	**STARTUP**
BREAK@	**HELP**	**RESET**	**STATUS**
CONT	**LIST**	**RUN**	**STEP**
DELPGM	**LIST#**	**SAVE**	
DIAG	**NEW**	**SELECT**	

AUTORUN

SYNTAX : AUTORUN *integer*
MODE: Command

The AUTORUN command is used to configure the program number that the ASCII BASIC Module automatically executes following a power-up or RESET condition. The *integer* is a numeric constant that refers to the program number stored in the PROGRAM file memory. The *integer* can be any value between 0 and 254 inclusive. Note that the module must be placed in STARTUP mode 2 before it runs the specified program following RESET.

If the integer value is zero, the program in DATA memory (program 0) is executed following a RESET. If this is desired, the CLRMEM 0 statement must be used to disable the DATA memory initialization, retaining program 0 in DATA memory.

If the specified program does not exist following RESET, the module defaults to STARTUP mode 1, immediately displaying the sign-on/status message and entering COMMAND mode.

SEE ALSO: STARTUP, RESET, CLRMEM

BREAK@

SYNTAX : BREAK@ *line_num*
MODE: Command

The BREAK@ command is used to set a breakpoint on a BASIC program. Setting the breakpoint does not alter the program. It merely configures the command interpreter to HALT whenever the specified *line_num* is executed (similar to the STOP statement). The BREAK@ command has a significant advantage over the STOP statement in that the breakpoint can be set without modifying the BASIC program. Insertion of the STOP statement requires program modification, which makes the CONT command invalid until the program is restarted. Using the BREAK@ command, the user can configure a breakpoint and then execute the CONT command.

Example

```
0>LIST
10        PRINT "This is line 10"
20        PRINT "This is line 20"
30        PRINT "This is line 30"
40        PRINT "This is line 40"
50        PRINT "This is line 50"
60        GOTO 10

Ready
0>BREAK@30

Ready
0>RUN
This is line 10
This is line 20

BREAK - In line 30
Ready
0>BREAK@10

0>CONT
This is line 30
This is line 40
This is line 50
BREAK - In line 10
```

Only one breakpoint can be active at any given time. If more than one breakpoint is required, then STOP statements can be used. Note that when the program is halted due to breakpoint, BREAK is displayed prior to the execution of the line number.

SEE ALSO : CONT, STEP, STOP

CONT

SYNTAX : CONT
MODE: Command

If an executing program is stopped by typing a CONTROL-C on the console device or by the execution of a STOP statement, program execution can be resumed from where it was interrupted by entering the CONT command. While program execution is halted, the value of variables can be examined and/or modified. The CONT command can not be used if the program has been modified or if the program was terminated due to an ERROR.

example

```
0>10      FOR I = 1 TO 1000
0>20      PRINT I
0>30      NEXT I
0>RUN
1
2
3                    <CONTROL-C TYPED ON CONSOLE DEVICE>

STOP!  In Line 20
Ready
0>PRINT I
6

0>I=9999

0>CONT
9999
10000
```

SEE ALSO : BREAK@, STEP, STOP

DELPGM

SYNTAX 1 : DELPGM *integer*
SYNTAX 2 : DELPGM *
Mode: Command

The DELPGM command is used to erase one of the programs from the PROGRAM file memory. The *integer* is a numeric constant that refers to the program number stored in the PROGRAM file memory. The *integer* can be any value between 0 and 254 inclusive.

If the *integer* value refers to a nonexistent program in the PROGRAM file memory, a "Program does not exist" error message is generated. If the *integer* value is zero, the program in DATA memory is erased. This is exactly the same as entering the **NEW** command.

If the erased program was followed by one or more programs in the PROGRAM file, the subsequent programs are shifted by one to fill the gap. For example, if six programs are stored in the PROGRAM FILE and the user erased program number is 3 using the DELPGM command, programs 4, 5 and 6 are moved and are now accessed as programs 3, 4 and 5 respectively.

Program 0 is SELECTed following a DELPGM command.

When an asterisk "*" is used as the argument to the DELPGM command, ALL of the programs stored in the PROGRAM file memory are deleted. The module displays the following prompt prior to erasing the programs:

Are you sure? (Y/N)

If a "Y" is entered, all programs in the PROGRAM file are erased. If any other key is pressed in response, the DELPGM command is ignored and the module returns to command mode. Note that the DELPGM * command does not affect program 0 in DATA memory.

SEE ALSO : EDIT, SAVE, SELECT

DIAG

SYNTAX : DIAG
MODE: Command

The DIAG command invokes the ASCII BASIC Module's firmware diagnostic routine. These diagnostic routines functionally test most of the circuitry on the ASCII BASIC Module.

When the DIAG command is entered, the module responds with the following message:

```
ASCII Basic Module Firmware Diagnostics - V 3.00
(c) Copyright 1991-1995 Horner Electric, Inc.

The diagnostics will run continuously until any key is pressed.
Press Y to begin...
```

If the user types any key other than "Y", the ASCII BASIC module returns to command mode, and the DIAG command is ignored. If the user types "Y" in response to the DIAG prompt, the firmware diagnostic routines runs. The result of each test is displayed as it is executed. When all tests have completed, the tests are restarted from the beginning.

To terminate the diagnostic test execution, the user must simply press any key. At that point, the module behaves as though it had just been reset. Note that any programs stored in the DATA memory are lost.

EDIT

SYNTAX : EDIT [*integer*]
MODE: Command

The EDIT command transfers the program specified by the *integer* into program 0 (DATA memory) so that it can be edited. If the *integer* is omitted, the currently selected program is transferred.

If program 0 (in DATA memory) exists when the EDIT command is issued, it is overwritten by the transferred program.

This command is most often used to place a PROGRAM FILE program into program 0 in DATA memory for editing and debugging.

HELP

SYNTAX : HELP [*keyword*]
MODE: Command

The ASCII BASIC module incorporates a very useful ON-LINE HELP system. If HELP is entered with no argument, a full screen of information is displayed containing the HELP syntax and all of the BASIC keywords implemented by the module.

If a *keyword* is specified following the HELP command, specific usage and syntax information is displayed pertaining to the BASIC *keyword*.

Note that keywords should be entered EXACTLY as they appear in the HELP screen, including parenthesis if required.

If an unrecognized *keyword* is entered following the HELP command, the HELP screen is displayed.

LIST

SYNTAX : LIST [*#*] [*start_line_num*] [*-end_line_num*]
MODE: Command

The LIST command prints the current program to the console device. Note that the list command "formats" the program in an easy to read manner. Spaces are inserted after the line number and before and after statements. This feature is designed to aid in the debugging of ASCII BASIC programs.

The LISTing of a program can be terminated at any time by typing a CONTROL-C character on the console device (unless the BREAK 0 option is in force).

If software handshaking (XON/XOFF) is being used, a LISTing can be paused by typing a CONTROL-S character on the console device, and resumed by typing a CONTROL-Q.

If a *start_line_num* is specified, the program is listed starting with the *start_line_num* and continuing to the end of the program.

If a *start_line_num* and an *end_line_num* are specified, the program is listed starting with the *start_line_num* and continuing through the *end_line_num.*

If the # is used then the program listing is directed to the designated AUXILIARY port.
SEE ALSO : SETCOM

NEW

SYNTAX : NEW
MODE: Command

When the NEW command is entered, the ASCII BASIC Module deletes program 0 in DATA memory. All variables are set to zero, and all strings are cleared. The real-time and millisecond clocks are not effected. Generally, the NEW command is used to erase the RAM program and variables.

SEE ALSO : DELPGM

NULL

SYNTAX : NULL [#] *integer*
MODE: Command

The NULL command is used to output 0 to 255 NULL characters (ASCII 0) to the console device following the transmission of a CARRIAGE RETURN character from the ASCII BASIC Module during COMMAND mode. The addition of null characters can be used to provide additional time that might be required for a printer to mechanically perform the carriage return operation. The number of null characters sent by the module is initially zero.

The NULL command affects only the PRIMARY port, while the NULL# command is used to configure the AUXILIARY port.

The NULL output option only operates while in COMMAND mode. To obtain the same functionality during RUN mode, see the DELAY statement.

SEE ALSO : DELAY

RESET

SYNTAX : RESET
MODE: Command

The RESET command effectively causes the module to perform a software RESET, just as though a hardware reset or power-up had been performed. The RESET command has been provided as a means to test the RESET options (STARTUP, AUTORUN, etc.) without having to manipulate the hardware.

SEE ALSO : AUTORUN, STARTUP, BREAK, CLRMEM

RUN

SYNTAX : RUN
MODE: Command

After RUN is typed, all variables are set equal to zero, any pending ONTIME interrupts are cleared and program execution begins with the first line number of the selected program. The RUN command and the GOTO statement are the only way the user can place the ASCII BASIC Module into the RUN mode from the COMMAND mode. Program execution can be terminated at any time by typing a CONTROL-C character on the console device.

Some BASICs allow a line number to follow the RUN command. The ASCII BASIC Module does not permit such a variation on the RUN command, the RUN command causes execution to begin with the first line number. To obtain the same functionality as the RUN[line_num] syntax, use the GOTO[line_num] statement instead. Note that variables and BASIC interrupts are not cleared if the CLRMEM 0 option is in force, and CONTROL-C can be disabled using the BREAK 0 option.

SEE ALSO : GOTO, RUN operator

SAVE

SYNTAX : SAVE [*integer*]
MODE: Command

The SAVE command copies the currently selected program into the specified program number in the PROGRAM file.

The *integer* value must be between 1 and 254 inclusive. If no *integer* is specified or if storing the program using the specified number would leave a gap in program numbers, the program is copied into the next available program space in the PROGRAM file. PROGRAM NUMBERS IN PROGRAM FILE MEMORY REMAIN CONTIGUOUS STARTING WITH PROGRAM NUMBER 1.

After SAVE is entered, the ASCII BASIC Module responds with the program number that the stored program occupies in the PROGRAM file memory. This number is used when accessing the program with the AUTORUN, SELECT, CHAIN, EDIT and DELPGM commands.

If the program number specified already exists in the PROGRAM file, the existing program and all subsequent programs in the PROGRAM file are moved and the selected program is inserted as program number *integer*. For example, if there are 6 programs in the PROGRAM file (1 through 6), and the currently selected program was SAVEd as number 4, programs 4, 5 and 6 in the PROGRAM file are moved to 5, 6 and 7 respectively to make room for the new program 4.

SEE ALSO : DELPGM, AUTORUN, SELECT, EDIT, CHAIN

SELECT

SYNTAX : SELECT *integer*
MODE: Command

The SELECT command causes the ASCII BASIC Module to select the specified program as the default program. The *integer* specifies the program number assigned to the program when it was SAVED.

If an *integer* is specified for a program in the PROGRAM FILE that does not exist, a "Program does not exist" error is generated. If no *integer* is specified, the module defaults to program 0.

The SELECT command does not cause the specified program to be transferred into program 0. It is possible to have several different programs in the PROGRAM FILE memory as well as a separate program 0 in DATA memory.

When a program is SELECTed, it can be RUN or LISTed, but only program 0 can be edited. If an attempt is made to modify a program in the PROGRAM FILE memory, an error is generated.

Note that the COMMAND mode prompt contains a number that represents the currently SELECTed program.

SEE ALSO : SAVE, DELPGM, CHAIN, AUTORUN

STARTUP

SYNTAX : STARTUP *integer*
MODE: Command

The STARTUP command is used to configure the behavior of the module following a power-up or RESET condition. Valid *integer* arguments are described below:

STARTUP 0 : In this mode, the module enters its automatic baud rate detection sequence following RESET, waiting for a SPACE character (ASCII 32) to be transmitted to the PRIMARY serial port so that the baud rate can be established. Once the SPACE character has been received, the module displays the sign-on/status screen and enters COMMAND mode.

STARTUP 1 :In this mode, the module configures the PRIMARY serial port with the last baud rate used and immediately displays the sign-on/status screen following RESET and enters COMMAND mode.

STARTUP 2 : In this mode, the module configures the PRIMARY serial port with the last baud rate used and immediately runs the BASIC program specified by the last AUTORUN command. If no AUTORUN command has been issued, program 0 is assumed. If the specified program does not exist, the module reverts to STARTUP mode 1.

The STARTUP value is accessible as a configuration parameter via the PLC programming device. This feature is useful should the module be configured to run a BASIC program that implements the BREAK function without providing a means to terminate the program. The STARTUP mode can be set to mode 0 or mode 1 to prevent the program from running following the next RESET.

SEE ALSO : AUTORUN

STATUS

SYNTAX : STATUS
MODE: Command

The STATUS command causes the ASCII BASIC Module to display a screen of useful information regarding the current memory usage and some of the BASIC special function operators. A sample STATUS display is shown below:

```
0>STATUS

DATA MEMORY:
        32K bytes present, from 0 to 32767 (7FFFH).
        No program exists in DATA memory, 1537 bytes occupied.
        MTOP = 32767 (7FFFH).
        31231 bytes free.

PROGRAM FILE MEMORY:
        32K bytes present, from 32768 (8000H) to 65023 (FDFFH).
        10 program(s) exist in PROGRAM FILE memory, 21452 bytes occupied.
        10803 bytes free.

SYSTEM STATUS:
        AUTORUN:        Program number for automatic execution is 0.
        STARTUP:        Startup mode is set to 0.
        BREAK:  Control-C break checking is enabled.
        CLRMEM:         Data memory initialization is disabled.
        BAUD:           Stored primary port baud rate is 4800.
```

STEP

SYNTAX : STEP
MODE: Command

The STEP command causes the ASCII BASIC module to execute the next BASIC program line and then halt. It then returns to COMMAND mode. This single step operation provides a means of tracing program execution.

If the current program has not yet been RUN or has been modified since the last halt, the STEP command causes the first program line to be executed. Otherwise, the next line is executed (the line number displayed as a result of the BREAK@, STEP or STOP execution).

Note that if multiple statements appear on the line to be executed (separated by colons ":"), all of the statements on that line are executed. The STEP command follows program execution even if control is passed using a GOTO or GOSUB statement. Whenever a new line number is encountered, execution is halted, and the line number of the next line to be executed is displayed.

example

```
0>LIST
10        PRINT "This is line 10"
20        PRINT "This is line 20"
30        PRINT "This is line 30"
40        PRINT "This is line 40"
50        GOTO 10

Ready
0>STEP
This is line 10

LINE-STEP - In line 20
Ready
0>STEP
This is line 20

LINE-STEP - In line 30
Ready
0>
```

Note that whenever program execution is halted due to the STEP command, the LINE-STEP is displayed prior to the line number of the next line to be executed. When BREAK is displayed, the program was halted because of a BREAK@ breakpoint, and STOP is displayed when execution is halted due to a STOP statement or a control-C break.

SEE ALSO : BREAK@, CONT, STOP

4.2 BASIC Statements and Operators

All of the BASIC statements are described in this section and are grouped below according to the type of function performed. The BASIC statements are listed in alphabetical order on the following pages.

4.2.1 Program Control Statements

These statements are used to alter program flow or to transfer program execution to a specified point in the program (or to a different program).

CHAIN	**DO-UNTIL**	**GOSUB-RETURN**	**ONERR**		**ONTIME**
CLEAR I	**DO-WHILE**	**GOTO**	**ON-GOSUB**	**REM**	
CLEAR S	**END**	**IDLE**	**ON-GOTO**	**RETI**	
DELAY	**FOR-TO-STEP-NEXT**	**IF-THEN-ELSE**	**ONPORT**	**STOP**	

4.2.2 Data Manipulation Statements

These statements are used to alter or initialize the values of numeric variables.

CLEAR	**DIM**	**LD@**	**PUSH**	
DATA-READ-RESTORE	**LET**	**POP**	**ST@**	

4.2.3 Serial Port Control Statements

These statements are used to send and receive data to and from the ASCII BASIC module's PRIMARY and AUXILIARY serial ports.

CHR()	**INBUF$**	**PH0.**	**RTS**	
USING()				
CMDPORT	**INKEY$**	**PH1.**	**SETINPUT**	
CTS	**INPUT**	**PRINT**	**TAB()**	

4.2.4 Unary Operators

These operators perform predefined numeric functions.

ABS()	**BNR()**	**INP()**	**NOT()**	**SIN()**	**XBY()**
ATN()	**COS()**	**INT()**	**OUT()**	**SQR()**	
BCD()	**EXP()**	**LOG()**	**SGN()**	**TAN()**	

4.2.5 String Operators

These operators manipulate character "strings". See chapter 6 for a complete discussion of string manipulation.

ASC()	**INSTR()**	**LEN()**	**STRING**		**VAL()**
CHR$()	**LCASE$()**	**MID$()**	**STR$()**		
CR	**LEFT$()**	**RIGHT$()**	**UCASE$()**		

4.2.6 Time Handling Operators

These operators allow manipulation of the ASCII BASIC module's two timers, the REAL-TIME clock and the millisecond clock. See Chapter 7 for a complete discussion of the module's TIME handling capability.

CLOCK **DATE$** **FTIME** **TIME** **TIME$**

4.2.7 Special Function Operators

These operators provide specific information regarding program size, memory usage, error status or special numeric values.

ERC **FREE** **MTOP** **PI** **RND** **RUN** **SIZE**

4.2.8 Configuration Statements

These statements allow configuration of some of the ASCII BASIC module's characteristics.

BREAK **CLRMEM** **RTRAP** **SETCOM**

4.2.9 Logical Operators

These operators perform logical and bit-wise Boolean functions.

.AND. **.OR.** **.XOR.**

ABS()

SYNTAX : ABS(*expr*)
MODE: Run, Command

The ABS() operator returns the ABSOLUTE VALUE of the numeric *expr*.

```
example     0>PRINT ABS(5)          0>PRINT ABS(-5)
            5                       5
```

.AND.

SYNTAX 1 : *var* = *expr1* .AND. *expr2*
SYNTAX 2 : *rel_expr1* .AND. *rel_expr2*
MODE: Run, Command

For syntax 1, a bit-wise AND function is performed on the two expressions and the result is placed in the *var*. Each binary bit of the two expressions is manipulated as shown in the truth table below:

```
example     0>PRINT 2.AND.3         0>PH0. 55H.AND.0C0H
            2                        50H
```

For syntax 2, a logical AND function is performed on the two relational expressions. If BOTH relational expressions are TRUE, a TRUE result (65535) is returned. If either relational expression is FALSE, a FALSE result (0) is returned.

```
example     0>PRINT (2=2).AND.(3=3)     0>PRINT (2=3).AND.(3=2)
            65535                       0
```

SEE ALSO : .OR., .XOR., NOT()

ASC() operator

SYNTAX : ASC(*string_expr* [,*position*])
MODE: Run, Command

The ASC() returns the numeric ASCII value of the character at the specified *position* in the *string_expr*. If the *position* argument is omitted, the ASCII value of the first character in the *string_expr* is returned.

example
```
0>PRINT ASC("A")
65

0>STRING 257, 15
0>$(0)="Horner Electric"
0>PRINT CHR($(0),1), ASC($(0), 1)
H 72
```

In the following example, the ASCII value of each character in the string is displayed using the ASC operator.

example
```
0>10 STRING 110, 10
0>20 $(0) = "ABCDEFGHIJK"
0>30 FOR I=1 TO 11
0>40 PRINT ASC($(0), I),
0>50 NEXT I
0>60 END
0>RUN

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

SEE ALSO : CHR$(), STR$(), VAL()

ASC() function

SYNTAX : ASC(*string_var*, *position*) = *char*
MODE: Run, Command

The ASC() function replaces the character at the specified *position* in the *string_var* with the ASCII character represented by the numeric expression represented by *char*.

example
```
0>10 STRING 110, 10
0>20 $(0) = "abcdefghijk"
0>30 PRINT $(0)
0>40 ASC($(0),1)=75
0>50 PRINT $(0)
0>60 ASC($(0),2)=ASC($(0),3)
0>70 PRINT $(0)
0>RUN

abcdefghijk
Kbcedfghijk
Kccedfghijk
```

SEE ALSO : MID$(), ASC() operator

ATN()

SYNTAX : ATN(*expr*)
MODE: Run, Command

The ATN() Operator returns the trigonometric ARCTANGENT of the numeric *expr*. The argument is expressed in radians an must be between +/- 200000. The calculation is carried out to 7 significant digits.

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

SEE ALSO : COS(), SIN(), TAN(), PI

BCD()

SYNTAX : BCD(*binary_expr*)
MODE: Run, Command

The BCD() operator returns the BINARY CODED DECIMAL equivalent of the *binary_expr*. The *binary_expr* is a valid numeric expression. Note that many values are invalid and cannot be converted to BCD. For example, the values 10 through 15 would all return invalid BCD values. If an attempt is made to convert an invalid *binary_expr* to BCD, an Invalid argument error is generated.

example
```
0>10 BINVAL = 85 : REM Initialize
0>20 PRINT BCD(BINVAL)
0>30 BINVAL = BINVAL+1
0>40 GOTO 20
0>RUN

 55
 56
 57
 58
 59

ERROR! Bad argument! - In line 20

20  PRINT BCD(BINVAL)
————————————X
```

SEE ALSO : BNR()

BNR()

SYNTAX : BNR(*bcd_expr*)
MODE: Run, Command

The BNR() operator returns the BINARY equivalent of the *bcd_expr*. The *bcd_expr* is a valid numeric expression that solves to an integer value between 0 and 9999 inclusive. If an attempt is made to convert an invalid BCD value, an Invalid argument error is generated.

example
```
0>PRINT BNR(9999)
39321
```

SEE ALSO : BCD()

BREAK

SYNTAX : BREAK *num*
MODE: Run, Command

In normal operating conditions, the ASCII BASIC Module halts program execution when a CONTROL-C character (ASCII 3) is received at the PRIMARY serial port. This can cause problems under certain circumstances. If the PRIMARY serial port is used to communicate with an external device during program execution, the CONTROL-C character might be used in some sort of communication protocol. In this case, the programmer must insure that the CONTROL-C character does NOT cause the ASCII BASIC program to halt its execution. Additionally, the programmer can wish to disable the CONTROL-C break feature to prevent end users from halting a program that utilizes an operator interface terminal.

The BREAK command is used to disable and enable the ASCII BASIC module's CONTROL-C BREAK feature.

BREAK 0 Following execution of the BREAK 0 statement, when a CONTROL-C character is received by the ASCII BASIC module, program execution is NOT halted. If the CONTROL-C character is received while the module is in COMMAND mode, the CONTROL-C character is ECHOED to the transmitting device. The character is only echoed during an INPUT statement if the character echo feature is enabled (See the SETINPUT statement).

BREAK 1 Following execution of the BREAK 1 statement, when a CONTROL-C character is received by the ASCII BASIC module, program execution is halted. The module assumes this configuration following a RESET.

STOP! It is possible to configure a program to automatically run following RESET. If the program immediately disables the CONTROL-C break feature and does not provide a means for re-enabling THERE WILL BE **NO WAY FOR THE PROGRAMMER TO STOP THE PROGRAM**. For this reason, the programmer must provide a means for re-enabling the CONTROL-C interrupt from within the program.

There are several methods that can be incorporated to allow re-enabling of the CONTROL-C break feature. Two methods are illustrated below.

example 1 THREE SECOND TIMEOUT:

```
0>10 TIME=0 : CLOCK1                          : REM Initialize the clock
0>20 IF INKEY$="" THEN END                          : REM If a key is pressed, exit
0>30 IF TIME<=3 THEN 20                       : REM Wait for 3 seconds
0>40 BREAK0                                   : REM No key pressed, disable CTRL-C
                    (Rest of program)
```

example 2 PASSWORD:

```
0>10 STRING 257,15                            : REM Allocate string storage
0>20 BREAK0                                   : REM Disable CONTROL-C
0>30 $(0)="PASSWORD" : L=0                    : REM Define the password
0>40 GOSUB 100 : $(1)=$(1)+INKEY$             : REM Read the keyboard
0>50 IF LEN($(1)) < LEN($(0)) THEN 40               : REM If not enough chars, continue
0>60 IF $(0)=$(1) THEN 80                     : REM Otherwise, see if strings match
0>70 $(1)="" : GOTO 40                        : REM If not, set input string to null
0>80 BREAK 1 : GOTO 40                        : REM Otherwise, enable CTRL-C
0>100
                    (Rest of program)
```

CHAIN

SYNTAX : CHAIN *expr*
MODE: Run, Command

The CHAIN statement immediately transfers program control to the program specified by the numeric *expr*. The specified program is executed starting at it's first line number. If the specified program does not exist, the CHAIN statement is ignored and execution resumes with the statement following the CHAIN statement.

The *expr* is a valid expression that solves to an integer value between 0 and 254 inclusive, any other value causes an Invalid argument error. If the CLRMEM 0 option is in force, all string, array dimension and variable values are preserved between CHAINs. If the CLRMEM 0 option is NOT in force, all variables and strings are set equal to zero. CHAIN clears any pending ONERR, ONTIME or ONPORT interrupts.

SEE ALSO : CLRMEM

CHR()

SYNTAX 1 : CHR(*expr*)
SYNTAX 2 : CHR(*string_var, position*)
MODE: Run, Command

This operator is only included in the BASIC instruction set for compatibility with earlier versions of the firmware. New programs should use the more versatile CHR$() operator.

Using syntax 1, the CHR() operator converts the numeric *expr* to an ASCII character (See Appendix C for a list of the ASCII character set). The CHR() operator CAN ONLY BE USED WITHIN A PRINT STATEMENT!

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

Using syntax 2, the CHR() operator can also "pick out" characters from within an ASCII string. This is done by including the string variable name within the parentheses of the CHR operator, followed by the *position* of the character to "pick out".

example
```
0>$(0)="Horner Electric"
0>PRINT CHR($(0),1)
H
```

In the following example, the string is displayed using the CHR operator, one character at a time. Then the string is displayed in reverse.

example
```
0>10 STRING 257,15
0>20 $(0) = "ASCII BASIC"
0>30  FOR I=1 TO 11
0>40  PRINT CHR($(0), I),
0>50  NEXT I
0>60 PRINT
0>70  FOR I=11 TO 1 STEP -1
0>80  PRINT CHR($(0), I),
0>90  NEXT I
0>100 END
0>RUN
ASCII BASIC
CISAB IICSA
```

SEE ALSO : ASC(), CHR$(), MID$()

CHR$()

SYNTAX : CHR$(*expr*)
MODE: Run, Command

The CHR$() operator returns a single character string whose ASCII value is *expr* (See Appendix C for a list of the ASCII character set). The *expr* argument must solve to an integer value between 0 and 255 inclusive. The CHR$ operator can be used in any valid string expression.

example
```
0>PRINT CHR$(65)
A

0>PRINT "This is"+CHR$(20H)+CHR$(61H)+CHR$(20H)+"test"
This is a test
```

The CHR$() operator can also be used to imbed control characters inside string variables.

example
```
0>STRING 257,31
0>$(0)="Horner Electric"+CHR$(13)+CHR$(10)+"ASCII BASIC"
0>PRINT $(0)

Horner Electric
ASCII BASIC
```

SEE ALSO : ASC(), CHR(), MID$()

CLEAR

SYNTAX : CLEAR
MODE: Run, Command

The CLEAR statement sets all variables equal to zero, sets all strings to NULL, and executes the equivalent of the CLEAR I and CLEAR S statements. CLEAR does NOT reset the memory that has been allocated for strings via the STRING statement, nor does it affect the real-time or millisecond clocks. In general, the CLEAR statement is used to erase all variables.

example
```
0>10 X=1
0>20 CLEAR
0>30 PRINT X
0>RUN
 0
```

SEE ALSO : CLEAR I, CLEAR S, NEW, RUN, CLRMEM

CLEAR I

SYNTAX : CLEAR I
MODE: Run, Command

When the CLEAR I statement is executed, the ASCII BASIC Module clears and disables the ONTIME and ONPORT interrupts. This is used to disable the interrupts during specific sections of the user's BASIC program. This command does not affect the millisecond clock, which is enabled by the CLOCK1 statement, it merely inhibits the interrupts. If the CLEAR I statement is used, the ONTIME and/or ONPORT statements must be reissued before the interrupts are again recognized.

example
```
0>10 TIME = 0 : CLOCK1
0>20 ONTIME TIME + 1, 100
0>30 IF INT(TIME) <> 3 THEN 30
0>40 CLEAR I
0>50 IF TIME > 6 THEN 20 ELSE 50
0>60 REM ** Timer interrupt subroutine. **
0>100 PRINT TIME
0>110 ONTIME TIME + 1, 100
0>120 RETI
0>RUN

 1.005
 2.005
 3.005
 7.005
 8.005
```

SEE ALSO : CLEAR, CLEAR S, NEW, RUN, CLRMEM

CLEAR S

SYNTAX : CLEAR S
MODE: Run, Command

The CLEAR S statement is used to reset the two ASCII BASIC Module stacks (the CONTROL stack and the ARGUMENT stack, discussed in section 3.3). This statement can be used to "purge" the stack should an unrecoverable error occur. Also, this statement can be used as an alternate means to exit a FOR-NEXT, DO-WHILE or DO-UNTIL loop.

example
```
0>10 PRINT "Multiplication test, you have 5 seconds"
0>20 FOR I = 2 TO 9
0>30 N = INT(RND * 10) : A = N * I
0>40 PRINT "What is ", N, " * ", I, "?"
0>50 CLOCK1 : TIME = 0
0>60 ONTIME 5, 200 : INPUT R : IF R<> A THEN 100
0>70 PRINT "That's right!" : NEXT I
0>80 PRINT "You did it! Good job." : END
0>100 PRINT "Wrong, try again..." : GOTO 50
0>200 REM ** Reset the control stack, too much time. **
0>210 CLEAR S : PRINT "You took too long..." : GOTO 10
```

SEE ALSO : CLEAR, CLEAR I, NEW, RUN, CLRMEM

CLOCK

SYNTAX 1 : CLOCK 1
SYNTAX 2 : CLOCK 0
MODE: Run, Command

The CLOCK1 statement is used to START the millisecond clock. After execution of the CLOCK 1 statement, the special function operator TIME is incremented once every 5 milliseconds. When the millisecond clock is running, the ASCII BASIC program executes at about 99.8% of its normal speed.

The CLOCK 0 statement is used to STOP the millisecond clock. After execution of the CLOCK 0 statement, the special function operator TIME is no longer incremented. Following a power-up or reset, the millisecond clock is STOPPED.

SEE ALSO : TIME, FTIME, ONTIME, TIME$, DATE$

CLRMEM

SYNTAX 1 : CLRMEM 0
SYNTAX 2 : CLRMEM 1
MODE: Run, Command

If the CLRMEM 0 statement is executed, the ASCII BASIC Module does NOT clear any of the DATA memory following a power-up or RESET condition or prior to running a CHAINed program. This option allows the user to retain program 0 in DATA memory without the danger of losing the program following a RESET.

If the CLRMEM1 statement is executed, the ASCII BASIC Module clears DATA memory (up to MTOP) following a power-up or RESET condition. This means that if program 0 exists in DATA memory it is lost, and the value of any variables is initialized to zero following a power-up or RESET.

SEE ALSO : CLEAR, CLEAR I, CLEAR S, NEW, RUN, CHAIN

CMDPORT

SYNTAX: CMDPORT [#]
MODE: Run, Command

The CMDPORT statement is used to assign the programming console to the desired serial device. The "console" is the port used for entering commands, statements and program lines. Initially, (following RESET) the console is assigned to the PRIMARY serial device. Using the CMDPORT# statement, the console can be assigned to the AUXILIARY serial device. When the AUXILIARY serial port is configured as the console, all commands, statements and program lines are input via the AUXILIARY serial device. All command mode transmission from the module is directed to the AUXILIARY serial device.

Note that all of the serial port control commands and statements (e.g. PRINT, INPUT, etc.) must still incorporate the use of the "#" character in order to act upon the AUXILIARY serial device. The following example assumes that the module is in it's initial state:

```
example     0>CMDPORT#                  <The command port is now assigned to the AUXILIARY device>
```

After entry of the CMDPORT# command, the following sequence can be performed via the AUXILIARY port:

```
example     Ready
            0>10 REM This is a test line
            0>LIST

            Ready                       <The LIST command sent the output to the PRIMARY port>
            0>LIST#                     <The LIST# command will send output to the AUXILIARY port>
            10  REM This is a test line
```

When the CMDPORT statement is entered with no "#" parameter, the console device is assigned to the PRIMARY port. If the CMDPORT# statement is entered and no AUXILIARY serial device is present, the CMDPORT statement is ignored and the PRIMARY port retains the console.

This feature could be used with the modem option to allow ASCII BASIC programming from a remote location. To do this, the module would have to be programmed to establish the connection, at which point a password routine could be implemented to allow the remote station access to the COMMAND mode.

COMBRK

SYNTAX : COMBRK
MODE: Run, Command

The COMBRK statement is a special operator that returns or detects a three character time break.

When used with the PRINT statement, the COMBRK function can be used to send the three character break out one of the serial ports. This function is particularly useful when attempting to transfer SNP protocol messages from one serial port to another.

The COMBRK operator can also be used to imbed a three character break inside a string variable.

SEE ALSO : PRINT

COS()

SYNTAX : COS(*expr*)
MODE: Run, Command

The COS() operator returns the trigonometric COSINE of the numeric *expr*. The argument is expressed in radians and must be between +/- 200000. The calculation is carried out to seven significant digits.

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

SEE ALSO : ATN(), SIN(), TAN(), PI

CR

SYNTAX : CR
MODE: Run, Command

The CR statement is a special operator that returns a single CARRIAGE RETURN character (with no LINE FEED). When used in the PRINT statement, the CR function can be used to create a line on the console device that is repeatedly updated.

```
example     >10 FOR I=1 TO 1000
            >20 PRINT I, CR,
            >30 NEXT I
```

The CR operator can also be used to imbed a CARRIAGE RETURN character inside a string variable.

```
example     0> STRING 257,63
            0> $(0)="       is hidden"+CR+"This string"
            0> PRINT $(0)

            This string is hidden
```

SEE ALSO: CHR$()

CTS

SYNTAX : CTS [#]
MODE: Run, Command

The CTS operator is used to control the state of the hardware handshaking output on one of the two serial ports. The CTS (Clear To Send) signal is pin 8 on the DB-9 connector. This signal can be activated (to it's "high" state) by setting it to a nonzero value:

```
example     0>CTS=1                    : REM Sets the PRIMARY port CTS line HIGH (+12V)
            0>PRINT CTS
            65535
```

Likewise, the CTS signal can be deactivated (to it's "low" state) by setting it to zero.

```
example     0>CTS#=0                   : REM Sets the AUXILIARY port CTS line LOW (-12V)
            0>PRINT CTS#
            0
```

As shown in the example above, if a "#" character is appended to the CTS operator, the AUXILIARY port CTS line is manipulated.
SEE ALSO: RTS

DATA

SYNTAX : DATA *expr* [,*expr ,expr ...*]
MODE: Run

The DATA statement specifies CONSTANT expressions that can be retrieved by a READ statement. If multiple expressions are to be used for a single DATA statement, the expressions must be separated by commas. More than one DATA statement can appear within the same program, in this case the expressions in the DATA statements appear to BASIC as one long DATA statement. DATA statements can appear anywhere in the program: they are not executed and do not generate an error.

SEE ALSO: READ, RESTORE

DELAY

SYNTAX : DELAY(*expr*)
MODE: Run

The DELAY() function causes the module to pause for the number of milliseconds specified by the numeric *expr*.

The following example program segment updates the DATE and TIME on the console device approximately once per second.

```
example     0>10 PRINT DATE$, " ", TIME$, CR,     : REM Display the current DATE/TIME
            0>20 DELAY(1000)                       REM Wait for one second
            0>30 GOTO 10                           : REM Go update the DATE/TIME again
```

SEE ALSO: ONTIME, ONPORT

DIM

SYNTAX : DIM *var(expr)* [*var(expr), var(expr)...*]
MODE: Run, Command

The DIM statement reserves memory storage space for ARRAY variables. ARRAY variables can be assigned a ONE DIMENSION subscript which can not exceed 254. Once a variable has been DIMensioned in a program, it can not be re-DIMensioned. If this is attempted, an array size error is generated.

If an arrayed variable is used that has NOT been dimensioned using a DIM statement, the maximum subscript for the array is 9 (10 elements, 0 through 9). All arrays are set equal to zero whenever a NEW or CLEAR command are executed. If the CLRMEM 1 option is in force, all arrays are cleared following the RUN command.

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

```
example     0>10 A(5)=10
            0>20 DIM A(10)
            0>RUN

            ERROR! Array size exceeded or not specified! - In line 20

            20  DIM A(10)
            ————————X
```

SEE ALSO: CLEAR

DO-UNTIL

SYNTAX : DO ... UNTIL *rel_expr*
MODE: Run

The DO-UNTIL statements provide a means of "loop" control within an ASCII BASIC program. All statements between the DO and the UNTIL *rel_expr* are executed until the relational expression following the UNTIL becomes TRUE. DO - UNTIL loops can be nested.

example

```
0>10 A = 0
0>20  DO
0>30  A = A + 1
0>40  PRINT A
0>50  UNTIL A = 4
0>60 PRINT "DONE"
0>RUN

 1
 2
 3
 4
DONE
```

```
0>10 A = 0 : B = 0
0>20  DO
0>30  A = A + 1
0>40  DO
0>50  B = B + 1
          0>60  PRINT A, B, A*B
0>70  UNTIL B = 3
0>80  B = 0
0>90  UNTIL A = 3
0>100 PRINT "DONE"
0>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
DONE
```

SEE ALSO: DO-WHILE, FOR-TO-STEP-NEXT

DO-WHILE

SYNTAX : DO ... WHILE *rel_expr*
MODE: Run

The DO-WHILE statements provide a means of "loop" control within an ASCII BASIC program. All statements between the DO and the WHILE *rel_expr* are executed as long as the relational expression following the WHILE is TRUE. DO - WHILE loops can be nested.

example

```
0>10 A = 0
0>20  DO
0>30  A = A + 1
0>40  PRINT A
0>50  WHILE A <> 4
0>60 PRINT "DONE"
0>RUN

 1
 2
 3
 4
DONE
```

```
0>10 A = 0 : B = 0
0>20  DO
0>30  A = A + 1
0>40  DO
0>50  B = B + 1
          0>60  PRINT A, B, A*B
0>70  WHILE B <> 3
0>80  B = 0
0>90  WHILE A <> 2
0>100 PRINT "DONE"
0>RUN

 1 1 1
 1 2 2
 1 3 3
 2 1 2
 2 2 4
 2 3 6
DONE
```

SEE ALSO: DO-UNTIL, FOR-TO-STEP-NEXT

END

SYNTAX : END
MODE: Run

The END statement terminates program execution and puts the ASCII BASIC Module into the COMMAND mode. The CONT command can not be used to resume program execution if the END statement is used to terminate the program execution (a Can't continue error is generated). If no END statement is used, the last statement in the program automatically causes the program to END.

The two examples shown below can be considered identical.

example

```
0>10 FOR I = 1 TO 4
0>20 PRINT I
0>30 NEXT I
0>RUN

 1
 2
 3
 4
```

```
0>10 FOR I = 1 TO 4
0>20 PRINT I
0>30 NEXT I
0>40 END
0>RUN

 1
 2
 3
 4
```

SEE ALSO: STOP, CONT

ERC

SYNTAX : ERC
MODE: Run, Command

ERC is a READ-ONLY special function operator that only returns a meaningful result while in RUN mode. If used in COMMAND mode, zero is always returned.

In RUN mode, ERC returns the type of arithmetic error that last occurred. The ERC special function operator is typically used in an error trapping routine (see the ONERR statement). The value returned by ERC is one of 5 values:

No errors	(ERC = 0)
Division by zero	(ERC = 10)
Arithmetic overflow	(ERC = 20)
Arithmetic underflow	(ERC = 30)
Bad argument	(ERC = 40)

Note that when ERC is read, it is set to zero. This means that a variable should be used to store the error type if multiple error type tests are to be performed.

SEE ALSO: ONERR

EXP()

SYNTAX : EXP(*expr*)
MODE: Run, Command

The EXP() Operator returns the number "e" (2.7182818) raised to the power of the numeric *expr*.

example

```
0>PRINT EXP(1)
2.7182818
```

```
0>PRINT EXP(LOG(2))
2
```

SEE ALSO: LOG()

FOR - TO - STEP - NEXT

SYNTAX : FOR *var* = *start_expr* TO *end_expr* [STEP *inc_expr*] ... NEXT [*var*]
MODE: Run, Command

The FOR-TO-STEP-NEXT statements are used to execute "iterative" loops (or loops that are executed a specified number of times).

The ***var*** is a numeric variable that is incremented each time the NEXT statement is executed.
The ***start_expr*** is a numeric expression whose value is assigned to the *var* upon entry into the FOR statement. The ***end_expr*** is a numeric expression that the *var* is compared to each time the NEXT statement is executed. The ***inc_expr*** is a numeric expression whose value is added to the *var* each time the NEXT statement is executed.

Each time the NEXT statement is encountered, the *var* is compared to the ***end_expr***. If the ***var*** is less than the ***end_expr***, program control is transferred back to the statement following the FOR statement and the ***var*** is incremented by ***inc_expr***. If the *var* is greater than or equal to the ***end_expr***, control resumes with the statement following the NEXT statement.

If the STEP and ***inc_expr*** are omitted, the ***inc_expr*** defaults to 1.

```
example     0>10 FOR I = 1 TO 4
            0>20 PRINT I
            0>30 NEXT I
            0>RUN

             1
             2
             3
             4
```

The ***inc_expr*** can be a negative value, thus the ***var*** is actually decremented each time through the loop.

```
example     0>10 FOR I = 4 TO 1 STEP -1
            0>20 PRINT I
            0>30 NEXT I
            0>RUN

             4
             3
             2
             1
```

The ***var*** is very useful for accessing variable array elements (among other things). For example, consider an array containing the number of days in each month:

```
example     0>110 FOR X = 1 TO 12
            0>120 PRINT "There are ", MONTH(X), " days in month ", X
            0>130 NEXT X
```

The FOR-NEXT loop can be used in COMMAND mode, provided the entire sequence fits on a single command line.

```
example     Ready
            0>FOR X=0 TO 7 : PRINT INP(X) : NEXT X
```

SEE ALSO: DO-UNTIL, DO-WHILE, FOR-TO-STEP-NEXT

FREE

SYNTAX : FREE
MODE: Run, Command

The FREE system control value returns the number of unused bytes in DATA memory that are available to the user. When the currently selected program is in RAM, the following relationship holds true.

FREE = MTOP-SIZE-1280

The FREE operator DOES NOT account for any defined variables or string space below MTOP.

The FREE operator is generally used to derive a new value for MTOP.

example

```
0>NEW

0>10 FOR I=512 TO 528 : REM Display program 0
0>20 PRINT XBY(I),
0>30 NEXT I
0>PRINT FREE
 32202
```

SEE ALSO: MTOP, SIZE

FTIME

SYNTAX : FTIME
MODE: Run, Command

The FTIME special function operator is used to assign a value to the fractional portion of the TIME operator.

When the TIME operator is set using a LET statement, only the integer portion is affected. This is to allow accurate one-second intervals when an ONTIME interrupt is used.

Consider the following program segment:

example

```
0>10 TIME=0 : CLOCK 1 : ONTIME 1, 100
0>20 IDLE : GOTO 20

0>100 PRINT TIME$, CR,
0>110 TIME=0
0>120 ONTIME 1, 100
0>130 RETI
```

When the TIME operator was set equal to 0 in line 110, a few milliseconds had already elapsed since TIME was equal to 1 (generating the ONTIME interrupt). If when setting the TIME operator, the fractional portion were also changed, accurate second-based interrupts would be impossible.

For this reason, the FTIME operator is included. The value assigned to FTIME must be less than 1! If a value greater than or equal to 1 is assigned to FTIME, an Invalid argument error is generated. Note that the value assigned to FTIME is truncated to the nearest 5 milliseconds.

SEE ALSO: TIME, CLOCK, ONTIME, TIME$, DATE$

GOSUB - RETURN

SYNTAX : GOSUB *line_num* ... RETURN
MODE: Run

The GOSUB statement transfers program control to the specified program *line_num*. The location of the GOSUB statement is saved on the CONTROL STACK. If the specified *line_num* does not exist, an Invalid line number error is generated.

When a RETURN statement is encountered, the location of the GOSUB statement is retrieved from the CONTROL STACK and program control is transferred to the statement following the GOSUB statement.

These two statements provide a means to incorporate SUBROUTINES. A subroutine is a program segment that can be executed from several points in the program. Instead of keeping several copies of an identical program segment in the program, a subroutine can be created to reduce program size and to ease the program maintenance and debug chores.

Subroutines can be nested. This means that a GOSUB can be used from within a subroutine.

example

```
0>10 FOR I = 1 TO 3
0>20 GOSUB 100
0>30 NEXT I
0>40 END

0>100 PRINT I,
0>110 GOSUB 200
0>120 RETURN

0>200 PRINT I*I
0>210 RETURN
0>RUN

 1 1
 2 4
 3 9
```

SEE ALSO: GOTO, ON-GOSUB, ON-GOTO

GOTO

SYNTAX : GOTO *line_num*
MODE: Run, Command

The GOTO statement causes BASIC to immediately transfer program control to the specified program *line_num*. If the specified *line_num* does not exist, an Invalid line number error is generated.

The GOTO statement can be used while in COMMAND mode. Unlike the RUN command, this action does NOT cause BASIC to clear the variable storage space or BASIC interrupts (unless the program has been modified).

SEE ALSO: GOSUB, ON-GOSUB, ON-GOTO

IDLE

SYNTAX : IDLE
MODE: Run, Command

The IDLE statement forces the ASCII BASIC Module to cease all program execution activity until an ONTIME or ONPORT interrupt is generated. The programmer must insure that one of these interrupts is pending before executing an IDLE statement, otherwise the module is IDLE forever.

Once the ONTIME or ONPORT interrupt is generated, the module breaks out of the IDLE mode and executes the interrupt service subroutine. When the RETI statement is encountered, program control returns to the statement following the IDLE statement.

example
```
0>10 TIME=0 : CLOCK1 : ONTIME 1, 100
0>20 IDLE : GOTO 20

0>100 PRINT TIME$, CR,
0>110 TIME=0 : ONTIME 1,100
0>120 RETI
```

SEE ALSO: ONPORT, ONTIME, RETI, CLEAR I

IF - THEN - ELSE

SYNTAX : IF *rel_expr* [THEN] *true_statement* [ELSE *false_statement*]
MODE: Run

The IF-THEN-ELSE statement provides a means to perform a conditional test. The *rel_expr* is used to determine which of the statements following the THEN to execute. If the *rel_expr* evaluates to 65535 (or is TRUE), the *true_statement* following the THEN is executed. Otherwise, if the *rel_expr* evaluates to zero (or is FALSE) the *false_statement* following the ELSE is executed.

The ELSE portion of this statement is optional and can be omitted. If it is omitted and the *rel_expr* is FALSE, the statement following the THEN is not executed and program control resumes with the statement following the IF statement.

In the following statement, IF A is equal to 100 THEN A would be assigned a value of 0, otherwise, A is incremented by 1.

example
```
0>10 IF A=100 THEN A=0 ELSE A=A+1
```

If program control is to be transferred to a different line number using the GOTO statement, the GOTO keyword can be omitted. The following examples are functionally equivalent.

example
```
0>10 IF A>100 THEN GOTO 50 ELSE GOTO 100
0>10 IF A>100 THEN 50 ELSE 100
```

Additionally, the THEN keyword can be replaced by any valid ASCII BASIC statement. Again, the following two statements are functionally equivalent.

example
```
0>10 IF A>100 THEN PRINT A
0>10 IF A>100 PRINT A
```

Multiple statements can be placed on same line following an IF-THEN-ELSE statement (using the colon ":" delimiter). The additional statements are executed if the LAST target statement of the IF-THEN-ELSE statement is executed.

example
```
0>10 X=0
0>20 IF X= 0 THEN PRINT "X is", : PRINT " equal to zero"
0>RUN

X is equal to zero

Ready
0>10 X = 1
0>20 IF X=0 THEN PRINT "zero" ELSE PRINT "Greater " : PRINT "than zero"
0>RUN

Greater than zero
```

Note: Multiple statements can not appear between the THEN and the ELSE.

INBUF$

SYNTAX : INBUF$ [#]
MODE: Run, Command

The INBUF$ operator is a READ-ONLY special function operator that returns a character string that represents all of the characters currently in the INPUT buffer for the specified serial port. The characters are NOT removed from the buffer. If the "#" character is appended to the INBUF$ operator, the AUXILIARY serial INPUT buffer is returned. Otherwise, the PRIMARY serial buffer is returned.

The INBUF$ operator is useful when dealing with a serial communications protocol. The string operators can be used to determine the number of characters waiting in the buffer or if a particular terminating character has been received.

example
```
0>100 IF LEN(INBUF$)>10 THEN 200                : REM Wait for 10 characters
0>110 IF INSTR(INBUF$,CR)>0 THEN 200            : REM Wait for a <CR>
0>120 GOTO 100

0>200 INPUT $(0)                                : REM Read the buffer
```

SEE ALSO: INKEY$, INPUT, SETINPUT, LEN()

INKEY$

SYNTAX : INKEY$ [#]
MODE: Run

The INKEY$ special function operator produces a meaningful result when used in the RUN mode. It returns a null string in COMMAND mode.

The INKEY$ operator returns a one character string that represents the next available character waiting in the serial INPUT buffer. If no characters are waiting in the INPUT buffer, INKEY$ returns a null string.

If one or more characters are available in the INPUT buffer, the INKEY$ operator REMOVEs the character from the buffer and returns it in the string.

If the "#" character is appended to the INKEY$ operator, the AUXILIARY serial INPUT buffer is read, otherwise, the PRIMARY serial INPUT buffer is read.

The following program simply reads all characters from the PRIMARY port and transmits them out the AUXILIARY port. Likewise, all characters received at the AUXILIARY port are transmitted to the PRIMARY port.

```
example     0>10 PRINT INKEY$ #,
            0>20 PRINT # INKEY$,
            0>30 GOTO 10
```

SEE ALSO: INBUF$, INPUT, SETINPUT

INP()

SYNTAX 1 : INP(*reg_num*)
SYNTAX 2 : INP(*reg_num, bit_num*)
MODE: Run, Command

The INP() operator appears as an 8 element READ-ONLY array of variables that represents the sixty-four 16-bit PLC output registers (AQ) for the ASCII BASIC Module. The *reg_num* is a numeric expression that yields an integer value between 0 and 7 inclusive. Any other value generates an Invalid argument.

```
example     0>10 X=INP(0) : REM Lower byte represents X coordinate
            0>20 Y=INP(1) : REM Upper byte represents Y coordinate
            0>30 PRINT "The X coordinate is",BCD(X),
            0>40 PRINT "and the Y coordinate is",BCD(Y)
            0>50 END
            0>RUN

            The X coordinate is 17 and the Y coordinate is 85
```

The INP() operator can also be used to obtain the state of a single BIT in any of the sixty-four input registers. The *reg_num* is a numeric expression that represents the register number (0 to 7) that contains the bit number (0 to 15) represented by *bit_num*. An Invalid argument error is generated if either expression is out of range.

If the specified bit is set (1), a TRUE result (65535) is returned. If the specified bit is clear (0), a FALSE result (0) is returned.

```
example     0>10 FOR X=0 TO 7                              : REM Do all 8 registers
            0>20 FOR Y=0 TO 15                             : REM Do all 15 bits
            0>30 IF INP(X,Y) PRINT "1", ELSE PRINT "0"     : REM Display state of bit
            0>40 NEXT Y                                    : REM Do next bit
            0>50 PRINT                                     : REM New line when end of reg
            0>60 NEXT X                                    : REM Do next reg
```

SEE ALSO : OUT()

INPUT

SYNTAX : INPUT [#] [*"prompt_string"*] [,] *var* [,*var ,var ...*]
MODE: Run

The INPUT statement reads data from one of the serial INPUT buffers and assign the data to the *var*(s) in the variable list.

If the "#" character is appended to the INPUT keyword, the AUXILIARY serial INPUT buffer is read Otherwise, the PRIMARY serial INPUT buffer is read.

One or more variables can be assigned data with a single INPUT statement. If more than one variable appears in the INPUT variable list, they must be separated by commas. If the user does not enter enough data, a warning message is displayed on the console device and the INPUT statement is re-executed. When more than one data item is to be entered in response to an INPUT statement, each data item must be separated by a comma.

Normally, a carriage return/line feed sequence and a question mark (?) are transmitted to the specified device. However, if a comma immediately precedes the first *var* in the variable list, the carriage return/line feed/question mark sequence is suppressed.

example
```
0>10 INPUT A,B
0>20 PRINT A,B
0>RUN

?1    <typed by user>

Data entered does not match variable type...try again

?1,2   <typed by user>
 1 5
```

An INPUT statement can be written so that a descriptive *"prompt_string"* is displayed on the console device prior to data entry. This prompt must appear between the INPUT keyword and the variable list, and must be enclosed in quotes (").

example
```
0>10 INPUT "Enter a number - ", A
0>20 PRINT SQR(A)
0>RUN

Enter a number - 100
 10
```

String variables can also be assigned using the INPUT statement. During INPUT, strings are normally terminated with a carriage return character (but any character can be configured as the terminating character using the SETINPUT statement). If more than one string is to be assigned with a single INPUT statement, the termination character must be sent following each string. The module prompts the user with a question mark between each string entry. If a comma is entered during the INPUT of a string variable, it is simply placed in the string.

example
```
0>10 STRING 110,10
0>20 INPUT "NAME: ",$(1)
0>30 PRINT "HI ",$(1)
0>RUN

NAME: Jim
HI Jim
```

```
0>10 STRING 110,10
0>20 INPUT "NAMES: ",$(1),$(2)
0>30 PRINT "HI ",$(1)," AND ",$(2)
0>RUN

NAMES: Jim
?Joe
HI Jim AND Joe
```

SEE ALSO: SETINPUT, INBUF$, INKEY$

INSTR()

SYNTAX : INSTR (*string_expr1, string_expr2*)
MODE: Run, Command

The INSTR() function searches for the first occurrence of *string_expr2* in *string_expr1* and returns the character position at which the match is found.

If *string_expr2* is not found in *string_expr1*, 0 is returned.

example
```
0>PRINT INSTR("This is a test", "is a")
 6

0>STRING 257,31
0>$(0)="Horner Electric"
0>PRINT INSTR($(0), CHR$(45H))
8

0>PRINT INSTR($(0), "F")
0
```

SEE ALSO: MID$(), LEN()

INT()

SYNTAX : INT(*expr*)
MODE: Run, Command

The INT() operator returns the INTEGER PORTION of the numeric *expr*.

example
```
0>PRINT INT(3.7)
3
```
```
0>PRINT INT(104.554)
104
```
```
0>PRINT INT(PI)
3
```

LCASE$()

SYNTAX : LCASE$(*string_expr*)
MODE: Run, Command

The LCASE$ function returns the *string_expr* with all of the alphabetic characters converted to lower case.

example
```
0>PRINT LCASE$("THIS is A tEsT")
this is a test

0>STRING 257,31
0>$(0)="HorNEr ELEctRIc"
0>PRINT LCASE$($(0))
horner electric
```

SEE ALSO: UCASE$()

LD@

SYNTAX : LD@ *expr*
MODE: Run, Command

The LD@ expr statement, in conjunction with the ST@ statement allow the programmer to store and retrieve floating-point values anywhere in the module's DATA memory.

The *expr* is a numeric expression that represents the DATA memory address of the value (placed using the ST@ statement). The value is copied starting at the specified address and is placed on the ARGUMENT STACK. The value can then be POPped off of the ARGUMENT STACK into a user variable.

Each floating-point value requires 6 bytes of storage. The expression specified in the statements represent the HIGHEST memory address used by the operation. For example, ST@ 32767 would actually cause the value to be stored at addresses 32767, 32766, 32765, 32764, 32763 and 32762.

The LD@ and ST@ statements are the only available means of passing floating-point values between CHAINed programs when the CLRMEM 1 option is in force (See CHAIN statement).

STOP! The LD@ and ST@ statements allow manipulation of DATA memory and does not check the specified addresses. It is possible to specify addresses in DATA memory that interfere with ASCII BASIC Module firmware operations or with the RAM program or string variable operations. To avoid problems, the programmer should use the SPECIAL FUNCTION OPERATOR, MTOP to set aside a protected area of memory for use by these instructions.

example

```
0>10 REM ** Save array **
0>20 FOR I=0 TO 9
0>30 PUSH A(I)                  : REM Put array on arg stack
0>40 ST@ 32767+(6*I)            : REM Store it, 6 bytes per value
0>50 NEXT I
0>60 REM ** Get array **
0>70 FOR I=0 TO 9
0>80 LD@ 32767+(6*I)
0>90 POP B(I)
0>100 NEXT I
```

SEE ALSO: ST , MTOP, FREE, SIZE, CHAIN

LEFT$()

SYNTAX : LEFT$(*string_expr, num*)
MODE: Run, Command

The LEFT$ function returns the leftmost *num* characters of the *string_expr*.

example
```
0>PRINT LEFT$("Horner Electric", 8)
Horner E

0>STRING 257,15

0>$(0)="This is a test"

0>PRINT LEFT$($(0), 6)
This i
```

SEE ALSO: RIGHT$(), MID$()

LEN()

SYNTAX : LEN(*string_expr*)
MODE: Run, Command

The LEN() function returns the number of characters contained in the *string_expr*.

example
```
0>PRINT LEN("Horner Electric")
15

0>STRING 257,15
0>$(0)="This is a test"
0>PRINT LEN($(0))
14
```

LET

SYNTAX : [LET] *var = expr*
MODE: Run, Command

The LET statement is used to assign a variable to the value of an expression.

Note that the equal sign (=) is not used to test equality, instead it causes the value of the *var* to be replaced by the value of the *expr*. Also note that the keyword LET is always OPTIONAL and can be omitted. When the LET keyword is omitted, the LET statement is called an IMPLIED LET statement.

The LET statement is also used to assign values to string variables and special function operators.
The following examples are ALL valid LET statements.

example
```
0>LET A=5
0>D(0)=15
0>$(0)="Horner Electric"

0>10 A=5
0>20 TIME=0
0>30 A=5 : B=PI/2 : C=COS(B)
```

LOG()

SYNTAX : LOG(*expr*)
MODE: Run, Command

The LOG() Operator returns the natural logarithm of the *expr*. The *expr* is a numeric expression and must solve to a positive value. The returned value contains up to 7 digits of significance.

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

SEE ALSO: EXP()

MID$()

SYNTAX : MID$(*string_expr, start_position* [,*num*])
MODE: Run, Command

The MID$() function returns *num* characters of the *string_expr* beginning with the character at *start_position*. If the *num* parameter is omitted, the remainder of the *string_expr* is returned.

The *start_position* and *num* arguments must be valid integer expressions between 1 and 255 inclusive. If *start_position* specifies a character position greater than the number of characters in the *string_expr*, a null string is returned. If *num* specifies more characters than are available in the *string_expr* or if *num* is omitted, the remainder of the *string_expr* is returned.

example
```
0>PRINT MID$("This is a test", 6, 4)
is a
```

```
0>10 STRING 257,15
0>20 $(0)=" Test program "
0>30 FOR X=0 TO 14
0>40 PRINT MID$($(0),X,3), CR,
0>50 DELAY(500)
0>60 NEXT X
0>70 GOTO 30
```

SEE ALSO: LEFT$(), RIGHT$(), ASC()

MTOP

SYNTAX : MTOP
MODE: Run, Command

Following RESET, the ASCII BASIC Module reads the last known value of MTOP from battery backed memory. Whenever the MTOP value is changed, it is stored in battery-backed memory. Initially, this value is set to the last available address of DATA memory. MTOP is used by basic to determine the location of variables and string storage space

The user can assign a different value to MTOP allowing a region of protected DATA memory for use with the XBY(), LD@ and ST@ statements. If MTOP is assigned a value beyond available DATA memory, a MEMORY ALLOCATION error is generated.

If a program modifies the MTOP value, it should be done in the FIRST statement of the program, as BASIC stores any referenced variable or string starting from MTOP down.

The amount of unused DATA memory can be determined using the LEN and FREE commands, described in this chapter.

```
example     0>PRINT MTOP
            32767

            0>MTOP=32700          : REM BASIC will not use memory from here to 32767

            0>PRINT MTOP
            32700
```

SEE ALSO: DIM, FREE, SIZE, STRING

NOT()

SYNTAX : NOT(*expr*)
MODE: Run, Command

The NOT() operator returns the 16-bit ONE'S COMPLEMENT of the *expr*. The *expr* is a numeric expression that must solve to a valid integer (0 to 65535 inclusive). Non-integers are truncated - NOT rounded.

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

SEE ALSO: .AND., .OR., .XOR.

ON - GOSUB

SYNTAX : ON *expr* GOSUB *line_num* [*, line_num, line_num ...*]
MODE: Run

The ON - GOSUB statement evaluates the numeric *expr* and transfer program control to one of the specified line numbers in the *line_num* list.

The *expr* is a numeric expression that must evaluate to an integer value from zero to the number of line numbers in the line number list. If the *expr* is less than zero or greater than the number of line numbers in the list, an Invalid argument error is generated.

After the successful execution of the ON-GOSUB statement, when a RETURN statement is encountered, program control resumes at the statement following the ON-GOSUB statement.

In the following example, if X is equal to 0, program control transfers to the subroutine at line 100. If X is equal to 1, the subroutine at line 200 is executed. If X is 2, we GOSUB line 300, and if X is 3, line 400 is executed.

example 0>10 ON X GOSUB 100, 200, 300, 400

SEE ALSO: GOSUB, GOTO, ON-GOTO

ON - GOTO

SYNTAX : ON *expr* GOTO *line_num* [*, line_num, line_num ...*]
MODE: Run, Command

The ON - GOTO statement evaluates the *expr* and transfer program control to one of the specified line numbers in the *line_num* list.

The *expr* is a numeric expression that must evaluate to an integer value from zero to the number of line numbers in the line number list. If the *expr* is less than zero or greater than the number of line numbers in the list, an Invalid argument error is generated.

In the following example, if X is equal to 0, program control transfers to line 100. If X is equal to 1, line 200 is executed. If X is 2, we GOTO line 300, and if X is 3, line 400 is executed.

example 0>10 ON X GOTO 100, 200, 300, 400

SEE ALSO: GOSUB, GOTO, ON-GOSUB

ONERR

SYNTAX : ONERR *line_num*
MODE: Run

The ONERR statement provides a means for the programmer to "handle" arithmetic errors that can occur during program execution. There are four types of arithmetic errors that cause program control to transfer to the specified *line_num*:

Division by zero	(ERC = 10)
Arithmetic overflow	(ERC = 20)
Arithmetic underflow	(ERC = 30)
Bad argument	(ERC = 40)

The error code value can be examined using the ERC special function operator.

Following the execution of the ONERR statement, if any of the above arithmetic errors are encountered, program control passes to the specified *line_num*. The *line_num* specified is the beginning of an error handling routine that processes the error in a manner appropriate to the application.

Note that the ONERR routine should not perform a RETURN or a RETI instruction. There is no way to "RESUME" program operation from where the error occurred. The error handling routine must GOTO the appropriate line number in the application program.

Typically, this statement is used to handle errors that can occur when the user has entered inappropriate data to an INPUT statement.

example

```
0>10 ONERR 100
0>20 FOR I=3 TO 0 STEP -1
0>30 PRINT 100/I
0>40 NEXT I
0>100 ETYPE = (ERC/10)-1
0>110 ON ETYPE GOTO 120, 130, 140, 150
0>120 PRINT "Division by zero" : END
0>130 PRINT "Underflow" : END
0>140 PRINT "Overflow" : END
0>150 PRINT "Bad argument" : END
0>110 END
```

SEE ALSO: ERC

ONPORT

SYNTAX : ONPORT [#] *line_num*
MODE: Run

The ONPORT statement provides a communications-based interrupt function. Following the execution of the ONPORT statement, the next character received at the specified serial port causes a BASIC interrupt to be generated, and program control is passed to the specified *line_num.*

If the "#" character is appended to the ONPORT keyword, the AUXILIARY serial port interrupt is armed, otherwise the PRIMARY serial port interrupt is armed.

Once an ONPORT interrupt is generated, the ONPORT interrupt is "disarmed". If subsequent serial interrupts are to be generated, the ONPORT statement should be issued from WITHIN the serial interrupt service subroutine.

The RETI statement must be used in place of the RETURN statement in the serial interrupt subroutine! Failure to do this causes the ASCII BASIC Module to ignore all future timer ONPORT interrupts.

example
```
0>10 ONPORT 100
0>20 ONPORT# 200
0>30 IDLE : GOTO 30
0>100 PRINT "Primary serial interrupt, character received was - ", INKEY$
0>110 ONPORT 100 : RETI
0>200 PRINT "Auxiliary serial interrupt, character received was - ", INKEY$#
0>210 ONPORT# 200 : RETI
```

SEE ALSO: IDLE, RETI, ONTIME

ONTIME

SYNTAX : ONTIME *expr, line_num*
MODE: Run

The ONTIME statement provides a time-based interrupt function. The ONTIME statement uses the special function operator, TIME. Whenever the TIME operator is greater than or equal to the specified *expr*, a timer interrupt is generated and program control is passed to the specified *line_num*.

Only the integer portion of the expression is compared to the integer portion of the TIME operator.

Once an ONTIME interrupt is generated, the ONTIME interrupt is "disarmed". If recursive timer interrupts are to be generated on a periodic basis, the ONTIME statement should be issued from WITHIN the timer interrupt service subroutine.

The RETI statement must be used in place of the RETURN statement in the timer interrupt subroutine! Failure to do this causes the ASCII BASIC Module to ignore all future interrupts.

example
```
0>10 TIME=0 : CLOCK1 : ONTIME 2, 100
0>20 DO
0>30 UNTIL TIME>10
0>40 END
0>100 PRINT “Timer interrupt at - “, TIME, “seconds”
0>110 ONTIME TIME+2, 100
0>120 RETI
0>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
```

Note that in the previous example the TIME displayed is 45 milliseconds greater than the time that the interrupt was supposed to be generated. This is due to the amount of time required to transmit the PRINTed string prior to the TIME display (at 4800 baud). To avoid this delay, assign a variable to TIME at the beginning of the interrupt routine.

example
```
0>10 TIME=0 : CLOCK1 : ONTIME 2, 100
0>20 DO
0>30 UNTIL TIME>10
0>40 END
0>100 A=TIME
0>110 PRINT “Timer interrupt at - “, A, “seconds”
0>120 ONTIME TIME+2, 100
0>130 RETI
0>RUN

Timer interrupt at - 2 seconds
Timer interrupt at - 4 seconds
Timer interrupt at - 6 seconds
Timer interrupt at - 8 seconds
Timer interrupt at - 10 seconds
```

SEE ALSO: TIME, FTIME, IDLE, RETI, ONPORT

.OR.

SYNTAX 1 : *var* = *expr1* .OR. *expr2*
SYNTAX 2 : *rel_expr1* .AND. *rel_expr2*
MODE: Run, Command

Using syntax 1, a bit-wise OR function is performed on the two expressions and the result is placed in the *var*. Each binary bit of the two expressions is manipulated as shown in the truth table below:

EXPR1	EXPR2	RESULT
0	0	0
0	1	1
1	0	1
1	1	1

example
```
0>PRINT 2.OR.3
3
```
```
0>PH0. 55H.OR.0C0H
D5H
```

Using syntax 2, a logical OR function is performed on the two relational expressions. If EITHER of the relational expressions are TRUE, a TRUE result (65535) is returned. If both relational expressions are FALSE, a FALSE result (0) is returned.

example
```
0>PRINT (2=2).OR.(3=2)
65535
```
```
0>PRINT (2=3).OR.(3=2)
0
```

SEE ALSO: .AND., .XOR., NOT()

OUT()

SYNTAX 1 : OUT(*reg_num*)
SYNTAX 2 : OUT(*reg_num, bit_num*) = *expr*
MODE: Run, Command

The OUT() operator appears as an 8 element WRITE-ONLY array of variables that represents the sixty-four 16-bit PLC input registers (AI) for the ASCII BASIC Module. The *reg_num* is a numeric expression that must yield an integer value between 0 and 7 inclusive. Any other value generates an Invalid argument error.

example
```
0>10 OUT(0)=BCD(INP(0))
0>20 OUT(1)=BNR(INP(1))
```

Using syntax 2, the OUT() operator can also be used to assign the state of a single BIT in any of the sixty-four output registers. The *reg_num* is a numeric expression which represents the register number (0 to 7) that contains the bit number (0 to 15) represented by *bit_num.* An Invalid argument error is generated if either expression is out of range.

If the *expr* is nonzero, the specified bit is set (1), otherwise the specified bit is cleared (0).

example
```
0>OUT(0,7)=1 : REM Set bit 7 of register 0.
0>OUT(X,Y)=0 : REM Clear bit Y of register X.
```

SEE ALSO: INP()

PH0.

SYNTAX : PH0. [#] *expr_list*
MODE: Run, Command

The PH0. statement is functionally identical to the PRINT statement, except that all numeric values are PRINTed in HEXADECIMAL (base 16) format.

The PH0. statement displays numeric values in at least two digits followed by an “H” character. If the value displayed is less than 10H, a leading zero is displayed. If the value displayed is greater than 255, three or four digits is displayed.

If a numeric value to be displayed is greater than 65535, the module prints the value in decimal format.

Numeric values are truncated to integers before being printed.

All format specifiers that are used with the PRINT statement can be used with PH0. and PH1. statements.

example
```
0>PH0. 2*2
04H

0>PH0. 1000
3E8H

0>PH0. PI
03H

0>PH0. 600000
600000
```

SEE ALSO: PRINT, PH1., TAB()

PH1.

SYNTAX : PH1. [#] expr_list
MODE: Run, Command

The PH1. statement is functionally identical to the PH0. statement except that four hexadecimal digits are PRINTed. Leading zeros are displayed where necessary.

example
```
0>PH1. 2*2
0004H

0>PH1. 1000
03E8H

0>PH1. PI
0003H

0>PH1. 600000
600000
```

SEE ALSO: PRINT, PH0., TAB()

PI

SYNTAX : PI
MODE: Run, Command

PI is a stored constant. The value returned by the PI constant is 3.1415926. Mathematicians notice that the value of PI is actually closer to 3.141592653 so proper rounding for PI yields a value of 3.1415927. The reason that the ASCII BASIC Module uses a "6" instead of a "7" for the least significant digit is that errors in the SIN, COS and TAN operators were found to be greater when 7 was used instead of 6. This is because the number PI/2 is needed for these calculations, and it is desirable to have the equation (PI/2+PI/2=PI) hold true. This cannot be done if the last digit of PI is an odd number so the last digit of PI was rounded to 6 instead of 7 to make these calculations more accurate.

example
```
0>PRINT PI
3.1415926
```

SEE ALSO: ATN(), COS(), SIN(), TAN()

POP

SYNTAX : POP *expr* [*,expr, expr ...*]
MODE: Run, Command

The POP statement, when used with the PUSH statement, provides a means of passing parameters to BASIC subroutines via the BASIC ARGUMENT STACK.

Note that more than one value can be POPped with one POP statement. The last value PUSHED onto the ARGUMENT STACK is always the first value POPped off of the ARGUMENT STACK.

The following example shows how the PUSH and POP statements can be used to "swap" two variables.

example
```
0>10 A=5 : B=10
0>20 PRINT A,B
0>30 PUSH A,B
0>40 POP A,B
0>50 PRINT A,B
0>RUN

5 10
10 5
```

SEE ALSO: PUSH

PRINT

SYNTAX : PRINT [#] expr_list
MODE: Run, Command

The PRINT statement directs the ASCII BASIC Module to output data to the specified serial device. The value of expressions, strings, literal values, variables or test strings can be manipulated by the PRINT statement. The various forms can be combined in the *expr_list* by separating them with commas.

If the "#" character is appended to the PRINT keyword, data is sent to the AUXILIARY serial device. Otherwise, the data is sent to the PRIMARY serial device.

Normally, a carriage return/line feed sequence is PRINTed following the last item in the *expr_list*. If the list is terminated with a comma, the carriage return/line feed is suppressed.

When numeric values are PRINTED, a leading SPACE is included if the value is positive. Otherwise, the minus sign (-) precedes the value. Also, if the numeric value is to be displayed in decimal notation, a trailing SPACE is appended to the displayed value, otherwise, the hexadecimal specifier “H” is displayed.

The PRINT keyword can be abbreviated as "P." or by a question mark (?).

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

0>PRINT "Hello, world!"
Hello, world!

0>P. 10-20, 1E3
-10
1000

0>? "The value of PI is ", PI
The value of PI is 3.1415926
```

SEE ALSO: SPC(), TAB(), USING(), PH0., PH1., CHR()

PUSH

SYNTAX : PUSH *expr* [,*expr, expr ...*]
MODE: Run, Command

The PUSH statement provides a means of passing parameters to BASIC subroutines via the BASIC ARGUMENT STACK. Note that more than one value can be PUSHed with one PUSH statement. The last value PUSHed onto the ARGUMENT STACK is always the first value POPped off of the ARGUMENT STACK.

The following example shows how the PUSH and POP statements can be used to swap two variables.

example
```
0>10 A=5 : B=10
0>20 PRINT A,B
0>30 PUSH A,B
0>40 POP A,B
0>50 PRINT A,B
0>RUN

 5 10
 10 5
```

SEE ALSO: POP

READ

SYNTAX : READ *var* [*,var ,var ...*]
MODE: Run

The READ statement is used to sequentially retrieve the expressions that appear in the DATA statement(s). It must be followed by one or more variable names. Each *var* following the READ statement is assigned the value of the next "unREAD" expression in the DATA list. If more than one *var* appears following the READ statement, they must be separated by commas. If all expressions in a program's DATA statements have been READ and another READ is attempted without RESTOREing, an Out of data error is generated.

```
example     0>10 FOR I=1 TO 3
            0>20 READ A,B
            0>30 PRINT A,B
            0>40 NEXT I
            0>50 RESTORE
            0>60 READ A,B
            0>70 PRINT A,B
            0>80 DATA 10,20,10/2,20/2,SIN(PI),COS(PI)
            0>RUN

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

SEE ALSO: DATA, RESTORE

REM

SYNTAX : REM [*comment*]
MODE: Run, Command

REM is short for REMark. It does nothing, but allows the user to add comments to a program. Comments are usually needed to make a program more legible and easier to understand.

Once a REM statement appears on a line, the remainder of that line is assumed to be a remark so a REM statement can not be terminated with a colon ":" delimiter. REM statements can, however, be placed AFTER a colon delimiter, allowing the programmer to comment each program line.

```
example     0>10 REM ** Input a variable **
            0>20 INPUT A
            0>30 REM ** Input another variable **
            0>40 INPUT B
            0>50 REM ** Multiply the two **
            0>60 Z=A*B
            0>70 PRINT Z : REM ** Z is the answer, print it **
```

The following REM statement illustrates that executable statements can NOT be placed following a REM statement on the same line. The PRINT statement is considered part of the REM statement and is not executed.

```
example     0>10 REM ** print the number ** : PRINT A
            0>RUN
```

The REM statement can be used in COMMAND mode. This is an important feature for those who use a host computer as the console device to implement a program download routine. REM statements can be placed in the source program without line numbers. When the program is downloaded, the REM statements do not occupy any of the valuable program memory space.

RESTORE

SYNTAX : RESTORE
MODE: Run

The RESTORE statement "resets" the DATA pointer to the beginning of the program's first DATA statement so that the data can be read again.

example
```
0>10 FOR I=1 TO 3
0>20 READ A,B
0>30 PRINT A,B
0>40 NEXT I
0>50 RESTORE
0>60 READ A,B
0>70 PRINT A,B
0>80 DATA 10,20,10/2,20/2,SIN(PI),COS(PI)
0>RUN

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

SEE ALSO: DATA, READ

RETI

SYNTAX : RETI
MODE: Run

The RETI statement must be used in place of the RETURN statement in the ONPORT and ONTIME interrupt service subroutines! Failure to do this causes the ASCII BASIC Module to ignore future interrupts.

SEE ALSO: ONPORT, ONTIME

RIGHT$()

SYNTAX : RIGHT$(*string_expr, num*)
MODE: Run, Command

The RIGHT$() function returns a character string composed of the right-most *num* characters of the *string_expr*. The *num* parameter is a valid integer value between 1 and 255 inclusive. If *num* is greater than the number of characters in *string_expr*, all of the *string_expr* is returned.

example
```
0>PRINT RIGHT$("Horner Electric", 10)
r Electric

0>STRING 257,31
0>$(0)="This is a test"
0>PRINT RIGHT$($(0), 6)
a test
```

SEE ALSO: LEFT$(), MID$()

RND

SYNTAX : RND
MODE: Run, Command

The RND operator returns a random number in the range between 0 and 1 inclusive. The RND operator uses a 16-bit binary seed and generates 65536 random numbers before repeating the sequence. The numbers generated are specifically between 0/65536 and 65535/65536 inclusive. Unlike most BASICs, the RND operator provided by the ASCII BASIC Module does not require an argument seed.

example
```
0>PRINT RND
.2023926

0>PRINT RND
.6832341
```

RTRAP

SYNTAX : RTRAP *const*
MODE: Run, Command

It is possible to "trap" the ASCII BASIC Module in the RUN mode. This option is evoked by executing the **RTRAP 1** statement.

After this is done, the ASCII BASIC Program is "trapped" in RUN mode forever or until the **RTRAP 0** statement is executed. If no program is present when the "trap" is set, the ASCII BASIC module continuously prints the READY message until the device is RESET.

This option is normally used to prevent the application program from halting execution due to a BASIC error.

STOP! It is possible to configure a program to automatically run following RESET. If the program immediately executes a RTRAP statement and does not provide a means for disabling it, THERE IS **NO WAY FOR THE PROGRAMMER TO STOP THE PROGRAM**. For this reason, the programmer must provide a means for executing a **RTRAP 0** statement within the program. Schemes can be implemented very similarly to those discussed for the BREAK command.

SEE ALSO: BREAK, CLRMEM, AUTORUN, STARTUP

RTS

SYNTAX : RTS [#]
MODE: Run, Command

The RTS operator returns the state of a the specified serial port's hardware handshaking input.

If the RTS signal for the specified port is currently HIGH (+12V), the RTS operator returns a TRUE result (65535). If the RTS signal for the specified serial port is LOW (-12V), the RTS operator returns a FALSE (0) result.

If the "#" character is appended to the RTS operator, the RTS signal state for the AUXILIARY serial port is returned, otherwise the RTS signal state for the PRIMARY serial port is returned.

SEE ALSO: CTS

RUN operator

SYNTAX : RUN
MODE: Run, Command

The RUN operator returns the numeric value that reflects the "RUN" status of the PLC. If the host CPU is currently in RUN mode, the RUN operator returns a TRUE result (65535). Otherwise, the RUN operator returns a FALSE result (0).

example
```
0>PRINT RUN
 0

0>10 IF RUN THEN PRINT "PLC CPU is running the ladder program"
```

SETCOM

SYNTAX : SETCOM [#] *baud_const* [*, parity, data_bits, stop_bits, handshake, port*]
MODE: Run, Command

The SETCOM statement is used to configure the communications parameters used by the specified serial device. If the "#" character is appended to the SETCOM keyword, the AUXILIARY serial port is configured, otherwise the PRIMARY serial port is configured.

Only the *baud_const* parameter is mandatory, the remaining parameters are optional. Note, however that if an optional parameter is specified, all of the preceding parameters must also be specified. For example, if the *stop_bits* parameter is specified, the *parity* and *data_bits* parameters must also be specified.

The parameters are described in detail below:

baud_const: An numeric constant whose value represents the baud rate to be used by the specified port. Valid values are 300, 600, 1200, 2400, 4800, 9600, 19200 and 57600. The AUXILIARY port is configured to 9600 baud after RESET.

parity: A single character representing type of parity implemented. Valid values are "N" (no parity), "E" (Even parity), and "O" (Odd parity). All ports default to "N".

data_bits: A single character representing the number of data bits to be received and transmitted. Valid values are "7" and "8". All ports default to “8”.

stop_bits: A single character representing the number of stop bits to be received and transmitted. Valid values are "1" and "2". All ports default to "1".

handshake: A single character representing the type of serial handshaking to perform during transmission and reception. Valid values are "N" (No handshaking), "S" (Software XON/XOFF handshaking) and "H" (Hardware RTS/CTS handshaking). All ports default to "S".

port: A numeric constant whose value represents the auxiliary port to be used. Valid values are "0" (RS-232), "1" (RS-485) and "2" (Modem if installed). This parameter is only valid when used with the "#" character.

example
```
0>10 REM Set the PRIMARY port for terminal communications
0>20 SETCOM 9600, N, 8, 1, S,

0>30 REM Set the AUX port for OIT communications
0>40 SETCOM# 19200, E, 7, 1, H, 1
```

NOTE: When switching between auxiliary ports, 1 second should be allowed for buffer initialization. If communications are attempted in this time period it is possible to lose or receive incorrect characters. This can be accomplished by using a DELAY command.

SEE ALSO: SETINPUT

SETINPUT

SYNTAX : SETINPUT *no_echo* [*, no_edit, terminator, length, first_wait, next_wait*]

Normally when an INPUT statement is executed, it waits for a carriage return character before returning control to the BASIC program. The SETINPUT statement allows very versatile configuration of the INPUT statement.

Only the *no_echo* parameter is mandatory, the remain parameters are optional. Note, however that if an optional parameter is specified, all of the preceding parameters must also be specified. For example, if the *length* parameter is specified, the *no_edit* and *terminator* parameters must also be specified.

The parameters are described in detail below:

no_echo: When nonzero, characters received during the INPUT statement are not echoed to the transmitting device. When zero, all characters received during the INPUT are echoed. The default is 0, echo.

no_edit: When nonzero, the module stores the BACKSPACE (ASCII 8) and DEL (ASCII 127) characters as normal ASCII characters. When zero, the BACKSPACE and DEL characters perform the BACKSPACE operation, deleting the last character from the INPUT buffer. The default is 0, enable BACKSPACE and DEL editing.

terminator: An ASCII value (0 to 255) that is to be used as the INPUT termination character. The INPUT statement stops reception and return to the program when this character is received. The default is 13 (carriage return).

length: An expression whose value represents the maximum number of characters that the INPUT instruction is to receive. When this number of characters has been received, the INPUT statement stops reception and returns to the program. If length is 0 or greater than 79, the length parameter is disabled. The default value is 0.

first_wait: An integer expression whose value represents the number of milliseconds that the INPUT instruction is to wait for the first character. If no character is received within the specified time limit, the BASIC program resumes execution. If *first_wait* is set to 0, the module waits forever for the first character. The default value is 0.

next_wait: An integer expression whose value represents the number of milliseconds that the INPUT instruction is to wait for subsequent characters. If no character is received within the specified time limit, the BASIC program resumes execution. If *next_wait* is set to 0, the module waits for the next character. The default value is 0.

example

```
0>10 REM Disable character echo
0>20 SETINPUT 1

0>10 REM Set the terminating character to "="
0>20 SETINPUT 0, 0, 61

0>10 REM Set the max length to 20 and the first timeout to 3 seconds
0>20 SETINPUT 0, 0, 13, 20, 3000, 0
```

SEE ALSO: INPUT, SETCOM

SGN()

SYNTAX : SGN(*expr*)
MODE: Run, Command

The SGN() operator returns a value that represents the SIGN of the numeric *expr*. If the expression solves to a positive value, 1 is returned. If the expression solves to 0, 0 is returned. If the expression solves to a negative value, -1 is returned.

```
example     0>PRINT SGN(52)      0>PRINT SGN(0)      0>PRINT SGN(-33)
            1                    0                   -1
```

SIN()

SYNTAX : SIN(*expr*)
MODE: Run, Command

The SIN() operator returns the trigonometric SINE of the *expr*. The *expr* is a numeric expression that must solve to a value between +/- 200000. The calculation is carried out to seven significant digits.

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

SEE ALSO: ATN(), COS(), TAN(), PI

SIZE

SYNTAX : SIZE
MODE: Run, Command

The SIZE operator returns the number of bytes occupied by the currently selected program.

SIZE is a READ-ONLY value and cannot be assigned a value. Any attempt to do so generates a syntax error.

Note that the SIZE operator returns a value of 1 when no program exists. This is because all programs (even null programs) contain an "end of file" character.

```
example     0>10 FOR I=1280 TO 1334 : REM Display the program
            0>20 PRINT XBY(I),
            0>30 NEXT I
            0>PRINT SIZE
            54
```

SEE ALSO: FREE, MTOP

SPC()

SYNTAX : SPC(*expr*)
MODE: Run, Command

The SPC() function returns a character string comprised of the number of SPACE characters (ASCII 32) specified by the numeric *expr*.

example
```
0>PRINT SPC(20), "Horner", SPC(10), "Electric"
      Horner    Electric

0>STRING 257, 31
0>$(0)="This is"+SPC(10)+"a test"
0>PRINT $(0)
This is    a test
```

SEE ALSO: TAB()

SQR()

SYNTAX : SQR(*expr*)
MODE: Run, Command

The SQR() operator returns a square root of the *expr*. The *expr* is a numeric expression that must solve to a positive value. The value returned is accurate to +/- 5 on the least significant digit.

example
```
0>PRINT SQR(9)
3

0>PRINT SQR(45)
6.7082035

0>PRINT SQR(PI*PI)
3.1415926
```

SEE ALSO: EXP()

ST@

SYNTAX : ST@ *expr* [,*expr* ,*expr ...*]
MODE: Run, Command

The ST@ statement allows the programmer to store floating-point values anywhere in the module's DATA memory.

The *expr* is a numeric expression that represents the DATA memory address where the value is to be placed. The last value PUSHed onto the ARGUMENT STACK is copied to the specified DATA memory address.

Each floating-point value requires 6 bytes of storage. The expr represents the HIGHEST memory address used by the operation. For example, ST@ 32767 would actually cause the value to be stored at addresses 32767, 32766, 32765, 32764, 32763 and 32762.

The LD@ and ST@ statements are the only available means of passing variables between CHAINed programs.

STOP! The LD@ and ST@ statements allow manipulation of DATA memory and does not check the specified addresses. It is possible to specify addresses in DATA memory that interfere with ASCII BASIC Module firmware operations or with the RAM program or string variable operations. To avoid problems, the programmer must use the SPECIAL FUNCTION OPERATOR, MTOP to set aside a protected area of memory for use by these instructions.

example
```
0>10 REM ** Save array **
0>20 FOR I=0 TO 9
0>30 PUSH A(I) : REM ** Put array on arg stack **
0>40 ST@ 32767+(6*I) : REM ** Store it, 6 bytes per value **
0>50 NEXT I
0>60 REM ** Get array **
0>70 FOR I=0 TO 9
0>80 LD@ 32767+(6*I)
0>90 POP B(I)
0>100 NEXT I
```

SEE ALSO: LD@, POP, PUSH

STOP

SYNTAX : STOP
MODE: Run, Command

The STOP statement allows the programmer to break program execution at a specific point in the program. After the program is STOPped, variables can be examined and/or modified. Program execution can be resumed at the point that it was STOPped using the CONT command (provided that the program was not modified).

The STOP and CONT commands are invaluable program debugging tools, and programmers are encouraged to provide line number gaps in their programs for their implementation during program debugging.

When an executing program encounters a STOP statement, the line number following the STOP statement is displayed prior to entering the COMMAND mode.

example
```
0>10 FOR I=1 TO 100
0>20 PRINT I
0>30 STOP
0>40 NEXT I
0>RUN

1
STOP - In line 40

Ready
0>I=50

0>CONT

51
STOP - IN LINE 40
```

SEE ALSO: CONT

STRING

SYNTAX : STRING *total_bytes, max_string_size*
MODE: Run, Command

The STRING statement is used to allocate memory for character string storage. Initially, NO MEMORY is allocated for string storage. If the user attempts to define a string variable such as $(1)="HELLO", before the STRING statement has been used to allocate string memory, a Memory allocation error is generated.

The *total_bytes* numeric expression following the string statement represents the total number of bytes the user wishes to allocate for string storage. The *max_string_size* numeric expression represents the maximum number of bytes that are in each string.

The meaning of these parameters is a bit ambiguous. The ASCII BASIC Module requires one additional byte of storage for each string, plus one additional byte overall. This means that the statement STRING 100,10 would allocate enough memory for 9 ten character string variables and all 100 bytes would be used (((10+1)*9)+1).

The total number of bytes of string storage memory required (M) can be derived using the following formula, given the maximum number of characters for each string (L) and the total number of strings (S):

M = ((L + 1) * S) + 1

(S) can not exceed 254 and the maximum value for (L) is limited only to the amount of available memory.

example
```
>10 S=25 : REM ** 25 strings **
>20 L=80 : REM ** 80 characters each **
>30 STRING (((L+1)*S)+1), L
```

STOP! Whenever the STRING statement is executed, the equivalent of a CLEAR statement is also executed. The STRING statement should be executed as early as possible in the program (after modifying MTOP but before execution of any DIM statements). The only way to DEALLOCATE string storage is to execute another STRING statement. The CLEAR statement won't affect string allocation.

STR$()

SYNTAX : STR$(*expr*)
MODE: Run, Command

The STR$() function returns the string representation of the value of the numeric *expr.*

Note that if the result is positive, the string returned contains a leading space.

example
```
0>PRINT STR$(PI)
3.1415926

0>STRING 257,31
0>$(0)=STR$(INT(RND*100))
0>PRINT $(0)
32
```

SEE ALSO: VAL()

TAB()

SYNTAX : TAB(*expr*)
MODE: Run, Command

The TAB() statement is a special PRINT statement formatting option specifier and can ONLY appear following a PRINT statement. The TAB() function is used to cause the next PRINTed item to be displayed at the column specified by the numeric *expr* on the output device.

Each time a character is transmitted from one of the serial ports, the printhead position is incremented. When the TAB() function is used, the module outputs the correct number of spaces require to move the printhead to the specified column. When a carriage return is transmitted, the "printhead" position is reset.

Note that the printhead position is maintained for both the PRIMARY and AUXILIARY serial devices separately.

If the *expr* specifies a position less than the CURRENT print position, the TAB function is ignored.

example
```
0>10 FOR I=1 TO 3
0>20 PRINT TAB(5), I, TAB(10), I*I
0>30 NEXT I
0>RUN

 1   1
 2   4
 3   9
```

SEE ALSO: SPC(), PRINT

TAN()

SYNTAX : TAN(*expr*)
MODE: Run, Command

The TAN() operator returns the trigonometric TANGENT of the *expr*. The *expr* is a numeric expression that solves to a value between +/- 200000. The calculation is carried out to seven significant digits.

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

SEE ALSO: ATN(), COS(), SIN(), PI

TIME

SYNTAX : TIME
MODE: Run, Command

TIME is a special function operator that is used to assign or retrieve a value to the millisecond clock. Following a power-up or reset, TIME is assigned a value of ZERO. After execution of the CLOCK1 statement, the TIME operator is incremented once every 5 milliseconds. The unit of TIME is seconds, when TIME reaches a value of 65535.995 seconds, it overflows back to a count of zero.

When the TIME operator is assigned a value, only the integer portion of TIME is affected. To assign a value to the fractional portion of the TIME operator, the FTIME operator must be used.

example
```
0>PRINT TIME
0

0>CLOCK 1
0>PRINT TIME
.735

0>PRINT TIME
1.24
```

SEE ALSO: CLOCK, FTIME, ONTIME

UCASE$()

SYNTAX : UCASE$(*string_expr*)
MODE: Run, Command

The UCASE$ function returns the *string_expr* with all of the alphabetic characters converted to upper case.

example
```
0>PRINT UCASE$("THIS is A tEsT")
THIS IS A TEST

0>STRING 257,31
0>$(0)="HorNEr ELEctRIc"
0>PRINT UCASE$($(0))
HORNER ELECTRIC
```

SEE ALSO: LCASE$()

USING()

SYNTAX : PRINT USING (*format*), [*expr_list*]
MODE: Run, Command

The USING() statement is a special PRINT statement formatting option specifier and can ONLY appear following a PRINT statement. The USING() function is used to cause numeric data displayed in a predefined decimal *format*. When a USING() option is invoked, the desired format is stored and used for all subsequent numeric displays until a new USING format is specified or the program terminates.

The USING keyword can be abbreviated "U.". The following *format*s are available with the USING statement.

USING(F*x*) : This forces all numeric data to be displayed in exponential floating-point format. The value of "*x*" determines how many significant digits of the mantissa is PRINTed. If *x* is zero, no trailing zeros are displayed and the number of significant digits displayed is dependent on the value. Otherwise, The module displays at least 3 significant digits, even if *x* is 1 or 2. The maximum value for *x* is 8.

example
```
0>10 PRINT USING(F3), 1, 2, 3
0>20 PRINT USING(F4), 1, 2, 3
0>30 PRINT USING(F5), 1, 2, 3
0>40  FOR I=10 TO 30 STEP 10
0>50  PRINT I
0>60  NEXT I
0>RUN

 1.00 E 0 2.00 E 0 3.00 E 0
 1.000 E 0 2.000 E 0 3.000 E 0
 1.0000 E 0 2.000 E 0 3.000 E 0
 1.0000 E+1
 2.0000 E+1
 3.0000 E+1
```

USING(#.#) : This forces all numeric data to be displayed in an integer and/or fractional format. The number of "#"s that appear before the decimal point determine how many significant digits of the integer portion is displayed. The number of "#"s that appear following the decimal point determine how many significant fractional digits is displayed. The decimal point can be omitted, in which case only the integer portion of the value is displayed. The maximum number of "#" characters that can appear is 8. If the value to be displayed is too large to fit in the specified format, a question mark (?) is printed and the valued is displayed in the USING(0) format, described below. Leading integer zeroes are suppressed.

example
```
0>10 PRINT USING(##.##), 1, 2, 3
0>20  FOR I=1 TO 120 STEP 20
0>30  PRINT I
0>40  NEXT I
0>RUN

 1.00  2.00  3.00
 1.00
 21.00
 41.00
 61.00
 81.00
? 101
```

USING(0) : This argument lets the ASCII BASIC Module determine what format to use. The rules are simple, if the number is between +/- 99999999 and +/- .1, the module displays integers and fractions. If it is out of this range, the module uses the USING(F0) format. Leading and trailing zeros are always suppressed. The module selects this format following RESET.

SEE ALSO: PRINT, SPC(), TAB()

VAL()

SYNTAX : VAL(*string_expr*)
MODE: Run, Command

The VAL() function returns the numeric value of the *string_expr*. The *string_expr* should be a sequence of characters that can be interpreted as a numeric value. The VAL() function stops reading the *string_expr* at the first character that is non-numeric.

The VAL() function ignores leading SPACE and TAB characters. If the first non-white character of the *string_expr* is non-numeric, the VAL() function returns zero.

example
```
0>STRING 257,31
0>$(0)=" 3.14 and more characters"
0>PRINT VAL($(0))
 3.14

0>PRINT VAL(STR$(PI))
 3.1415926
```

SEE ALSO: STR$()

XBY()

SYNTAX : XBY(*address*)
MODE: Run, Command

The XBY() operator is used to assign or retrieve a value to one of the battery-backed DATA memory or PROGRAM FILE memory locations in the ASCII BASIC module. The XBY operator returns and expects an 8-bit value ranging from 0 to 255 inclusive. The *address* parameter is a numeric expression and must solve to an integer value between 0 and 65535 inclusive.

This operator is useful in applications that use several different programs in the PROGRAM file. Data can be placed in the DATA memory using the XBY operator and then retrieved by a program invoked with the CHAIN statement.

> **STOP!** The DATA memory used by the ASCII BASIC firmware varies between models. However ALL models reserve the first 1280 bytes of DATA memory for important system usage. If the data at these addresses is modified by the ASCII BASIC program, the results are **UNPREDICTABLE.**

If an ASCII BASIC program 0 is present in DATA memory, this program is located starting at address 1280 through the size of the program. The LEN, FREE and MTOP operators can be used to determine how much unused DATA memory is available for general XBY access.

example
```
0>10 MTOP = 32700 : REM Protect upper DATA memory
0>20 STRING 2026,80
0>30 $(0)="This is a string"
0>40  FOR I=1 TO 16 : REM Store string in protected memory
0>50  XBY(32701+I)=ASC($(0),I)
0>60  NEXT I
0>70 CHAIN 2 : REM Program 2 can now access data stored.
```

SEE ALSO: SIZE, FREE, MTOP, LD@, ST@

.XOR.

SYNTAX 1 : *var* = *expr1* .XOR. *expr2*
SYNTAX 2 : *rel_expr1* .XOR. *rel_expr2*
MODE: Run, Command

Using syntax 1, a bit-wise XOR function is performed on the two expressions and the result is placed in the *var*. Each binary bit of the two expressions is manipulated as shown in the truth table below:

EXPR1	EXPR2	RESULT
0	0	0
0	1	1
1	0	1
1	1	0

example
```
0>PRINT 2.XOR.3
1
```
```
0>PH0. 55H.XOR.0C0H
95H
```

Using syntax 2, a logical XOR function is performed on the two relational expressions. If one of the relational expressions is TRUE and the other is FALSE, a TRUE result (65535) is returned. If both relational expressions are TRUE or both are FALSE, a FALSE result (0) is returned.

example
```
0>PRINT (2=2).XOR.(3=3)
0
```
```
0>PRINT (2=2).XOR.(3=2)
65535
```

SEE ALSO: .AND., .OR., NOT()

4.3 Interrupt Priority

All three of the BASIC interrupts (ONPORT, ONPORT#, and ONTIME) can be armed concurrently. The ONTIME interrupt has the highest priority, this means that if an ONPORT or ONPORT# interrupt service subroutine is being executed when an ONTIME interrupt occurs, the ONTIME interrupt is immediately executed. When the ONTIME interrupt service subroutine RETI instruction is executed, control is passed back to the ONPORT service subroutine.

If the ONTIME interrupt is being executed, it runs until it's RETI instruction is encountered. The ONPORT interrupts cannot supersede the ONTIME interrupt.

The ONPORT and ONPORT# interrupts share equal priority. This means that if one of the interrupt service subroutines is being executed and the other interrupt occurs, the second interrupt service subroutine is not executed until the active interrupt service subroutine's RETI statement is encountered. If both the ONPORT and ONPORT# interrupts occur simultaneously, the ONPORT interrupt takes priority over the ONPORT# interrupt.

NOTES

CHAPTER 5: ARITHMETIC AND RELATIONAL OPERATORS

5.1 Operator precedence

The hierarchy of mathematics dictates that some operations are carried out before others. If you understand the hierarchy of mathematics, it is possible to write complex expressions using only a minimal amount of parenthesis. It's easy to illustrate what precedence is. For example:

4 + 3 * 2 = ?

Should you add (4 + 3) and then multiply seven by 2 or should you multiply (3 * 2) then add 4? The hierarchy of mathematics dictates that multiplication has precedence over addition, so the answer is

4 + 3 * 2 = 10.

The rules for this hierarchy are simple. 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, the addition could not be performed because the multiplication has higher precedence.

In the ASCII BASIC Module, the precedence of operators from highest to lowest is as follows:

1. Operators that use parenthesis ().
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.)

Whenever in doubt about the rules for operator precedence, use parenthesis.

5.2 Arithmetic Operators

The arithmetic operators supported by the ASCII BASIC Module are listed below:

+ (addition) **- (subtraction)** *** (multiplication)**
/ (division) **** (exponentiation)**

(+) Addition Operator

The addition operator, when used in a numeric expression, returns the sum of the two operands.

```
example     0>PRINT 3+2                    0>PRINT PI+5
             5                              8.1415926
```

When used in a string expression, the addition operator concatenates the two string operands (See chapter 6 for more details).

```
example     0>PRINT "This is "+"a test"
            This is a test
```

(-) Subtraction Operator

The subtraction operator returns the difference of the two numeric operands.

```
example     0>PRINT 3-2                    0>PRINT PI-2
             1                              1.1415926
```

(*) Multiplication Operator

The multiplication operator returns the product of the two numeric operands.

```
example     0>PRINT 3*2                    0>PRINT PI*4
             6                              12.566370
```

(/) Division Operator

The division operator returns the quotient of the two numeric operands.

```
example     0>PRINT 3/2                    0>PRINT PI/2
             1.5                            1.5707963
```

(**) Exponentiation Operator

The Exponentiation operator returns the value of the first operand raised to the power of the second operand.

```
example     0>PRINT 3**2                   0>PRINT PI**3
             9                              31.006275
```

5.3 Relational operators

Relational expressions involve the operators listed below. These operators are typically used to “test” a condition. Unlike most BASICs, the ASCII BASIC Module returns a result of TRUE (65535) when the relational expression is true or FALSE (0) if the relational expression is false.

It can seem strange to have a relational expression actually return a result, but is offers a unique benefit in that relational expressions can actually be "chained" together using the LOGICAL operators .AND., .OR., .XOR. and NOT(). This makes it possible to test a rather complex condition with ONE statement.

```
example          0>10 IF A<B .AND. A>C .OR. A>D THEN...
```

Additionally, the NOT() operator can be used to "invert" the result of a relational expression.

```
example          0>10 IF NOT(A<B) .AND. A>C THEN...
```

When using logical operators to link together relational expressions, it is very important that the programmer pay careful attention to the precedence of operators. The logical operators were assigned lower precedence, relative to the relational operators, just to make the linking of relational expressions possible without using parentheses.

The relational operators supported by the ASCII BASIC Module are:

= **(equal)**
< **(not equal)**
> **(greater than)**
> **(less than)**
>= **(greater than or equal)**
<= **(less than or equal)**

(=) Equal operator

Used to test equality. If the two expressions are equal, a TRUE result (65535) is returned, otherwise a FALSE result (0) is returned.

```
example          0>PRINT 0=0                          0>PRINT 1=0
                  65535                                0

                 0>10 IF 0=0 THEN PRINT "Equal" ELSE PRINT "Not equal"
```

(<>) Not equal operator

Used to test the relation of two values. If the first expression is NOT EQUAL to the second expression, a TRUE result (65535) is returned, otherwise a FALSE result (0) is returned.

```
example          0>PRINT 0<>0                         0>PRINT 1<>0
                  0                                    65535

                 0>10 IF 0<>0 THEN PRINT "Not equal" ELSE PRINT "Equal"
```

(>) Greater than operator

Used to test the relation of two values. If the first expression is GREATER THAN the second expression, a TRUE result (65535) is returned, otherwise a FALSE result (0) is returned.

```
example     0>PRINT 0>0                          0>PRINT 1>0
             0                                    65535

            0>10 IF 0>0 THEN PRINT "Greater than" ELSE PRINT "Less than or equal to"
```

(<) Less than operator

Used to test the relation of two values. If the first expression is LESS THAN the second expression, a TRUE result (65535) is returned, otherwise a FALSE result (0) is returned.

```
example     0>PRINT 0<0                          0>PRINT 0<1
             0                                    65535

            0>10 IF 0<0 THEN PRINT "Less than" ELSE PRINT "Greater than or equal to"
```

(>=) Greater than or equal operator

Used to test the relation of two values. If the first expression is GREATER THAN OR EQUAL TO the second expression, a TRUE result (65535) is returned, otherwise a FALSE result (0) is returned.

```
example     0>PRINT 0>=0                         0>PRINT 0>=1
             65535                                0

            0>10 IF 0>=0 THEN PRINT "Greater than or equal to" ELSE PRINT "Less than"
```

(<=) Less than or equal operator

Used to test the relation of two values. If the first expression is LESS THAN OR EQUAL TO the second expression, a TRUE result (65535) is returned, otherwise a FALSE result (0) is returned.

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

            0>10 IF 0>=0 THEN PRINT "Less than or equal to" ELSE PRINT "Greater than"
```

CHAPTER 6: STRING HANDLING

6.1 What are STRINGS?

A STRING is a character or several characters that are stored in memory. Usually, the characters stored in a string make up a word or sentence. Strings are useful because they allow the programmer to deal with words instead of numbers, an invaluable aid to writing "user-friendly" programs.

The ASCII BASIC Module supports a ONE dimensional string variable, $(*expr*). The dimension of the string value (the *expr* value) ranges from 0 to 254. This means that 255 different strings can be defined and manipulated in BASIC.

Initially, NO memory is allocated for string storage. Memory is allocated for string storage using the STRING statement, described in chapter 4.

There are several operators discussed in chapter 4 that are used to manipulate strings:

CHR()	**INKEY$**	**LEN()**	**STRING**
CHR$()	**INSTR()**	**MID$()**	**STR$()**
DATE$	**LCASE$()**	**RIGHT$()**	**TIME$**
INBUF$	**LEFT$()**	**SPC()**	**UCASE$()**

6.2 Combining Strings

The ASCII BASIC module allows string concatenation using the addition operator (+). Whenever a string expression is required by a string operator, the addition operator can be used to combine two or more strings.

example
```
0>STRING 257,31
0>$(0)="This is " + "a test"
0>PRINT $(0)
This is a test
```

This feature allows quite complex string manipulation WITHIN the string operators.

example
```
0>PRINT LEFT$(MID$(UCASE$("GE" + " " + "Fanuc"), 8, 4), 3)
ELE
```

The CR and SPC() operators can also be used in string concatenation.

example
```
0>PRINT "This is" + CR + CHR$(10) + "really" + SPC(12) + "a test"
This is
really a test
```

6.3 How Strings are Stored

Character string variables used in an ASCII BASIC program are allocated memory using the STRING statement (discussed in Chapter 4). When the STRING statement is executed, the module allocates the specified amount of memory starting from MTOP down. For example:

```
example          0>STRING 257, 31
```

The example allocates 257 bytes of memory for string storage. If MTOP is set to 32767, string memory begins at (32767-257) or 32510. All of the memory from this address through the MTOP address is allocated for string storage. The first string variable ($(0)) begins at 32510 and occupies 32 bytes. The first byte (at 32510) is reserved as the "length" byte for the $(0) variable. This byte contains the number of characters contained the string variable. Initially, the length of all strings is set to 0. The remaining bytes (from 32511 through 32542) contain the characters that comprise the $(0) string variable. The second string variable ($(1)) immediately follows at address 32543.

Note that no terminating character is used and that all ASCII values from 0 to 255 inclusive are valid string components.

6.4 Strings in Relational Expressions

The relational operators (=, <>, >, <, >= and <=) can be used to compare the characters in two string expressions. When used with string expressions, relational expressions returns a value (TRUE or FALSE) exactly as when used with numeric expressions (see chapter 5).

Parenthesis are NOT ALLOWED nor are they necessary when defining a relational string expression.

```
example          0>10 IF ($(0) = "TEST") THEN PRINT "Equal"
                 0>RUN

                 ERROR! Invalid syntax! - In line 10
                 10 IF ($(0)="TEST") THEN PRINT "Equal"
                 ----------X
```

The relational operators perform a character by character comparison of the two string expressions.

string_expr1 = string_expr2

When using the "=" operator in a relational string expression, if the two string expressions are identical (every character is the same and the string lengths are equal), then a TRUE (65535) result is returned. If the string expressions are in any way different, a FALSE (0) result is returned.

```
example          PRINT "TEST" = "TEST"
                  65535

                 PRINT "TEST" = "TEST1"
                  0

                 10 IF "TEST" = "TEST1" THEN PRINT "Equal" ELSE PRINT "Not Equal"
```

string_expr1 <> string_expr2

The "<>" operator is the complement of the "=" operator, if the two string expressions are identical (every character is the same and the string lengths are equal), then a TRUE (65535) result is returned. If the string expressions are in any way different, a FALSE (0) result is returned.

When using the >, <, <= or >= operators in a relational string expression, the two string expressions are compared character by character until a "non-match" is encountered (or until the end of one of the strings is reached). If a character non-match is found, the ASCII values of the two characters are compared and the result is based on these values. If the end of one of the strings is reached, the result is based on the comparison of the string lengths.

string_expr1 > string_expr2

example	0>PRINT "TEST" > "test" 0	0>PRINT "TEST" > "TEST" 0	0>PRINT "test" > "TEST" 65535

string_expr1 < string_expr2

example	0>PRINT "TEST" < "test" 65535	0>PRINT "TEST" < "TEST" 0	0>PRINT "test" < "TEST" 0

string_expr1 >= string_expr2

example	0>PRINT "TEST" >= "test" 0	0>PRINT "TEST" >= "TEST" 65535	0>PRINT "test" >= "TEST" 65535

string_expr1 <= string_expr2

example	0>PRINT "TEST" <= "test" 65535	0>PRINT "TEST" <= "TEST" 65535	0>PRINT "test" <= "TEST" 0

NOTES

CHAPTER 7: ERROR HANDLING

7.1 Error Messages

The ASC100 provides a relatively sophisticated ERROR processor. When an error is encountered in an executing BASIC program, the module generates an error message in the following format:

```
example     ERROR: XXX - In line YYY
            YYY BASIC STATEMENT
            ————————————X
```

Where XXX is the TYPE of ERROR and YYY is the line number in the program where the error occurred. A specific example is:

```
example     ERROR! Invalid syntax! - In line 100
            100 PRINT I*4*
            ————————————X
```

Notice that a dashed line followed by an "X" is generated below the error-ridden line. The "X" signifies approximately where the ERROR occurred in the BASIC line. This location can be off by one or two characters or expressions, depending on the type and location of the error encountered. If an error is encountered while in COMMAND mode, only the error TYPE is displayed - not the line number or the pointer line.

Invalid syntax

An Invalid syntax error means that either an invalid ASCII BASIC COMMAND, STATEMENT or OPERATOR was entered and BASIC cannot process the entry. The user should check to insure that the line was typed correctly, and that no imbedded BASIC keywords appear in any user variable names.

Invalid argument

An Invalid argument error means that the argument of an operator is not within the limits of that operator. For example, BCD(10) generates an Invalid argument error because 10 can not be converted to a legal BCD value. Similarly, OUT(0)=-1 generates an Invalid argument error, because the assignment argument for the OUT() operator must be between 0 and 65535.

Arithmetic underflow

If the result of an arithmetic operation exceeds the lower limit of an ASCII BASIC floating-point number, an Arithmetic underflow error is generated. The smallest floating-point number that the ASCII BASIC Module can process is + or - 1E-127.

Arithmetic overflow

If the result of an arithmetic operation exceeds the upper limit of an ASCII BASIC floating-point number, an Arithmetic overflow error is generated. The largest floating-point number that the ASCII BASIC Module can process is + or - .99999999E+127.

Division by zero

If zero appears as the denominator in a division operation, a Division by zero error is generated.

Out of data

If a READ statement is executed and no DATA statement exists or all of the data in the DATA statement(s) has been READ without execution of a RESTORE statement, an Out of data error is generated.

Can’t continue

Program execution can be halted by either typing a CONTROL-C to the console device or by execution of a STOP statement. Normally, program execution can be resumed by executing the CONT command, however, if the user modifies the program after halting execution and attempts to execute the CONT command, the CAN’T CONTINUE error is generated.

While programming

If an error occurs while the ASCII BASIC Module is storing a program into the PROGRAM FILE memory, this error is generated. This error should only occur when the program being stored is larger than the available PROGRAM FILE memory. If this error occurs, the PROGRAM FILE structure is disrupted and the user is not able to save any further programs in the PROGRAM FILE memory.

Argument stack overflow

If the ARGUMENT STACK pointer is forced "out-of-bounds", an Argument stack error is generated. This can happen if the user attempts to PUSH too many values onto the ARGUMENT STACK or by attempting to POP data from the ARGUMENT STACK when no data is present.

Control stack overflow

If the CONTROL STACK pointer is forced "out-of-bounds", a Control stack error is generated. 158 bytes of memory are allocated to the CONTROL STACK, FOR-NEXT loops require 17 byte of CONTROL STACK, DO-UNTIL and DO-WHILE and GOSUB statements require 3 bytes of CONTROL STACK. If too many "nested" loops are implemented, the CONTROL STACK overflows, and the Control stack error is generated. Additionally, if a NEXT statement is executed before a FOR statement or if an UNTIL or a WHILE statement are executed before a DO or if a RETURN is executed prior to a GOSUB, this error occurs.

Internal stack overflow

The Internal stack overflow error indicates that the ASCII BASIC module has run out of internal "expression" analysis stack space. This error should never occur, if it does, simplify the expression that generates the error.

Array size exceeded or not specified

If an array is dimensioned by a DIM statement and then the user attempts to access a variable that is outside the dimensioned bounds, this error is generated.

Memory allocation

This error is generated when the user attempts to access strings that are "outside" of the defined string limits. Additionally, if the MTOP system control value is assigned to a value greater than the available DATA memory, this error is generated.

Invalid line number

This error only occurs if the program structure with a BASIC program has been corrupted. This does not normally occur, but if the XBY() or ST@ statements are used to store data in the area of memory reserved for the PROGRAM file, the BASIC program(s) might be corrupted.

Only program 0 may be edited

This error is generated whenever a BASIC program line is entered while a program other than program number 0 is selected. The COMMAND mode prompt displays the number of the currently selected program.

Nothing to save

This error is generated whenever an attempt is made to SAVE a null program.

Specified program does not exist

This error is generated whenever the argument to the SELECT, DELPGM or EDIT commands specify a program that does not exist.

7.2 Warning messages

The following WARNING messages are displayed under certain circumstances but do NOT cause an executing BASIC program to terminate.

WARNING! Extra input ignored!

This message is displayed whenever more numeric values are entered during an INPUT statement than are required. For example, if 3 variables are listed as the target to an INPUT statement and 5 values are entered when the INPUT statement executes, this warning is displayed.

WARNING! String length exceeded, destination string truncated!

This warning message is displayed whenever too few numeric values are entered during an INPUT statement or when an attempt is made to store more characters in a string variable than have been configured using the STRING statement.

CHAPTER 8: OCS/RCS INTERFACE

8.1 ASCII BASIC Register Mapping

Chapter Eight provides information for an important feature of the ASC100 - the interface between the ASC100C module and the OCS/RCS. There are two operators that allow data transfer between the ASCII BASIC module and the OCS/RCS. The operators are the **INP()** operator and the **OUT()** operator (see Chapter 4).

The ASCII BASIC module shares up to sixty-four 16-bit input registers (AI) and sixty-four 16-bit output registers (AQ) with the OCS/RCS. The type of data placed in these registers is completely dependent on the application. There are no predefined special function registers.

The ASCII BASIC module is an intelligent I/0 module as described in the OCS/RCS documentation. The ASCII BASIC module's registers are mapped in the "%AI" and "%AQ" space exactly as in the same manner as the High Speed Counter module. This means that the ASCII BASIC module's I/O registers can be mapped anywhere in the %AI and %AQ space.

8.2 Asynchronous Program Execution

The ASCII BASIC program runs completely independent of the OCS/RCS ladder program. The ASCII BASIC program begins execution at the first program line (the line number with the smallest value) and continues executing as the BASIC instructions direct the program flow. The OCS/RCS ladder program begins execution with the first rung of ladder logic and continues through the entire ladder, manipulating it's I/O point and registers as directed by the ladder program.

8.3 Register usage

In most applications, the sixty-four input registers and sixty-four output registers provide adequate communications between the ASCII BASIC module and the OCS/RCS. A particular register can be defined to contain a specific piece of information at all times.

For example, the ASCII BASIC program can be written to perform a PID function (processing data from an analog input module and returning data to be sent to an analog output module). In its simplest form, only one of the INP() registers and one of the OUT() registers need to be used. The INP() register is written to by the OCS/RCS as the ladder program is scanned and is read by the ASCII BASIC module periodically each time the PID loop is executed. Conversely, the ASCII BASIC program writes to the OUT() register once each time the PID loop is performed while the OCS/RCS ladder program is continuously read the value and copies it to the analog output module.

There are no ill affects if the PID loop is only executed once each several seconds since the registers passed between the ASCII BASIC module and the OCS/RCS always contain the most recent "reading" for the same type of data. The INP() register contains the most recent analog input value while the OUT() register contains the most recent value to be sent to the analog output.

The interface becomes more complex if a large quantity of data must be shared between the ASCII BASIC module and the OCS/RCS. For example, assume that the ASCII BASIC program is to perform several PID loops and that the PID gain values are also to be passed from the OCS/RCS to the ASCII BASIC module. In this case, the sixty-four input registers and sixty-four output registers need to be multipurpose.

8.4 Using Register Protocol

When more data is to be passed between the ASCII BASIC module and the OCS/RCS than the amount of data that fits in the eight input and eight output registers, a communications protocol must be established in the ASCII BASIC and ladder programs. Since the OCS/RCS ladder program and the ASCII BASIC program execute asynchronously, the protocol must be used to synchronize the transfer of the various sets of data between the two programs.

As the OCS/RCS ladder program begins, it initially writes the first PID channel data to the ASCII BASIC module's INP() registers. The BASIC module reads the data and begins processing the first PID loop. If the OCS/RCS program blindly updates the BASIC module's INP() registers with the second PID channel data (without some kind of verification from the module that the data previously written has been read), the OCS/RCS program might overwrite the INP() registers *before* the BASIC module has finished reading them. Additionally, the OCS/RCS program might attempt to read the BASIC module's OUT() registers WHILE the module is in the process of updating them. The OCS/RCS program might read half of the registers that pertain to one of the PID channels and half that pertain to another.

To avoid this problem, the programs need to use one of the INP() registers and one of the OUT() registers as protocol variables.

An **example implementation** is to use the "bits" of one of the registers as status flags. For instance, if the OUT(7) register is used as the ABM to OCS/RCS protocol variable, it could be defined as follows:

OUT(7,0) This bit is used to tell the OCS/RCS that new OUT() data is available. It is set after the ABM has finished writing the new data to the OUT() registers, and cleared when the OCS/RCS has read the OUT() register data (when INP(7,1) gets set by the OCS/RCS).

OUT(7,1) This bit is used to tell the OCS/RCS that the ABM has read the new INP() register data. This bit is set when the ABM has finished reading the INP() register data and cleared when the OCS/RCS signifies that new INP() data is available (when INP(7,0) gets set by the OCS/RCS).

Similarly, the INP(7) register can be used as the "PLC to ABM" protocol register:

INP(7,0) This bit is used to tell the ABM that new INP() data is available. It is set after the OCS/RCS has finished writing the new data to the INP() registers, and cleared when the ABM has read the INP() register data (when OUT(7,1) gets set by the ABM).

INP(7,1) This bit is used to tell the ABM that the OCS/RCS has read the new OUT() register data. This bit is set when the OCS/RCS has finished reading the OUT() register data and cleared when the ABM signifies that new OUT() data is available (when OUT(7,0) gets set by the ABM).

The following is an example of a BASIC program for this protocol implementation.

example:

```
10      REM Example PID program
20      OUT(7,1)=0 : OUT(7,0)=0         : REM Clear the protocol bits
30      GOSUB 1000                      : REM Go read the INPut registers
40      GOSUB 2000                      : REM Go perform the PID function
50      GOSUB 3000                      : REM Go write the OUTput registers
60      GOTO 30                         : REM Go do it again.

1000    DO : UNTIL INP(7,0)                     : REM Wait for the OCS/RCS to update the registers
1010    OUT(7,1)=0                      : REM Tell the OCS/RCS that the registers are busy
1020    FOR X=0 TO 6                    : REM Read the INPuts
1030    PIDIN(X)=INP(X)
1040    NEXT X
1050    OUT(7,1)=1                      : REM Tell the OCS/RCS that registers are available
1060    RETURN

2000    REM PID loop goes here
2010    RETURN

3000    DO : UNTIL INP(7,1)                     : REM Wait for OCS/RCS to finish with registers
3010    OUT(7,0)=0                      : REM Tell the OCS/RCS that the registers are busy
3020    FOR X=0 TO 6                    : REM Write the OUTputs.
3030    OUT(X)=PIDOUT(X)
3040    NEXT X
3050    OUT(7,0)=1                      : REM Done with registers.
3060    RETURN
```

NOTES

CHAPTER 9: GETTING STARTED

Chapter Nine takes an ASCII BASIC programmer through steps required to enter, edit, store and execute an ASCII BASIC program.

9.1 Prepare to Use the Module

This chapter assumes that the user has the ASCII BASIC Module's primary port connected to an IBM PC or compatible computer running the supplied TERM.EXE dumb terminal emulation program (See Appendix E).

After executing the TERM.EXE program on the host computer, a sign-on message appears and then the CONFIGURATION menu appears. If this menu does not appear, simply press F1. The default configuration is used and sets the communication parameters as follows:

COM port:	COM1 (this should be set to the port that the user is using)
Baud rate:	9600
Parity type:	No parity
Data bits:	8
Stop bits:	1
Handshake type:	XON / XOFF
Display mode:	ASCII

The ENTER key on the keyboard can be pressed to invoke these parameters and initiate the "terminal" mode.

At this point, the ASCII BASIC Module RESETs. After installing the ASCII BASIC Module, turn the power to the OCS/RCS rack on. This causes the module to reset.

Following the RESET (after the OK light located on the OCS/RCS turns on), the ASCII BASIC module performs its reset sequence and then enters its baud rate detection mode. The **FIRST** character received by the module must be a **SPACE** character in order for the module to properly calculate the baud rate and initiate communication. When the SPACE bar is pressed on the host computer's keyboard, the module responds with the following sign on\status message:

```
SmartStack™ ASCII Basic Co-processor Module

DATA MEMORY:
 32K bytes present, from 0 to 32767 (7FFFH).
 No program exists in DATA memory, 1537 bytes occupied.
 MTOP = 32767 (7FFFH).
 31231 bytes free.

PROGRAM FILE MEMORY:
 32K bytes present, from 32768 (8000H) to 65471 (FDFFH).
 0 program(s) exist in PROGRAM FILE memory, 16 bytes occupied.
 32239 bytes free.

SYSTEM STATUS:
 AUTORUN: Program number for automatic execution is 0.
 STARTUP: Startup mode is set to 0.
 BREAK: Control-C break checking is enabled.
 CLRMEM: Data memory initialization is enabled.
 BAUD:  Stored primary port baud rate is 9600.

Ready
0>
```

If the module responds erratically, reset the module and try again. If the response is still erratic, recheck the communication parameters and try again. The OCS must be in the IDLE mode during this process.

9.2 Entering a Simple Program

After the "0>" prompt character is displayed, enter and **LIST** the following simple program:

```
example     0>10 P.
            0>65535 P.
            0>LIST

            10  PRINT
            65535 PRINT

            Ready
            0>
```

Now, RESET the ASCII BASIC module (turn the power off, then on) and press the space bar once again. The module responds once again with the sign-on message. At this point, attempt to list the program entered previously:

```
example     0>LIST

            Ready
            0>
```

9.3 Saving a Program in DATA Memory

Notice that the program entered previously is GONE. This is because the ASCII BASIC module clears its DATA memory following a RESET. To avoid this problem, enter the following commands prior to RESETting the module:

```
example     0>STARTUP 1
            0>CLRMEM 0

            0>10 P.
            0>65535 P.
            0>LIST

            10  PRINT
            65535 PRINT

            Ready
            0>
```

Now, RESET the ASCII BASIC module again. This time, the space bar need not be pressed to produce the sign-on message, as the baud rate information was saved when the STARTUP 1 command was entered. The module automatically initializes itself with the stored baud rate and immediately generate the sign-on message with no interaction.

List the program, notice that this time the program "survived" through the RESET sequence (thanks to the CLRMEM 0 command entered earlier):

```
example     0>LIST

            10 PRINT
            65535 PRINT

            Ready
            0>
```

9.4 Using the PROGRAM FILE memory

Using the SAVE command, programs entered into the DATA memory can be more permanently stored into the PROGRAM FILE memory. Erase the program entered earlier, and enter the new program as shown:

```
example     0>NEW

            0>10 P."This is my first ASCII BASIC program!"
            0>RUN

            This is my first ASCII BASIC program!

            Ready
            0>SAVE
             1

            Ready
            0>
```

The SAVE command was used to copy program 0 from DATA memory into the PROGRAM FILE memory. The module responded with the "1" following the SAVE command to indicate the program's number in the program file memory. Since no programs existed prior to the execution of the SAVE command, this program was assigned to number 1 or the first program stored in the PROGRAM FILE.

At this point, two identical copies of the program exist in the ASCII BASIC module, the original copy still exists in program 0 in DATA memory while the PROGRAM FILE memory contains a second copy.

Now, use the CLRMEM1 command to restore the ASCII BASIC module to it's original RESET configuration. (This means that DATA memory is once again be CLEARED following a RESET).

```
example     0>CLRMEM1
```

Now RESET the module again. Once the sign-on message appears, an attempt to list the program in DATA memory proves that the DATA memory has in fact been cleared. However, the program was copied to the PROGRAM FILE memory so the SELECT command can be used to select the program in the PROGRAM FILE memory:

```
example     0>LIST

            Ready
            0>SELECT 1

            Ready
            1>LIST
            10  PRINT "This is my first ASCII BASIC program!"

            Ready
            1>
```

Notice that the ASCII BASIC prompt has changed from **0>** to **1>.** This is because program number 1 is now selected. The prompt signifies the currently selected program. (The selected program means the program that is used when the LIST or RUN commands are entered).

Once a program has been placed into the PROGRAM FILE, it can be executed using three methods:

1. The user can "select" the program in the PROGRAM file and then issue the RUN command. The first program in the PROGRAM FILE is still currently selected.

```
example     1>RUN

            This is my first ASCII BASIC program!
```

2. The user can issue the CHAIN command to select a program from the PROGRAM FILE to be executed. In this case, it does not matter which program is currently selected.

```
example     1>SELECT 0

            Ready
            0>CHAIN 1

            This is my first ASCII BASIC program!
```

3. The user can configure the module to AUTOMATICALLY execute any program following a RESET. This is done by executing the STARTUP 2 command to place the module in STARTUP mode 2 AND by using the AUTORUN command to select which program is to be executed following the RESET.

```
example     1>STARTUP 2
            1>AUTORUN 1

            Ready
            1>RESET

            This is my first ASCII BASIC program!
```

9.5 Deleting a Program from the PROGRAM FILE

The DELPGM command is provided to allow programs to be REMOVED from the PROGRAM FILE. Enter another program and save it in the PROGRAM FILE. Note that ONLY PROGRAM 0 MAY BE EDITED. This means that the SELECT 0 command must be entered before the new program can be entered.

```
example     1>SELECT 0

            0>10 P. "This is my second ASCII BASIC program!"
            0>SAVE
             2

            Ready
            0>DELPGM 1
```

By deleting program number 1, the new program (number 2) was shifted into program number 1. Since the STARTUP and AUTORUN modes are still in affect for program 1 to be executed following a RESET, the new program is now be executed. Reset the ASCII BASIC module to verify this:

```
example     This is my second ASCII BASIC program!
```

APPENDIX A: SERIAL PORT WIRING

1 Primary Port Wiring

The ASCII BASIC Module features two serial ports. The PRIMARY or programming port, follows the RS-232 standard. It can be connected to RS-232 devices in a point-to-point fashion over a distance of 50 feet.

The SECONDARY port is RS-422, RS-232 and optionally Modem compatible. When connected in RS-422 mode, the secondary port can be connected to one or more devices over a total network distance of 4000 feet.

The PRIMARY port is typically connected to a dumb terminal or more commonly an IBM compatible personal computer. In this manual, all cable diagrams feature pin-outs labeled according to function, and not to the EIA standard. In the table below, the ASCII BASIC Module's RS-232 pin-out is listed, with the designation used in the wiring diagrams of this manual. Also listed is the direction of the signal.

Table 1 – Primary Port

Pin #	Signal Name	Direction
1	(DCD) Always High	Output
2	(TXD) Transmit Data	Output
3	(RXD) Receive Data	Input
4	No Connection	N/A
5	(GND) Signal Ground	N/A
6	(DSR) Always High	Output
7	(CTS) Clear To Send	Input
8	(RTS) Request To Send	Output
9	(RI) Always High	Output

Primary Port Cable Pin-outs. The pin-outs on the next page show connections of common devices to the ASCII BASIC Module's primary RS-232 port. These pin-outs are typical and do not represent all possible connections.

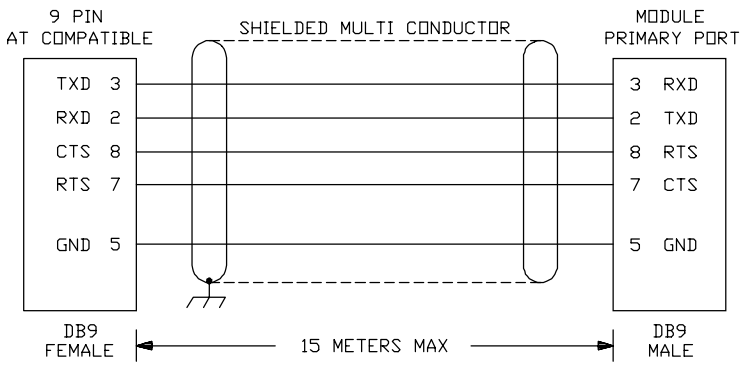

Figure 1 – PC with 9-pin RS232 Port

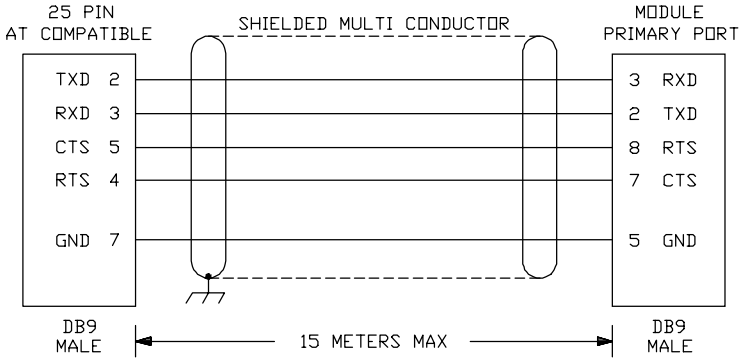

Figure 2 – PC with 25-pin RS232 Port

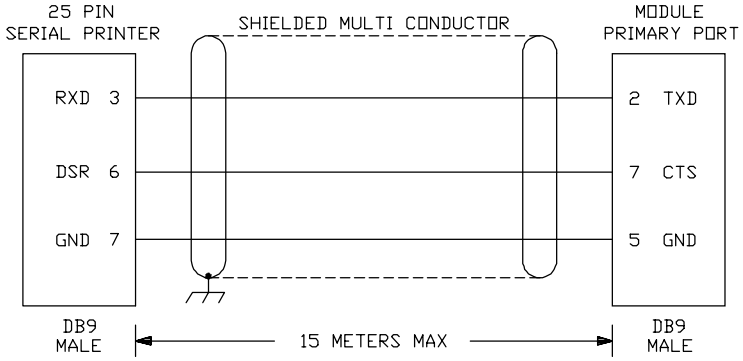

Figure 3 – Typical Serial Printer

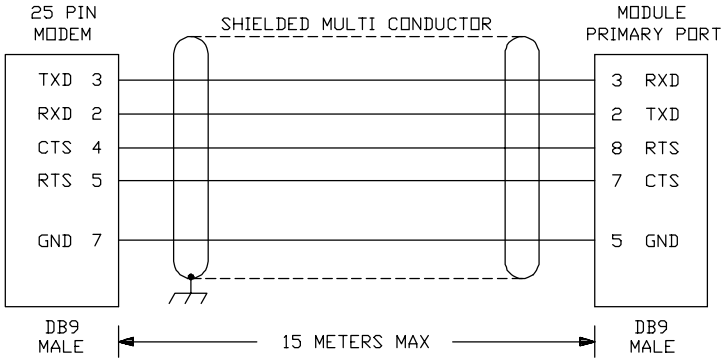

Figure 4 – Typical Modem

2 Auxiliary Port Wiring

2.1.1 Pin-out

The Auxiliary serial port has connections for both RS-232 and RS-485. The RS-232 connection can be made point-to-point over a distance of 50 feet. The RS-485 connection can be made point-to-point or multidropped over a total network distance of 1.5 km.

The Auxiliary port is typically connected to serial printers, modems or the programming port of the PLC. In this manual, all cable diagrams feature pin-outs labeled according to function and not to the EIA standard. In the table below, the ASCII BASIC Module's RS-232/485 pin-out is listed with the designation used in the wiring diagrams of this manual. Also listed is the direction of the signal.

Table 2 – Port 0/1 Pin-out

		ASC100 Port 0/1
Output	1	**DCD** (Data Carrier Detect)
Output	2	**RXD** (Receive Data)
Input	3	**TXD** (Transmit Data)
Input	4	**DTR** (Data Terminal Ready)
GND	5	**Signal Ground**
Output	6	**DSR** (Data Set Ready)
Input	7	**CTS** (Clear to Send)
Output	8	**RTS** (Request to Send)
Output	9	**RI** (Ring Indicate)

Note: For ports 0 and 1, the signal names reflect the EIA RS232 signal names for a DCE device.

The names do not necessarily reflect the signal direction with respect to the ASC100 module.

Table 3 – Port 2 Pin-out

		ASC100 Port 2
Input	1	**RXD-** (Receive Data -)
Output	2	**TXD-** (Transmit Data -)
Output	3	**CTS-** (Clear to Send -)
Input	4	**RTS-** (Request to Send -)
GND	5	**GND** (Signal Ground)
Input	6	**RXD+** (Receive Data +)
Output	7	**TXD+** (Transmit Data +)
Output	8	**CTS+** (Clear to Send +)
Input	9	**RTS+** (Request to Send +)

NOTES

APPENDIX B: RESERVED WORD LIST

The following is an alphabetic list of all of the ASCII BASIC reserved and key words. These words can NOT be used as BASIC variable names.

KEYWORD	DESCRIPTION
.AND.	Logical or bit-wise AND
.OR.	Logical or bit-wise OR
.XOR.	Logical or bit-wise XOR
ABS	Returns absolute value
ASC(	Returns ASCII character code
ATN	Returns ARCTANGENT
AUTORUN	Configures program to run after RESET
BCD	Binary to BCD conversion
BNR	BCD to Binary conversion
BREAK	Enable/disable ctrl-c or sets a breakpoint
CHAIN	Runs PROGRAM FILE memory program
CHR(	Returns ASCII character
CHR$(	Returns ASCII character
CLEAR	Clears all BASIC variables
CLOCK	Starts/stops millisecond clock
CLRMEM	Enables/disables RESET memory init.
CMDPORT	Assigns console to specified serial port
COMBRK	Detects or transmits a long break
CONT	Continue program execution after STOP
COS	Returns COSINE4-21
CR	Prints a carriage return, no line feed4-21
CTS	Sets or returns the state of the CTS signal
DATA	List of constant data
DELAY	Causes the program to pause
DELPGM	Erases a program from PROGRAM FILE
DIAG	Firmware diagnostics
DIM	Defines max subscript for array variables
DO	Iterative loop control
EDIT	Moves PROGRAM FILE to DATA memory
ELSE	Conditional statement (see IF)
END	Terminates program
ERC	Returns arithmetic error code
EXP	Returns “e” (2.7182818) to the “x”
FOR	Iterative loop control
FREE	Returns amount of available memory
FTIME	Assigns/returns frac. portion of TIME
GOSUB	Executes a subroutine
GOTO	Jumps to specified line
HELP	Displays ON-LINE help information
IDLE	Waits for a BASIC interrupt
IF	Conditional statement
INBUF$	Returns all characters in the INPUT buffer
INKEY$	Returns next character to INPUT buffer
INP(	Returns WL register
INPUT	Reads serial console input
INSTR(	Returns position of string2 in string1

KEYWORD	DESCRIPTION
INT	Returns integer portion of argument
LCASE$(	Returns string argument in lower case
LD@	Stores a floating-point value
LEFT$(	Returns leftmost characters of string
LEN(	Returns length of the current program
LET	Assigns a value to a variable
LIST	Outputs program listing
LOG	Returns natural logarithm
MID$(	Returns a portion of a string
MTOP	Assigns/returns protected memory
NEW	Erases DATA memory program
NEXT	Iterative loop control (see FOR)
NOT	Returns ONES complement
NULL	Sets NULL count following CR
ON	Case sensitive program vector control
ONERR	Error trapping control
ONPORT	Serial interrupt control
ONTIME	Timer interrupt control
OUT(	Assigns WL output register
PH0.	Print values in HEX format (2 digit)
PH1.	Print values in HEX format (4 digit)
PI	Returns value of PI (3.1415926)
POP	Gets floating-point value from stack
PRINT	Serial output
P. or ?	Same as PRINT
PUSH	Puts floating-point value to stack
READ	Assigns constant from DATA list to var
REM	Comment
RESTORE	Initializes DATA pointer
RESET	Causes a software RESET of the module
RETI	Returns from timer interrupt routine
RETURN	Returns from subroutine
RIGHT$(	Returns rightmost characters of a string
RND	Returns a random number
RTRAP	Enables/disables run trap option
RTS	Returns the state of the RTS signal
RUN	Runs selected program
SAVE	Stores a program in the PROGRAM file
SELECT	Selects a program
SETCOM	Configures one of the serial ports
SETINPUT	Configures the INPUT statement
SGN	Returns the sign of the value
SIN	Returns the SINE of the value
SIZE	Returns the SIZE of the current program
SPC	Outputs specified number of spaces
SQR	Returns square-root
STARTUP	Configures the modules behaviour after
STATUS	Displays memory and configuration data
ST@	Stores floating point value
STEP	Single step or Iterative loop control
STOP	Halts program execution
STRING	Allocates memory for STRING storage
STR$	Returns the string equivalent of an expr
TAB	Outputs spaces until at specified pos
TAN	Returns TANGENT

KEYWORD	DESCRIPTION
THEN	Conditional statement (see IF)
TIME	Assigns/returns millisecond clock
TO	Iterative loop control (see FOR)
UCASE$(	Returns string as upper case
UNTIL	Iterative loop control (see DO)
USING(	Defines numeric output format
U.(	Same as USING(
VAL(	Returns numeric equivalent of string
WHILE	Iterative loop control (see DO)
XBY	Assigns/returns data at given address

NOTES

APPENDIX C: ASCII CHARACTER SET

DEC	HEX	Character	DEC	HEX	Character	DEC	HEX	Character
00	00H	NULL	44	2CH	,	88	58H	X
01	01H	SOH	45	2DH	-	89	59H	Y
02	02H	STX	46	2EH	.	90	5AH	Z
03	03H	ETX	47	2FH	/	91	5BH	[
04	04H	EOT	48	30H	0	92	5CH	\
05	05H	ENQ	49	31H	1	93	5DH	]
06	06H	ACK	50	32H	2	94	5EH	^
07	07H	BELL	51	33H	3	95	5FH	_
08	08H	BS	52	34H	4	96	60H	‘
09	09H	HT	53	35H	5	97	61H	a
10	0AH	LF	54	36H	6	98	62H	b
11	0BH	VT	55	37H	7	99	63H	c
12	0CH	FF	56	38H	8	100	64H	d
13	0DH	CR	57	39H	9	101	65H	e
14	0EH	SO	58	3AH	:	102	66H	f
15	0FH	SI	59	3BH	;	103	67H	g
16	10H	DLE	60	3CH	<	104	68H	h
17	11H	DC1	61	3DH	=	105	69H	i
18	12H	DC2	62	3EH	>	106	6AH	j
19	13H	DC3	63	3FH	?	107	6BH	k
20	14H	DC4	64	40H	@	108	6CH	l
21	15H	NAK	65	41H	A	109	6DH	m
22	16H	SYN	66	42H	B	110	6EH	n
23	17H	ETB	67	43H	C	111	6FH	o
24	18H	CAN	68	44H	D	112	70H	p
25	19H	EM	69	45H	E	113	71H	q
26	1AH	SUE	70	46H	F	114	72H	r
27	1BH	ES1	71	47H	G	115	73H	s
28	1CH	FS2	72	48H	H	116	74H	t
29	1DH	GS	73	49H	I	117	75H	u
30	1EH	RS	74	4AH	J	118	76H	v
31	1FH	US	75	4BH	K	119	77H	w
32	20H	SPACE	76	4CH	L	120	78H	x
33	21H	!	77	4DH	M	121	79H	y
34	22H	“	78	4EH	N	122	7AH	z
35	23H	#	79	4FH	O	123	7BH	{
36	24H	$	80	50H	P	124	7CH	|
37	25H	%	81	51H	Q	125	7DH	}
38	26H	&	82	52H	R	126	7EH	~
39	27H	‘	83	53H	S	127	7FH	DEL
40	28H	(	84	54H	T			
41	29H	)	85	55H	U			
42	2AH	*	86	56H	V			
43	2BH	+	87	57H	W			

NOTES

APPENDIX D: MEMORY CONFIGURATION

1 ASCII BASIC Memory Map

The ASCII BASIC Module is configured with 64K EPROM firmware memory (Operating system), 32K (battery-backed if installed) RAM data memory (Variable storage) and 32K EEPROM program memory (BASIC program storage).

STOP! Two areas of memory are reserved for the ASCII BASIC module firmware and must NEVER be manipulated by the XBY() or ST@ commands. These areas are located from address 0 through address 1535 (05FFH), and from address 61440 (F000H) through 65535 (FFFFH).

DATA memory provides storage memory for BASIC program number 0 (the only program that can be edited) as well as all variable and string storage space. The following table illustrates the DATA memory map:

FROM		TO	
Decimal	Hexadecimal	Decimal	Hexadecimal
1536	600H	32767	7FFFH

PROGRAM FILE memory is the memory used to store BASIC programs using the SAVE command. These programs can be deleted using the DELPGM command, but cannot be edited. The following table illustrates the PROGRAM FILE memory map:

FROM		TO	
Decimal	Hexadecimal	Decimal	Hexadecimal
32768	8000H	61439	EFFFH

STOP! The first 16 bytes of the PROGRAM FILE memory are used to store important configuration information such as the primary port baud rate, the STARTUP mode and the AUTORUN program number. These bytes must NEVER be manipulated by the XBY() or ST@ commands.

NOTES

APPENDIX E: TERMINAL EMULATION SOFTWARE USER MANUAL

TERM - Dumb Terminal Emulation Program
Version 2.26

SOFTWARE LICENSE AGREEMENT

1 INTRODUCTION

1.1 What is TERM?

TERM is an executable program that can be run on any IBM Personal Computer (PC), PC/XT, PC/AT or close compatible. Essentially, TERM converts the host computer into a dumb terminal (a keyboard and a display screen). Utilizing one of the host computer’s RS232 COM ports, TERM displays characters received at the COM port and transmits characters that are typed on the keyboard to the COM port. Although there are some enhancements (discussed in detail later), that is TERM’s primary function.

1.2 Equipment Requirements

As stated above, TERM runs on any IBM PC, PC/XT, PC/AT or close compatible running DOS 5.0 or later with at least one COM port. At least one floppy drive is required. TERM requires approximately 100K bytes of available RAM memory to run. Color displays are supported but not required.

2: INVOCATION – RUNNING TERM

2.1 General

TERM was written with ease of use in mind at all times. After TERM has been invoked, there are help or status messages on the display to inform the user of his/her options or to show what operation is currently taking place.

2.2 Installing TERM

Before running TERM, make a working copy of the distribution diskette and put the distribution diskette in a safe place in case the working copy fails. To make a backup copy of the disk:

1. Type "DISKCOPY A: A:" (without the quotes).
2. When prompted to insert the source diskette, place the TERM distribution diskette into the floppy drive and press the “ENTER” key.
3. When prompted to insert the destination diskette, place a formatted diskette into the floppy drive and press the "ENTER" key.

To install to a hard disk, insert the distribution diskette into the floppy drive, log to a directory on the hard disk, and type:

"COPY A:\TERM\TERM.EXE /V" (without the quotes).

TERM is now installed and ready for use.

2.3 Running TERM for the First Time

To run TERM, change to the drive/directory that contains TERM.EXE, and type:

TERM <Enter>

Once the program is loaded into memory, TERM displays a sign-on message, which remains on the screen for five seconds or until a key on keyboard is pressed.

Initially, TERM searches the current drive/directory for a file called TERM.CFG. If this file is found, the terminal screen is displayed following the sign-on message. If the TERM.CFG file is not present, the configuration menu is displayed. See Section 3 for more information about the configuration menu. The TERM.CFG file is updated every time the configuration is changed. Therefore, once TERM is run the first time, the terminal screen appears after the sign-on message, and the configuration is set as it was during the last session with TERM.

2.4 Screen Colors

If using a color display, the colors generated by TERM are those used by DOS when the TERM program is invoked. Use the DOS PROMPT command (or several third party packages) to alter the screen colors used by DOS prior to running TERM if a color display is desired.

2.5 Exiting TERM

At any time during terminal mode, the <F10> key can be pressed to cause TERM to terminate and return control to DOS.

3: <F1> - CONFIGURING TERM

3.1 The TERM.CFG Configuration File

TERM is distributed as a single file called TERM.EXE. Following the initial invocation of TERM, an additional file called TERM.CFG is located on the directory that was logged when TERM was invoked. This file contains the information regarding the COM port configuration as it was set during the previous session with TERM. The following information is stored in the TERM.CFG file (the information in parentheses denotes the value displayed on the configuration menu if no TERM.CFG file is present):

COM port	(1)
Baud rate	(9600)
Parity type	(N)
Number of data bits	(8)
Number of stop bits	(1)
Handshake type	(XON / XOFF)
Display mode	(ASCII)

Each of these parameters is discussed in detail later in this section.

The TERM.CFG file is placed on the "current" or "logged" directory. If TERM is run from a different directory where no TERM.CFG file exists, a new one is created. This is done purposely, our philosophy is that if one wants to run TERM from a different directory, chances are good that it is being used for a different project and probably requires a different configuration. This method allows TERM to be used for several different projects with several different configurations without having to configure the port every time TERM is run.

3.2 What Happens when F1 is Pressed

If, during the display of the configuration menu, the ENTER key is pressed, the parameters displayed are stored in the TERM.CFG file and are used to configure the specified COM port.

If, during the display of the configuration menu, the ESCape key is pressed, the parameters displayed are used to configure the specified COM port, but the TERM.CFG file is left intact.

The configuration menu allows alteration of seven parameters. Each of these parameters is associated with a numeric key on the keyboard. Each of these parameters is discussed in Sections 3.3-3.9.

3.3 COM Port Selection

Item number one (1) on the configuration menu pertains to the COM port selection. During the display of the configuration screen, the (1) key can be pressed to select COM1:, COM2:, COM3: or COM4: as the device used by TERM in terminal mode. Only COM ports that are present in the host computer are displayed for selection.

3.4 Baud Rate Selection

Item number two (2) on the configuration menu pertains to the Baud rate selection or data transmission rate used by TERM in terminal mode. During the display of the configuration menu, the (2) key can be pressed to select any of the following values: 110, 300, 600, 1200, 1800, 2400, 3600, 4800, 9600, 19200, 38400 or 57600 baud. If higher baud rates are to be selected (19200 and up), the cabling distance should be kept to a minimum.

3.5 Parity Type Selection

Item number three (3) on the configuration menu pertains to the type of parity used by TERM in terminal mode. During the display of the configuration menu, the (3) key can be pressed to select any of the following values: None, Even, Odd, Marking or Spacing parity. Parity is a sort of "built-in" mode of error checking for RS232 communications. To simplify, set the parity to that of the device to which you are communicating.

3.6 Data Bit Selection

Item number four (4) on the configuration menu pertains to the number of RS232 data bits that are transmitted and received by TERM in terminal mode. During the display of the configuration menu, the (4) key can be pressed to select 5, 6, 7 or 8 data bits. Again, set this number to that of the device to which you are communicating.

3.7 Stop Bit Selection

Item number five (5) on the configuration menu pertains to the number of RS232 stop bits that are transmitted and received by TERM in terminal mode. During the display of the configuration menu, the (5) key can be pressed to select 1 or 2 stop bits. Again, set this number to that of the device to which you are communicating.

3.8 Handshake Type Selection

Item number six (6) on the configuration menu pertains to the type of RS232 handshaking that is to be performed by TERM in terminal mode. During the display of the configuration menu, the (6) key can be pressed to select NONE, XON/XOFF or RTS/CTS handshaking.

3.9 Display Type Selection

Item number seven (7) on the configuration menu pertains to the display mode of received characters by TERM in terminal mode. During the display of the configuration menu, the (7) key can be pressed to select ANSI, ASCII or HEX, display mode. When the ASCII mode is selected, data received is displayed as ASCII characters. When the HEX mode is selected, the ASCII values of the characters received are displayed as hexadecimal numbers. The HEX mode is useful for debugging communication intensive operations. The ANSI display mode is identical to the ASCII mode except that some of the ANSI-standard ESCape sequences are supported. See Section 7 for a complete discussion of the ANSI display mode.

4: THE TERMINAL SCREEN

4.1 General

Once TERM has been configured, the terminal screen is displayed. A flashing cursor is displayed in the upper left corner of the display. This cursor represents the location of the next character received.

4.2 Transmitting and Receiving Data

As data characters are received at the selected COM port, they are displayed on the screen. Each time a key on the keyboard is pressed, it is transmitted to the selected COM port. TRANSMITTED CHARACTERS ARE NOT ECHOED ON THE DISPLAY UNLESS THE RECEIVING DEVICE SENDS THEM.

Carriage return characters (ASCII 13) and line feed characters (ASCII 10) are displayed as intended. A carriage return causes the cursor to revert to the first column of the current line. Line feeds advance the cursor down one line but maintain the current column.

All other ASCII characters are displayed as the IBM standard character set.

As data is received, the cursor advances. If the cursor reaches the 24th line of the display, and a line feed character is received (or data is received beyond the 79th column), the display scrolls up one line, and the data received is displayed on the 24th line.

4.3 Error Messages

No error message is built-in to the TERM program. Communication errors are ignored. Disk errors make TERM crash.

5: <F2> - FILE DOWNLOAD

5.1 General

TERM has the ability to transmit a disk file to the selected COM port. This is initiated by pressing the <F2> key during the display of the terminal screen.

5.2 Selecting a File to Download

When the <F2> key is pressed, a prompt box appears in the center of the terminal screen and the user is prompted to enter the filename of the file to be transmitted. The user can enter up to 50 characters of path/filename. Once the filename is typed, the user must press the **ENTER** key. TERM then searches for the file in the specified directory (or in the current directory if no pathname is specified). If the file is not found, an error message is displayed and the user is returned to the terminal screen. If the file exists, it is immediately transmitted to the COM device. When the entire file has been downloaded, the user is returned to the terminal screen. The user can abort the entry of a filename or the download at any time by pressing the <ESCape> key.

Below is a list of steps to be taken if difficulties arise while downloading to the module.

Step 1 Turn on software handshaking in both the module and TERM (XON/XOFF).

Step 2 Slow down the baud rate in both the module and TERM (9600 or 4800).

Step 3 Reset the module by powering down and back up.

Step 4 Verify the integrity of the cable.

Step 5 Ensure that the program being downloaded has no tab characters, correct line numbers and minimal blank lines.

6: <F3> - FILE UPLOAD

6.1 General

TERM has the ability to store received data into a disk file. This is initiated by pressing the <F3> key during the display of the terminal screen.

6.2 Selecting a Filename

When the <F3> key is pressed, a prompt box appears in the center of the terminal screen and the user is prompted to enter the filename of the file to which the received data is to be written. The user can enter up to 50 characters of path/filename. Once the filename is typed, the user must press the "ENTER" key. TERM then searches for the file in the specified directory (or in the current directory if no pathname is specified). If the file exists, the user is asked if the existing file is to be deleted. If the user enters "N" (for NO), control returns to the terminal screen. If the user enters "Y" (for YES) or if the file doesn't exist, TERM begins the upload operation.

6.3 What Happens during the Upload

Once the file has been opened, TERM begins writing all received characters to the specified disk file. To upload a file from the ASCII BASIC Module, begin the upload process. Once the file has been opened use the command LIST to generate a listing of the program. This listing is captured by TERM and stored to disk when the ESCape key is pressed. The disk file is closed (the UPLOAD operation is terminated) when the user presses the ESCape key.

This process is also useful in capturing a printout of the characters printed to the display while the BASIC program is running. Simply begin the upload process before running the BASIC program. Term captures all characters until the ESCape key is pressed.

7: ANSI COMPATIBILITY

7.1 General

The TERM program supports the following ANSI escape sequences when configured in ANSI display mode:

In the descriptions below, <ESC> appears whenever the ESCape character is referenced. All of the ANSI escape sequences begin with this character (ASCII 1BH, 27 decimal). (Numeric parameters appear in *italic* print) character.

The ANSI escape sequences supported by TERM are documented below:

Set absolute cursor position: <ESC>[*r*:*c*H

r = row# (1 to 24), *c* = col# (1 to 80). If unspecified, *r* and *c* default to 1. If to large, *r* and *c* default to max.

Move cursor up: <ESC>[*r*A

r = number of rows. If unspecified, *r* defaults to 1. If (current position)-*r* < 1, cursor is moved to line 1.

Move cursor down: <ESC>[*r*B

r = number of rows. If unspecified, *r* defaults to 1. If (current position)+r > 24, cursor is moved to line 24.

Move cursor right: <ESC>[*c*C

c = number of columns. If unspecified, *c* defaults to 1. If (current position)-*c* < 1, cursor is moved to column 1.

Move cursor left: <ESC>[*c*D

c = number of columns. If unspecified, *c* defaults to 1. If (current position)+c > 80, cursor is moved to column 80.

Save cursor position: **<ESC>[s**

Restore cursor position: **<ESC>[u**

Erase display: **<ESC>[2J**

Set graphics rendition: **<ESC>[*p*m**
p = 7 for reversed video, *p*=0 for normal video.

Invisible cursor: **<ESC>[i**

Visible cursor: **<ESC>[v**

NOTES