SIEMENS

SIMATIC TI505

BASIC Module

User Manual

Order Number: PPX:505-8101-2
Manual Assembly Number: 2586546-0031
Second Edition

01/21/92

MANUAL PUBLICATION HISTORY

SIMATIC TI505 BASIC Module User Manual
Order Manual Number: PPX:505–8101-2

Refer to this history in all correspondence and/or discussion about this manual.

Event	Date	Description
Original Issue	12/89	Original Issue (2592497–0001)
Second Edition	01/93	Second Edition (2592497–0002)

LIST OF EFFECTIVE PAGES

Pages	Description	Pages	Description
Cover/Copyright	Second		
History/Effective Pages	Second		
iii — viii	Second		
1-1 — 1-4	Second		
2-1 — 2-24	Second		
3-1 — 3-77	Second		
A-1 — A-6	Second		
B-1 —B-6	Second		
C-1 — C-2	Second		
D-1 — D-6	Second		
Registration	Second		

Contents

Chapter 1 Introduction

1.1 Introduction 1-2

1.2 Module Indicators 1-3

Chapter 2 Installation and Initialization

2.1 Introduction 2-2

2.2 Setting the Configuration Switches 2-3

2.2.1 Configuring 6-switch Sets 2-5
2.2.2 Configuring the 4-switch Set 2-5
2.2.3 Installing the Module 2-7

2.3 Connecting Cables and Wiring 2-9

2.3.1 Communications Ports 2-9
2.3.2 RS-232-C/432 Communications 2-10

2.4 Connecting a VPU 2-11

2.5 Connecting Other Devices 2-15

2.6 Starting-up the Programmable Basic Module 2-16

2.6.1 User-Initiated Test 2-16
2.6.2 Verifying PLC-module Communication 2-16

2.7 Starting-up the System 2-18

2.7.1 Starting-up the VPU 2-18
2.7.2 Checking Communications 2-20
2.7.3 Caps Lock On 2-20
2.7.4 Enabling VPU Auxiliary (Printer) Port 2-21
2.7.5 Error Diagnostics 2-21
2.7.6 Starting-up Other Programming Devices 2-22

2.8 Hardware Troubleshooting 2-23

2.8.1 Programming Device and/or Module Failure 2-23
2.8.2 Battery Failure 2-23
2.8.3 Changing the Battery 2-24

Chapter 3 Programming Language

3.1 Introduction **3-3**

3.2 Operation Modes **3-4**

3.2.1 RUN Mode 3-4
3.2.2 PROGRAM Mode 3-4
3.2.3 Power-up Modes 3-4
3.2.4 Source Statement 3-6
3.2.5 Constants 3-7
3.2.6 Variables 3-8

3.3 BASIC Commands **3-11**

3.3.1 CONTINUE Command 3-11
3.3.2 LIST Command 3-12
3.3.3 LOAD Command 3-12
3.3.4 NEW Command and Retentive Memory 3-13
3.3.5 RUN Command 3-14
3.3.6 SAVE Command 3-14
3.3.7 SIZE Command 3-15

3.4 BASIC Statements **3-16**

3.4.1 DEF Statement 3-16
3.4.2 DIM Statement 3-17
3.4.3 ERROR Statement 3-18
3.4.4 ESCAPE/NOESC Statement 3-20
3.4.5 FIND Statement 3-21
3.4.6 FOR/NEXT Loop Statement 3-22
3.4.7 INPUT Statement 3-25
3.4.8 LET Statement 3-27
3.4.9 PRINT Statement 3-28
3.4.10 RANDOM Statement 3-35
3.4.11 REM Statement 3-36
3.4.12 STOP/END Statements 3-36
3.4.13 TAB Statement 3-37
3.4.14 TIME Statement 3-38
3.4.15 UNIT Statement 3-39
3.4.16 Internal Data Statements 3-40
3.4.17 Branch Statements 3-41
3.4.18 CALL (PLC Interface Statement) 3-44
3.4.19 Subroutine Statements 3-52

3.5 BASIC Functions 3-54
3.5.1 ABS Function 3-54
3.5.2 ASC Function 3-54
3.5.3 ATN Function 3-54
3.5.4 BIT Function 3-55
3.5.5 COS Function 3-58
3.5.6 EXP Function 3-58
3.5.7 INP Function 3-59
3.5.8 LEN Function 3-59
3.5.9 LOG Function 3-60
3.5.10 MCH Function 3-60
3.5.11 MEM Function 3-61
3.5.12 MWD Function 3-61
3.5.13 NKY Function 3-62
3.5.14 RND Function 3-63
3.5.15 SIN Function 3-63
3.5.16 SRH Function 3-64
3.5.17 SQR Function 3-64
3.5.18 SYS Function 3-64
3.5.19 TIC Function 3-66
3.5.20 String Manipulations 3-66

3.6 BASIC Operations 3-72
3.6.1 Arithmetic Operators 3-72
3.6.2 Logical Operators 3-72
3.6.3 Boolean Operators 3-73
3.6.4 Relational Operators 3-73
3.6.5 Order of Operator Evaluation 3-74

3.7 Editing a Program 3-75
3.7.1 General Editing Procedures 3-75
3.7.2 Editing with a VPU 3-77

Appendix A Summary of BASIC Language

A.1 Introduction A-2

A.2 Table of BASIC Functions A-3

A.3 Table of BASIC Statements A-4

Appendix B VPU Hex-ASCII Codes

B-1 VPU Character Codes B-2

Appendix C Reserved Words

C.1 Reserved Words List C-2

Appendix D Error Format

D.1 Introduction D-2

D.2 Error Messages D-3

List of Figures

1-1 Programmable BASIC Module Indicators 1-3
2-1 Switch Configuration Sets 2-3
2-2 Inserting the Programmable BASIC Module 2-8
2-3 Ports Available on the Module 2-9
2-4 VPU Ports 2-12
2-5 Pin Diagram 2-13
2-6 VPU Keycap Placement 2-14

List of Tables

1-1 Programmable BASIC Module Specifications 1-4

2-1 Configuration Switch Settings 2-5
2-2 Configuring the 4-switch Set 2-7
2-3 Pin Definition 2-13

3-1 Formatting String Characters 3-35
3-2 VPU Editing Symbols 3-76
3-3 VPU Editing Keys 3-77

A-1 BASIC Commands A-2
A-2 BASIC Functions A-3
A-3 BASIC Statements A-4

B-1 VPU Character Codes B-2

D-1 Error Messages D-3

Chapter 1

Introduction

1.1 Introduction **1-2**

1.2 Module Indicators **1-3**

1.1 Introduction

This manual is designed to help you install the Programmable BASIC Module, prepare it for operation, and learn the BASIC language needed for programming. Chapter 2 shows how to install and prepare the module for operation. Chapter 3 describes the commands, statements, and functions of the BASIC language.

The Programmable BASIC Module is an intelligent I/O module with 28k-bytes of memory. This module is compatible with all Series 500™/505™ Programmable Logic Controllers (PLCs), but operates asynchronously for minimal effect on PLC scan time. The Programmable BASIC Module may be programmed with a VPU, IBM® AT™ computers and compatibles running terminal emulation software, or other compatible non-intelligent terminals. It is used in applications where:

- Complex mathematical evaluations are needed (as in PID, statistical analysis, and gas flow calculations).
- Machine diagnostics are required.
- Parallel processing is needed to shorten the PLC scan time.
- Operation with specialized devices such as bar code readers and CRTs is desired.
- Applications using serial (RS-232-C) communications are required.

The module also features retentive memory and dual serial ports for communication with operators and machines. The module increases the control and memory capabilities of the PLC. The 3-volt lithium battery in the module backs-up the retentive memory areas and the internal clock.

1.2 Module Indicators

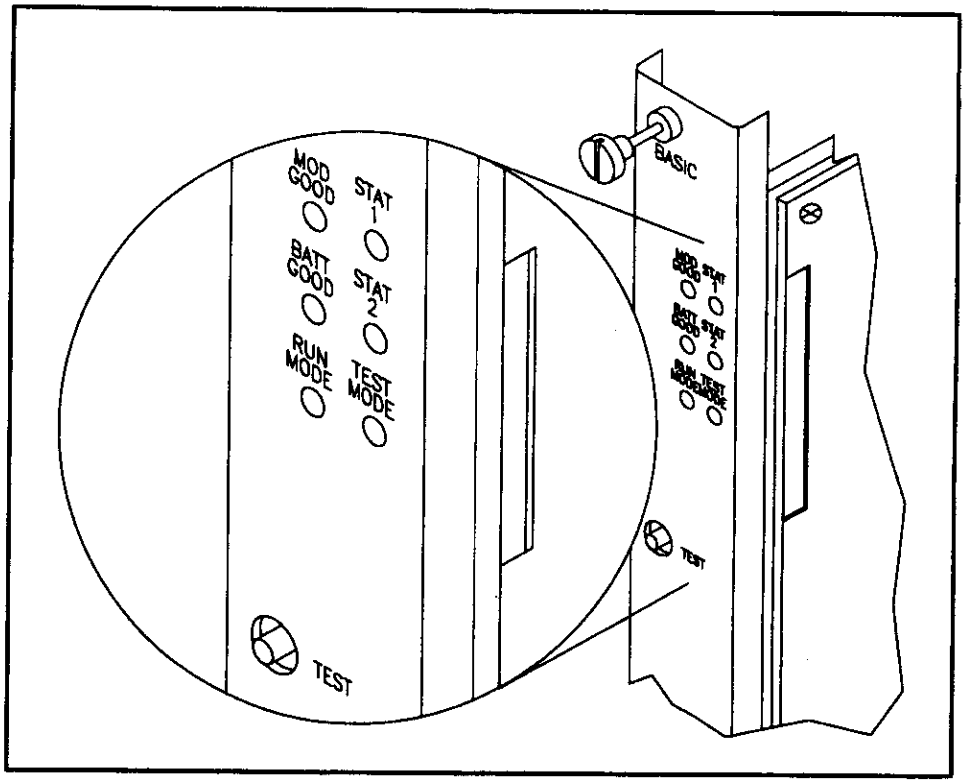


Figure 1-1 Programmable BASIC Module Indicators

As shown in Figure 1-1, there are six indicators on the face of the module:

- MOD GOOD (module good)
- BATT GOOD (battery good)
- RUN MODE (program is being executed)
- TEST MODE (diagnostic check)
- STAT 1 (status)
- STAT 2 (status)

Module Indicators (continued)

The MOD GOOD indicator is on when no fatal errors are detected during diagnostics. The BATT GOOD indicator is on when the battery is in place and can back up the memory and internal clock. When a program is being executed by the Programmable BASIC Module, the RUN MODE indicator is on. The TEST MODE indicator is on while the module performs diagnostics. STAT 1 and STAT 2 are for future enhancements.

Table 1-1 Programmable BASIC Module Specifications

Operating Temperature:	0° to 60°C
Storage Temperature:	−40° to 85°C
Operating Humidity	0 to 95% noncondensing
Vibration:	NAVMAT P9492
Module width:	Single wide I/O module
Power Consumption (From Base)	6 watts
Battery Type:	Lithium coin type, 3.0 VDC, BR2325
Battery Shelf Life:	5 years
Enabled Life:	6 months
User Memory:	28k bytes
Number of Ports:	2 RS-232-C/423
Memory Buffers:	Input: 128 ASCII characters Output: 128 ASCII characters (Port 1) 1024 ASCII characters (Port 2)
Baud Rate:	110 to 19,200 bps
Comm. Distance:	220 m (721 ft.) maximum
Parity:	Even, odd, or no parity
Bit Size:	7 or 8 bit characters
Start Bits:	1
Stop Bits:	2, 110 baud, 1 at other rates
Operation mode:	DTE
Agency Approvals:	UL® Listed CSA Certified FM Approved Class 1, Div. 2
Technical Assistance:	Siemens Industrial Automation, Inc. distributor

Chapter 2

Installation and Initialization

2.1 Introduction **2-2**

2.2 Setting the Configuration Switches **2-3**

2.2.1 Configuring 6-switch Sets 2-5
2.2.2 Configuring the 4-switch Set 2-5
2.2.3 Installing the Module 2-7

2.3 Connecting Cables and Wiring **2-9**

2.3.1 Communications Ports 2-9
2.3.2 RS-232-C/432 Communications 2-10

2.4 Connecting a VPU **2-11**

2.5 Connecting Other Devices **2-15**

2.6 Starting-up the Programmable Basic Module **2-16**

2.6.1 User-Initiated Test 2-16
2.6.2 Verifying PC-module Communication 2-16

2.7 Starting-up the System **2-18**

2.7.1 Starting-up the VPU 2-18
2.7.2 Checking Communications 2-20
2.7.3 Caps Lock On 2-20
2.7.4 Enabling VPU Auxiliary (Printer) Port 2-21
2.7.5 Error Diagnostics 2-21
2.7.6 Starting-up Other Programming Devices 2-22

2.8 Hardware Troubleshooting **2-23**

2.8.1 Programming Device and/or Module Failure 2-23
2.8.2 Battery Failure 2-23
2.8.3 Changing the Battery 2-24

2.1 Introduction

This chapter describes how to install the Programmable BASIC Module in a Series 505 I/O base, connect the communication cables to the programming device (and any other peripheral devices), and prepare the system for operation.

2.2 Setting the Configuration Switches

There are three sets of configuration switches as shown in Figure 2-1. The two 6-position switch sets (S1 & S2) are for port 1 and port 2 (as labeled). The 4-position switch (S3) is for the battery and power-up mode. Each of these is discussed in the following sections.

2.2.1 Configuring 6-switch Sets

The first three switches (numbers 1–3) control the port baud rates The baud rates range from 110 to 19.2k bps. The setting for each baud rate is given in Table 2-1.

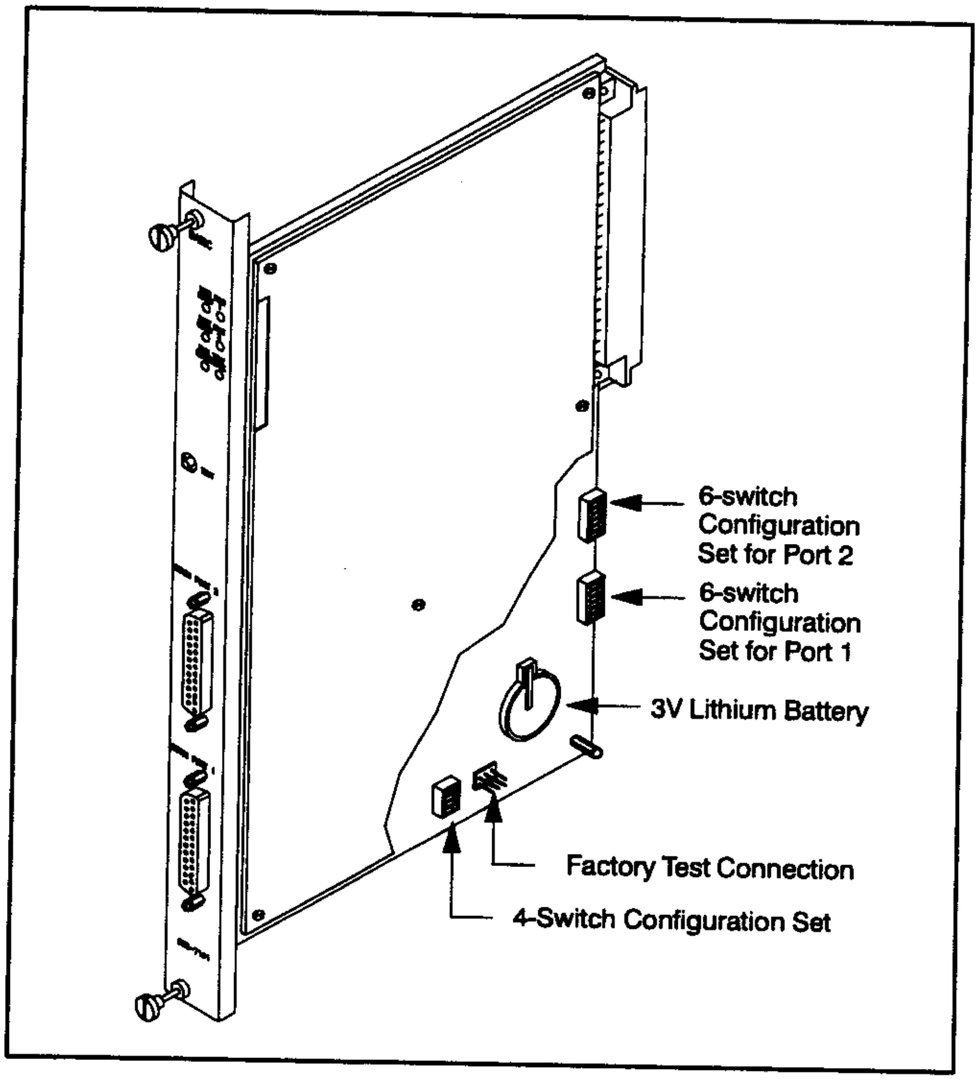

6-switch Configuration Set for Port 2
6-switch Configuration Set for Port 1
3V Lithium Battery
Factory Test Connection
4-Switch Configuration Set

Figure 2-1 Switch Configuration Sets

Switches 4 and 5 control parity operation. Switch 4 either enables parity (if set to open) or disables parity (if set to closed). Switch 5 is used to choose odd or even parity: if the switch is set to open, odd parity is chosen; if it is set to closed, even parity is chosen.

Switch 6 allows either 7- or 8-bit characters. If the switch is set to open, 8-bit characters are selected. If it is set to closed, 7-bit characters are selected.

Table 2-1 summarizes the configuration settings for switches 1 through 6. Select OFF by setting the switch to open; select ON by setting the switch to closed.

Table 2-1 Configuration Switch Settings

Switch	Switch Position							
6	OFF = 8-Bit Character ON = 7-Bit Character							
5	OFF = Odd Parity ON = Even Parity							
4	OFF = Parity Enabled ON = Parity Disabled							
Baud Rate	110	300	600	1200	2400	4800	9600	19.2k
3	ON	ON	ON	ON	OFF	OFF	OFF	OFF
2	ON	ON	OFF	OFF	ON	ON	OFF	OFF
1	ON	OFF	ON	OFF	ON	OFF	ON	OFF

2.2.2 Configuring the 4-switch Set

The 4-switch configuration set is used for the battery and for power-up mode.

Switch 1 determines whether the module will return to PROGRAM or RUN mode after a power interruption. If it is set to open, the module will be in RUN mode when power is restored. Execution will restart at the beginning of the program without prompting from the keyboard. If the switch is set to closed, the module will be in PROGRAM mode when the power is restored, and no action will occur until restart is executed from the keyboard.

⚠ **WARNING**

When the module is set to RUN for power-up mode, field devices under the control of the module application program may begin operating when power is restored. To minimize the risk of personal injury and equipment damage, check the following prior to restoring power to the module: 1) Ensure that all personnel are clear of machinery areas controlled by the module application program; 2) Ensure that all equipment is prepared for start-up.

Switches 2 and 3 are not currently in use.

Switch 4 turns the battery on (closed) or off (open). When on, the battery will be enabled and will be the back-up for the internal time-of-day clock, program memory, and retentive variable memory. Circuit design provides for 30 minutes data retention while changing the battery.

NOTE: A good battery must be present at power-up to avoid loss of program.

NOTE: Before initial program load, you should perform the following steps:

1. Disable the battery (refer to Table 2-2).
2. Install the module in the base (refer to Section 2.2.3.
3. Power up the base for 20 seconds or more

   This will clear any existing 'battery backed' user RAM data. If a programming device (VPU, Remote Terminal) is connected, a "MEMORY CLEARED" message will be displayed on the power up screen.
4. Power down the base.
5. Remove the module and set the appropriate switches for the desired operation as defined in Table 2-2.

Table 2-2 summarizes the 4-switch set. Select OFF by setting the switch to open; select ON by setting the switch to closed.

Table 2-2 Configuring the 4-Switch Set

Switch	Switch Position	Function
4	OFF ON	Battery Disabled Battery Enabled
2–3	Not Used	Not Used
1	OFF ON	RUN MODE PROGRAM MODE

2.2.3 Installing the Module

The BASIC Module may be installed in any available I/O slot. Do not touch the printed circuit board (PCB) while handling the module. This could cause electrostatic damage to the components on the PCB. See the *SIMATIC® TI545™ System Manual* for more information on installing the module.

⚠ **WARNING**

To avoid the risk of personal injury, disable all power to the system before installing or removing I/O modules.

To install the module:

1. Position the module so that the front bezel is facing you as shown in Figure 2-2.

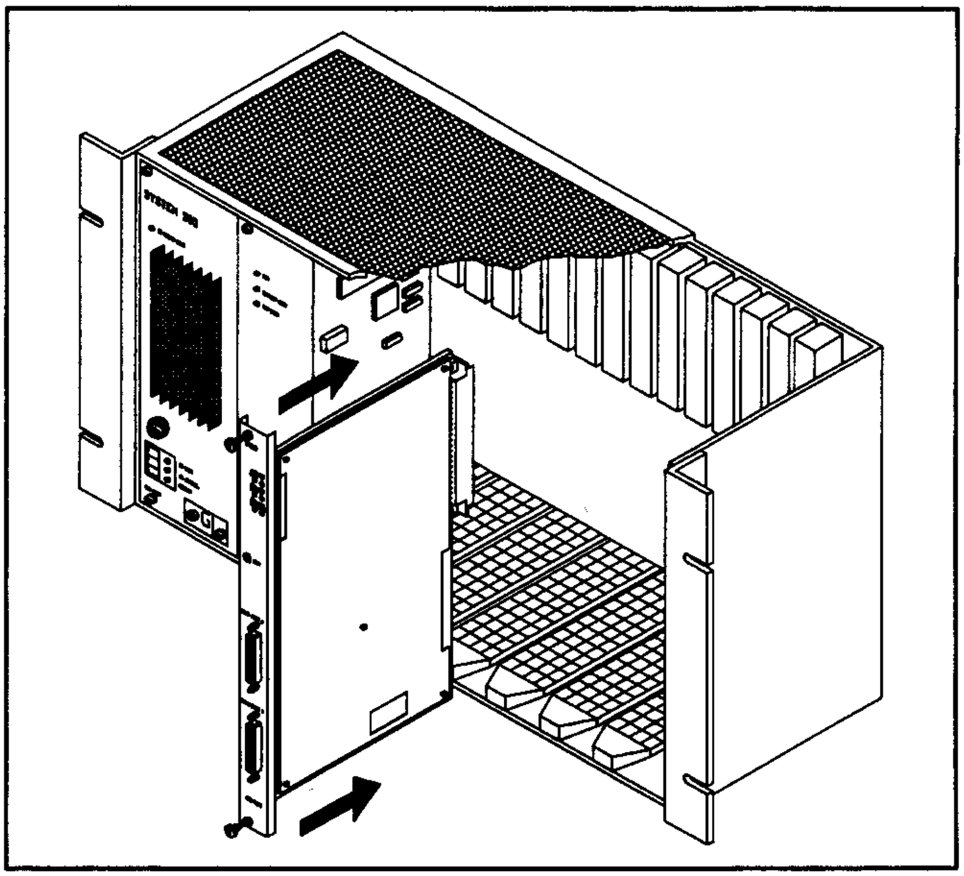


Figure 2-2 Inserting the Programmable BASIC Module

2. Hold the top and bottom of the bezel and slide the module carefully into the slot, pushing it all the way into the base. If you have inserted the module correctly, you will feel a slight increase in resistance as the module mates with the base plane connector.

3. Use a flat-head screwdriver to tighten the screws at the top and bottom of the bezel. This grounds the module to the base. Do not overtighten.

2.3 Connecting Cables and Wiring

For programming, the cable must be connected to port 1 on the module. The interface at port 1 is RS-232C/423. (Once programming is completed, you may connect a different device to port 1). Port 2 may be connected to any input or output device as desired. The following sections describe cabling procedures for a VPU and for any other programming (or non-programming) device.

⚠ **WARNING**

To avoid the risk of personal injury and equipment damage, all field devices should be powered down before attempting to connect cables.

2.3.1 Communications Ports

Communications port 1 is at the bottom of the module, with port 2 located directly above port 1 as shown in Figure 2-3. Both port 1 and port 2 are RS-232-C/423 interfaces.

For outputs, port 1 will buffer up to 128 ASCII characters, and port 2 will buffer up to 1024 ASCII characters. For inputs, both ports will buffer up to 128 ASCII characters.

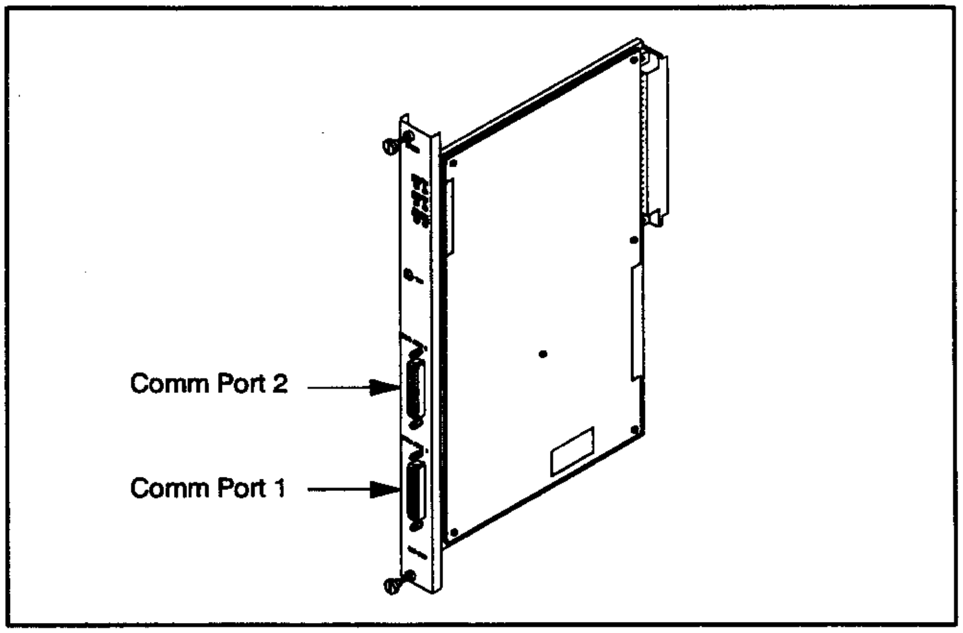


Figure 2-3 Ports Available on the Module

NOTE: If the output memory buffer becomes full, character transmissions will stop until there is room in the buffer for more characters.

Programming is allowed only through port 1.

2.3.2 RS-232-C/432 Communications

Communication for the RS-232-C/423 interfaces is bit serial asynchronous with 2 stop bits at 110 baud and 1 stop bit at other baud rates. "RS-232-C/423" refers to the standard interface in which an RS-232-C signal definition is used with RS-423 receive and transmit levels. The interface functions normally with RS-232-C input and output levels.

The output voltage for the RS-232-C/423 interface is +/− 4 volts for data and control signals.

2.4 Connecting a VPU

The VPU emulates a non-intelligent terminal after the BASIC Operating System Disk is inserted and initialized. The VPU will only transmit keystrokes and wait for messages from the module. To load or save a program with the VPU, the Operating System Disk must be replaced by a formatted disk. Each disk can store one program. All data transmissions are first buffered in the VPU memory and then transferred to the Programmable BASIC Module or the formatted disk.

The PC port on the VPU is used for programming. This port is DTE with full duplex capability and odd parity generation. The auxiliary port on the VPU may be used to connect a printer to the VPU/BASIC system (a printer could also be connected directly to the module). The auxiliary port has only half-duplex capability with no diagnostic or handshaking abilities. The VPU ports are shown in Figure 2-4.

If a printer is used with the VPU, the auxiliary port may be turned on or off as needed. The port is turned on by pressing the CTRL key and the R key at the same time. The port is disabled by pressing the CTRL key and the T key at the same time. The sequence of turning the port on or off could also be accomplished by pressing the PRINT key on the small keypad to the right of the main keys. The key will "toggle" between enabling and disabling the auxiliary port. When the printer is enabled, all communication echoed to the VPU will be sent to the printer.

The VPU screen will show all communication to and from the BASIC module in a 24-rows-by-80-columns display. The bottom line of the screen is reserved for messages describing the status of the system. All VPU error codes and messages will appear in this position. The messages will remain on the screen until a new message is generated or the Return key is pressed.

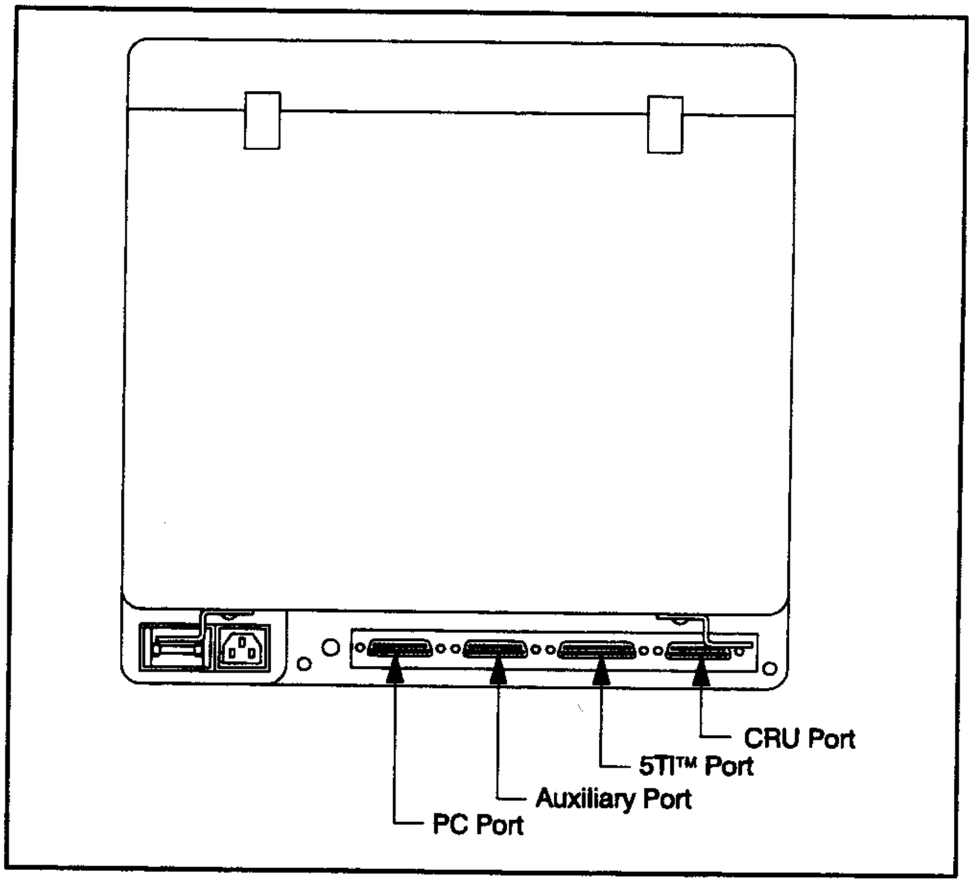


Figure 2-4 VPU Ports

You must use an RS-232-C/423 interface to program the module with a VPU. The interface connects the PC port on the VPU with port 1 of the module. An RS-232-C/423 cable (PPX:2462553–0003) is recommended for superior shielding and noise immunity.

Figure 2-5 shows a pin diagram for the RS-232-C/423 cable, should you decide to make your own. Table 2-3 defines the signals for each pin.

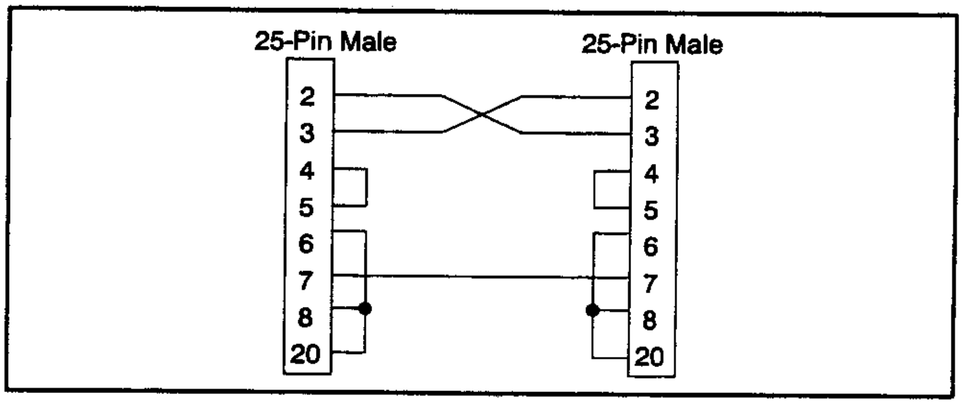


Figure 2-5 Pin Diagram

To connect the RS-23-2C/423 cable:

1. Carefully insert one end of the RS-232-C/423 cable into port 1 of the module and the other into the PLC port on the VPU.
2. Secure the cable by tightening the screws on both sides of the connector.

Table 2-3 Pin Definition

Pin #	Assignment	DTE
2	Transmit Data	Output
3	Received Data	Input
4	Request to Send	Output—turned on during transmission
5	Clear to Send	Input
6	Data Set Ready	Input
7	Signal Ground	Input
8	Received Line Signal Detector	Input
20	Data Terminal Ready	Output-driven Active

NOTE: The receivers are always enabled. In DTE mode, pins 5 and 6 must be active to enable the transmitter. The status of pin 5 may be checked with SYS(9) or SYS(10). Refer to SYS functions, Section 3.5.18.

At this point, it is a good idea to put the keycaps on the keyboard of the VPU. The keycaps come with the Programming BASIC Operating System Disk. The keycaps are placed on keys 1 through 0, the comma key, and the period key as shown in Figure 2-6.

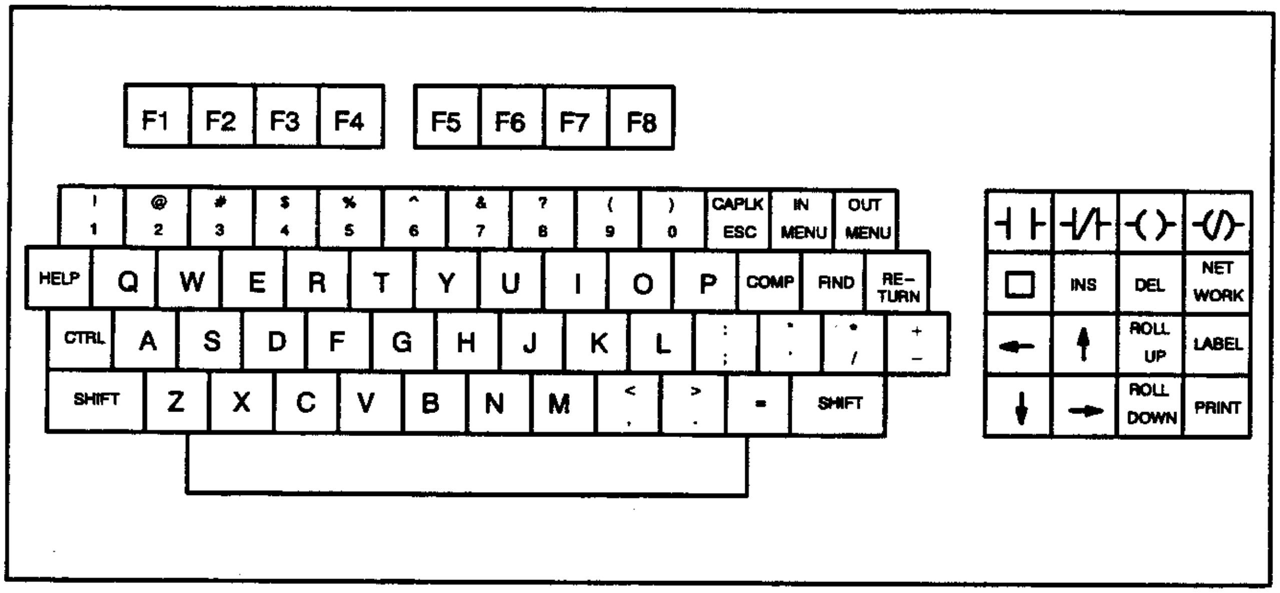


Figure 2-6 VPU Keycap Placement

2.5 Connecting Other Devices

If you use a programming device other than a VPU, connect it to port 1 of the module. Once programming is completed, a different input or output device may be connected to port 1. The device manual should tell how to connect the programming device to the Programmable BASIC Module.

2.6 Starting-Up the Programmable Basic Module

After the module has been configured, installed, and connected to external devices, power may be applied to the I/O BASE. Upon power-up, the module indicators should be observed to ensure that the module passes the self-test diagnostics. This is indicated by the MOD GOOD indicator remaining ON (after approximately 5 seconds). The BATT GOOD indicator will be ON if the battery has been enabled.

2.6.1 User-Initiated Test

The self-test diagnostics performed at power-up may be initiated at other times if the module is in the PROGRAM mode. To do this, press the TEST switch located on the face of the module. All indicators (with the exception of STAT1 and STAT2) should illuminate temporarily, blink four times in unison, and remain on as appropriate. This test should be completed within 13 seconds.

2.6.2 Verifying PLC-module Communication

After the module has been installed and powered-up, make sure that the PLC registers the presence of the module. This is important because the module will appear to be operating even if it is not communicating with the PLC.

The VPU must be connected to the PLC to verify PLC-module communications. Initialize the VPU with the PLC operating system disk.

Refer to your software manual for information on how to perform the Configure I/O Base function.

The PLC will then respond with a chart listing all slots on the base and the inputs or outputs associated with each slot. If no I/O module is inserted in a slot, that row on the chart will be left blank.

Look at the chart for the number corresponding to the slot occupied by the Programmable BASIC Module for your particular module. If the "SF" (special function) field indicates "yes," and word memory locations appear on this line, the module is registered in the PLC memory and you may proceed with the installation. If the line is blank or erroneous, check the module to be sure it is firmly seated in the slots and enter the Read Basic Function once more. If the line is still wrong, you should contact your local distributor for further assistance.

NOTE: Only 16 special function modules (BASIC, PEERLINK™, or Network Interface) are allowed in any one Series 500 PLC I/O channel. The Series 505 PLCs have one I/O channel (up to 16 special function modules allowed). The SIMATIC® TI560™/TI565™ PLCs have eight I/O channels (up to 128 special function modules allowed).

2.7 Starting-up the System

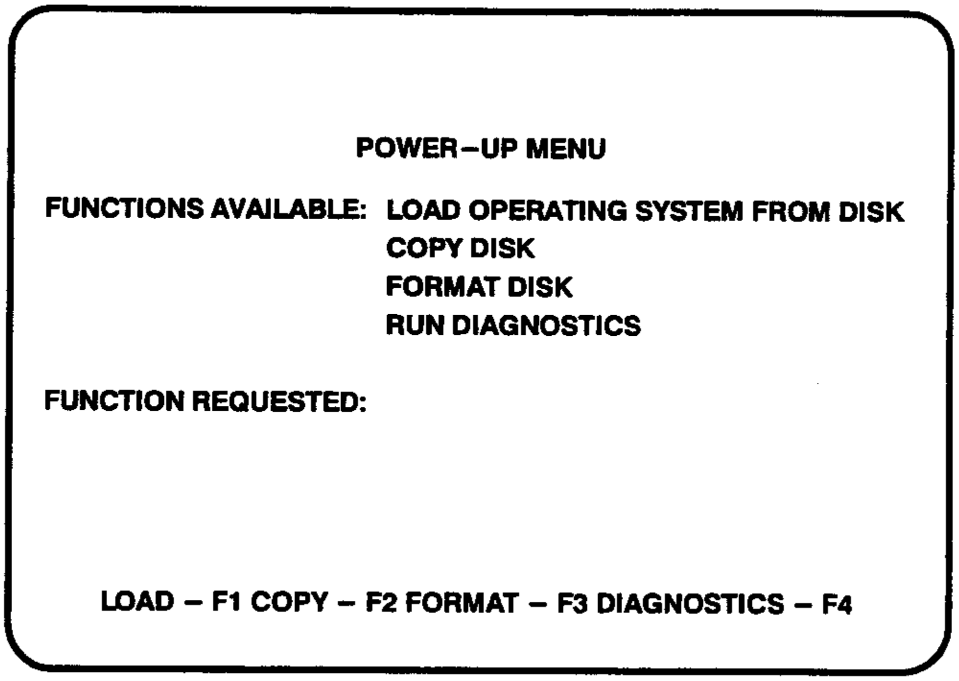
This section describes how to start-up the programming device for operation with the Programmable BASIC Module. The first section shows how to start-up the VPU for programming, and the second section describes the procedure for other programming devices.

2.7.1 Starting-up the VPU

To start-up the system for programming, first turn on the VPU. Then remove the cardboard insert from the disk drive of the VPU and replace it with the Operating System Disk (If you do not have a BASIC Operating System Disk, please see your dealer to obtain a Programmable BASIC Software Package, which includes the Operating Systems Disk).

NOTE: The VPU210 will accept both low and high density diskettes, but the basic operating system will only operate on a high density diskette. When making archive copies of the operating system, be sure to use a high density diskette.

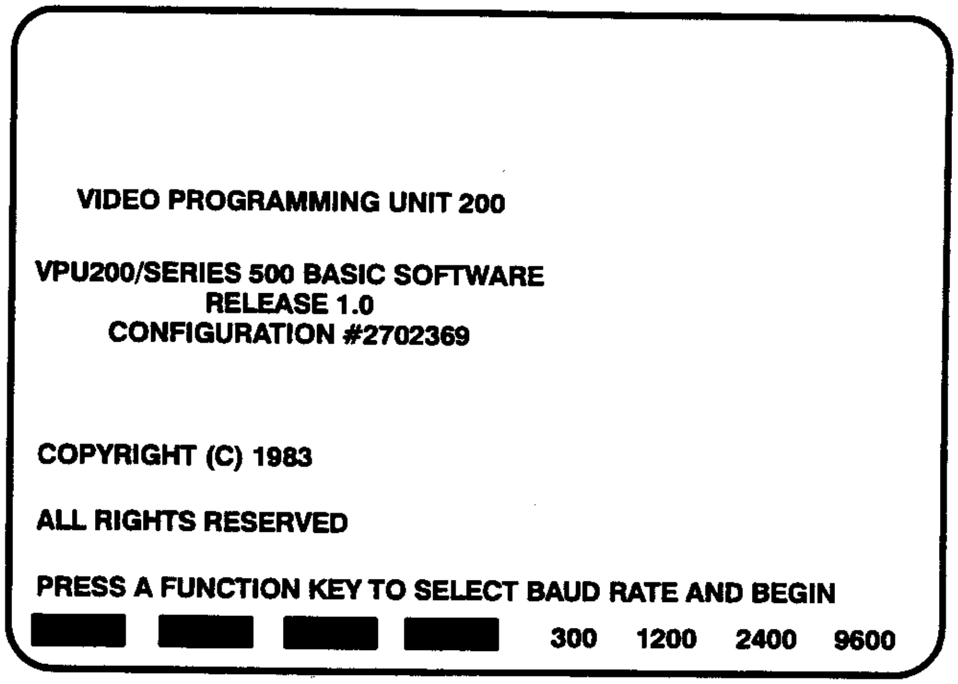
When you power up the VPU, and press the space bar, the following screen appears:



```
                          POWER-UP MENU

FUNCTIONS AVAILABLE:  LOAD OPERATING SYSTEM FROM DISK
                      COPY DISK
                      FORMAT DISK
                      RUN DIAGNOSTICS

FUNCTION REQUESTED:



    LOAD - F1 COPY - F2 FORMAT - F3 DIAGNOSTICS - F4
```

To continue, press F1 (the choice for loading the Operating System Disk) and Return. Press F5. The VPU will now load the Operating System Disk. As the VPU loads the data, a series of dots will move across the screen. When the dots stop appearing on the screen, the disk will have been loaded without any errors.

When the Operating System Disk is loaded, the following appears:


```
   VIDEO PROGRAMMING UNIT 200

VPU200/SERIES 500 BASIC SOFTWARE
           RELEASE 1.0
    CONFIGURATION #2702369


COPYRIGHT (C) 1983

ALL RIGHTS RESERVED

PRESS A FUNCTION KEY TO SELECT BAUD RATE AND BEGIN
                                  300   1200   2400   9600
```

To continue, select one of the baud rates displayed at the bottom of the screen. (There is no default value for the baud rate). Each baud rate is written above the function key used to select that baud rate.

NOTE: Be sure that the rate selected agrees with the rate set for port 1 on the module and the rate needed for any printer connected to the VPU210.

If you need to change the baud rate after starting-up the system (for example, connecting a new printer to the VPU210), you will have to re-start the VPU210.

To clear the screen, press a function key for a baud rate. The VPU will then be in non-intelligent terminal mode and will display a blank screen with a blinking cursor in the lower left corner. The VPU is now ready for communicating with the Programmable BASIC Module. Begin programming from the keyboard or remove the Operating System Disk from the VPU and insert a disk to load a program into the module. If a program is already in the module, enter RUN to begin the execution of the program.

2.7.2 Checking Communications

Press the Esc key to ensure that the module is communicating with the programming unit. If *READY appears, communication is occurring. If not, recheck cabling and switch settings.

Stop any operation (except LOAD) by pressing the Esc key. The operation will halt, and ***READY** will appear on the VPU screen.

2.7.3 Caps Lock On

All program statements and variables must be in uppercase. Program documentation may be in lowercase. When the VPU is started-up, all letters will be in uppercase. To obtain mixed case, press the CTRL key and S key at the same time. (The keyboard will return to uppercase if you press CTRL S once more). You could also change from uppercase to mixed case by pressing the CAPLK key, which is located on the top row of the keyboard.

2.7.4 Enabling VPU Auxiliary (Printer) Port

The auxiliary port is turned on by pressing the CTRL key and the R key simultaneously. The port is disabled by pressing the CTRL key and T key at the same time. You can also turn the port on or off by pressing the Print key on the small keypad to the right of the main keys. This key toggles between enabling and disabling the auxiliary port.

2.7.5 Error Diagnostics

Error diagnostics are performed by the VPU after the baud rate is selected. Any hardware or software error is printed on the bottom line of the VPU screen. There are two general categories of errors: disk errors and communications errors. The disk errors include the following:

- Format error.
- Read/Write attempt on operating system disk.
- Hardware error.
- Read/Write attempt on non-formatted disk.
- Bad disk track.
- Read/Write error.
- 5-second time-out.

When there is an error in either saving or loading a disk, one of the following messages is displayed:

DISK SAVE ERROR

DISK LOAD ERROR

A parity error causes the following message to be displayed:

RECEIVE COMMUNICATION ERROR

This error does not halt the command being executed; however, since a communication error will yield unpredictable data, you must re-execute the current command after fixing the communication error. Ensure that you have the correct cable connections and that the configuration switches are set correctly.

2.7.6 Starting-up Other Programming Devices

For a programming device other than the VPU, read the device user manual for information on starting-up and connecting it to the Programmable BASIC Module.

2.8 Hardware Troubleshooting

Hardware problems most likely to occur are:

- Display failure—the VPU or other programming device fails to display anything.
- Module failure.
- Battery failure.

2.8.1 Programming Device and/or Module Failure

If the programming device does not become active or the MOD GOOD indicator light does not light, check all connections to the module, the programming device, and the Series 505 I/O base. Make sure that the module and programming device are receiving power. Power down the device, disable the battery, wait two minutes, then enable power. This sometimes clears a fatal error (the program in the module will be lost). If the module or programming device remains inactive after you have checked the connections and power, and powered down and back up, contact your local distributor.

NOTE: If you press Esc while UNIT 0 or UNIT 2 is in effect, keystrokes after pressing Esc are not shown (See Section 3.4.15). To see the key-strokes on the device, you must type UNIT 1 and press Enter (key-strokes will not be seen on the device connected to port 1).

2.8.2 Battery Failure

The BATT GOOD indicator will be lit when the battery is in place and able to back up the BASIC memory and internal clock. If the indicator is not lit or goes out during operation (and switch 4 of the 4-switch configuration set is set to closed), the battery must be replaced.

2.8.3 Changing the Battery

Follow these steps to change the battery:

1. Disconnect power from the base and remove the BASIC module.
2. Remove the screw that connects the shield to the standoff which is located in the corner near the battery. (NOTE: Save the star washer installed between the standoff and the copper surface of the shield).
3. The shield can be flexed to allow removal of the battery. Lift the battery hold-down clip and slide the battery from the holder.
4. Insert a new battery.
5. Replace the star washer between the copper side of the shield and the standoff.
6. Reinstall the screw in the standoff.

NOTE: In order to insure proper operation in a high electrical noise environment, ensure that step 5 above is properly performed.

Chapter 3

Programming Language

3.1 **Introduction** 3-3

3.2 **Operation Modes** 3-4

3.2.1 RUN Mode 3-4
3.2.2 PROGRAM Mode 3-4
3.2.3 Power-up Modes 3-4
3.2.4 Source Statement 3-6
3.2.5 Constants 3-7
3.2.6 Variables 3-8

3.3 **BASIC Commands** 3-11

3.3.1 CONTINUE Command 3-11
3.3.2 LIST Command 3-12
3.3.3 LOAD Command 3-12
3.3.4 NEW Command and Retentive Memory 3-13
3.3.5 RUN Command 3-14
3.3.6 SAVE Command 3-14
3.3.7 SIZE Command 3-15

3.4 **BASIC Statements** 3-16

3.4.1 DEF Statement 3-16
3.4.2 DIM Statement 3-17
3.4.3 ERROR Statement 3-18
3.4.4 ESCAPE/NOESC Statement 3-20
3.4.5 FIND Statement 3-21
3.4.6 FOR/NEXT Loop Statement 3-22
3.4.7 INPUT Statement 3-25
3.4.8 LET Statement 3-27
3.4.9 PRINT Statement 3-28
3.4.10 RANDOM Statement 3-35
3.4.11 REM Statement 3-36
3.4.12 STOP/END Statements 3-36
3.4.13 TAB Statement 3-37
3.4.14 TIME Statement 3-38
3.4.15 UNIT Statement 3-39
3.4.16 Internal Data Statements 3-40
3.4.17 Branch Statements 3-41
3.4.18 CALL (PLC Interface Statement) 3-44
3.4.19 Subroutine Statements 3-52

3.5 BASIC Functions **3-54**
3.5.1 ABS Function 3-54
3.5.2 ASC Function 3-54
3.5.3 ATN Function 3-54
3.5.4 BIT Function 3-55
3.5.5 COS Function 3-58
3.5.6 EXP Function 3-58
3.5.7 INP Function 3-59
3.5.8 LEN Function 3-59
3.5.9 LOG Function 3-60
3.5.10 MCH Function 3-60
3.5.11 MEM Function 3-61
3.5.12 MWD Function 3-61
3.5.13 NKY Function 3-62
3.5.14 RND Function 3-63
3.5.15 SIN Function 3-63
3.5.16 SRH Function 3-64
3.5.17 SQR Function 3-64
3.5.18 SYS Function 3-64
3.5.19 TIC Function 3-66
3.5.20 String Manipulations 3-66

3.6 BASIC Operations **3-72**
3.6.1 Arithmetic Operators 3-72
3.6.2 Logical Operators 3-72
3.6.3 Boolean Operators 3-73
3.6.4 Relational Operators 3-73
3.6.5 Order of Operator Evaluation 3-74

3.7 Editing a Program **3-75**
3.7.1 General Editing Procedures 3-75
3.7.2 Editing with a VPU 3-77

3.1 Introduction

This section describes the modes of operation for the BASIC programming language, general aspects of the system, and procedures for editing. BASIC is composed of commands, statements, operators, and functions. Each is discussed in a separate section. Appendix A provides a summary of the commands, statements, and functions discussed in this chapter.

3.2 Operation Modes

BASIC has two modes of operation: RUN and PROGRAM.

3.2.1 RUN Mode

The program is read and/or executed when one of the following commands is entered from the keyboard:

- RUN
- CONTINUE
- GOTO

RUN mode is terminated (and PROGRAM mode entered) when:

- The end of executable statements is reached.
- The Esc key is pressed.
- An error is encountered.
- When either of the following commands is entered from the keyboard:

  END

  STOP

3.2.2 PROGRAM Mode

The program is loaded or edited in PROGRAM mode. The following commands, entered one line at a time, function in the PROGRAM mode:

- CONTINUE
- LIST
- LOAD
- NEW
- RUN
- SAVE
- SIZE

Control returns to the keyboard after each line is executed.

3.2.3 Power-up Modes

The module may be configured to power-up in either PROGRAM or RUN mode. In PROGRAM mode, the module waits for a command from the input device before beginning any action. In RUN mode, the module begins execution of the installed BASIC program without any prompting from the keyboard. You may select the power-up operating mode by setting the proper configuration switch.

⚠ WARNING

When the module is set to "RUN" for power-up mode, field devices under the control of the module application program may begin operating when power is restored. The sudden start-up of these devices could endanger personnel and equipment. To avoid personal injury or damage to equipment, before restoring power to the module: 1) Ensure that all personnel are clear of machinery areas controlled by the module application program; 2) ensure that all equipment is prepared for start-up operation.

After a power interruption, the module enters the mode selected by the configuration switch if the battery is good and enabled. All memory areas, except the retentive and program memory areas, are cleared; all system pointers are initialized; and execution begins at the initial program statement again (if in RUN mode). If in PROGRAM mode, the following message appears on the display after power is restored:

```
SERIES 505 PROGRAMMABLE BASIC MODULE *READY
```

If the battery is weak, disabled, or absent, then all memory areas, including retentive and program memory areas, are cleared. After power is restored, the module will be in PROGRAM mode regardless of the setting of the configuration switch and will display the following message:

```
MEMORY CLEARED
SERIES 505 PROGRAMMABLE BASIC MODULE *READY
```

After this message appears, enter the program that was lost during the power interruption and begin execution again.

3.2.4 Source Statement

The following are two sample formats for BASIC:

- <line number> <statement>::<optional statement 2>
- <line number> <statement>!<optional remark>

For example, the following is a valid BASIC statement:

```
50 LET A=10::PRINT A ! PRINT THE VALUE OF A
```

The format for any BASIC statement begins with a line number between 1 and 32767 inclusive. The line number should have no embedded blanks. The line numbers do not have to be entered in consecutive order, but the interpreter reads the statement numbers in ascending numerical order regardless of how the statements are entered. After the program is entered, BASIC rearranges and stores the statements in an ascending numerical order. It is wise to leave "gaps" between statement numbers (incrementing by 5 or 10 for each statement number) so that additions to the program may be easily inserted. This may be done manually, or automatically, by using the Line Feed key (or CTRL and J keys at the same time on a VPU) rather than the Return key when you generate a new line of the program. This will automatically increase the line numbers by 10.

Multiple statements on one line are allowed. A double colon (::) is used to separate the statements.

The characters used in BASIC are A–Z, 0–9, in addition to several special characters that will be introduced in this chapter. Characters may be used in mixed cases, but all keywords must be in uppercase. If your input or output device does not print all characters, then the character must be represented by its HEX-ASCII code. The HEX-ASCII code is enclosed in angle braces and inserted where the non-printable character would be. HEX-ASCII codes are only allowed in PRINT statements and string constants. Appendix B contains the HEX-ASCII codes for the VPU; for other programming devices, consult that device manual for the appropriate codes.

Non-executable statements, such as remarks, are separated from the rest of the line with an exclamation point (!) (see Section 3.2.6). The non-executable statement must be the last entry on the line. You may also use the REM statement to include entire remark statements that will be ignored during execution.

3.2.5 Constants

The Programmable BASIC Module supports HEX (Hexadecimal) integer constants, decimal integer constants, decimal real constants, and string constants.

A decimal integer is any integer between −32,768 and 32,767; it requires 48 bits (6 bytes) to store. Decimal reals (floating point numbers) are numeric values with decimal fractions. As with integers, 48 bits (6 bytes) are used to represent floating point numbers. With 48 bits (6 bytes), accuracy to roughly 11 significant digits and numbers between +/−1E74 and +/−1E−74 may be expressed. A floating point number may be expressed as either a number and a decimal fraction or as an exponent with the base number. For example, both 123.4 and 1.234E2 are proper decimal constants.

Hex constants are terminated with an H. There can be no embedded blanks, and A−F cannot be used as initial HEX digits unless the letter is preceded by a 0. If more than four digits are given, only the right-most four are used. Valid combinations range from 0H to 0FFFFH.

A string constant is a set of ASCII characters enclosed in either single or double quotes. Single quotes may be used to enclose double quotes and vice versa. For example, 'THIS IS A "VALID" STRING' and "SO IS THIS". Non-printable characters may be placed in a string by using their HEX-ASCII equivalents in angle braces. When the HEX-ASCII character is stored, it will occupy 4 bytes of memory; however, when it is used, it is interpreted as 1 byte. When the string is stored, the brackets are stored with the string, and when the string is used, the brackets are removed with HEX-ASCII being changed to HEX. Therefore, attempting to compare both HEX-ASCII strings will result in errors. Following is an example of a HEX-ASCII constant stored in the variable A:

```
10 $A = "<07>"
```

3.2.6 Variables

BASIC supports simple numeric variables, numeric array variables, simple string variables, and string array variables. All arrays must be dimensioned before they are used. If this declaration is not made, numeric array variables will result in an error, and string array variables will be set to null when they are first used. As with constants, variables are represented in 48 bits (6 bytes), which allows accuracy to roughly 11 significant digits and numbers between +/−1E74 and +/−1E−74 to be expressed.

Variable names must be in capital letters. The name may be composed of one, two, or three letters. A variable name may also be a single capital letter followed by a number between 0 and 127. The letters assigned to the variable cannot be the same as a BASIC command, statement, or function. For example, neither COS nor LIS could be a variable name because they conflict with the COSINE function and the LIST command. See Appendix C for a list of reserved words.

A string variable must be preceded by a dollar sign ($) except when in the DIM declaration for an array and in the subroutines of the CALL statement. The dimension declaration is discussed in Section 3.4.2. CALL is used with the PLC interface subroutine. For example, even though they share the same memory area, "A" is a numeric variable but "$A" is a character variable. There is no difference between numbers and characters at the memory level; the difference arises from how they are used (i.e., character strings must have a dollar sign but numeric variables must not). For clarity, it is best to name each variable differently, whether that variable is to be a number or a character.

Each array name must be followed by a subscript to indicate a particular location in the array. There is no limit to the number of subscripts that an array may have. For example, both A(2,1) and D12(10) are valid arrays. The number of positions in array A would be six: A(0,0), A(0,1), A(1,0), A(1,1), A(2,0), and A(2,1). Similarly, array D12 will have 11 positions. Note that the positions of arrays begin at 0 and not 1. More information on arrays, including the memory level representation, is found in the discussion of the DIM statement (Section 3.4.2) and the BIT function (Section 3.5.4).

One hundred twenty-eight unique variable names are allowed; however, dimensional variables may be used to allow more data storage. For example, XYZ is a unique variable name and can hold one variable of data, whereas XYZ(10) is one unique variable name that can hold 11 variables of data (XYZ(0) through XYZ(10)).

If you try to assign more than 128 variable names, the **TOO MANY VARIABLES** error will appear. To clear this error:

1. Reduce the number of variables used.
2. Store the program to disk.
3. Clear the module memory using the NEW command.
4. Reload the program.

Any variable name assigned is maintained in memory even though you have deleted it from the program. The only way to reassign variable memory is to clear the memory.

There is also a special array used for storing values that you wish to protect from being lost in a power interruption or after execution is restarted with a RUN command. This array is named RET, which is an abbreviation for retentive memory area. RET is treated like any other array except that it cannot be dimensioned with the DIM statement. The size of the array is set to 1 at power up. This allows six bytes of retentive memory. You may increase the size of the retentive memory area by using the NEW command.

NOTE: The size of the retentive array is not stored on disk. Before loading a program that uses retentive memory from disk, use the NEW command to set the size of the RET array. Since the NEW command clears all program and variable values, set the size of RET *before* you begin to enter a program.

String variables may have up to 5 characters, with a null being automatically inserted after the last character. The null is important because it is the flag to tell BASIC to stop reading the string. This can be useful when you wish to use strings that are longer than 5 characters. For example, you would use an array to store the following names: WAREHOUSE, INVENTORY and MANAGEMENT. Each of these is too long to fit in a single simple string, so an array must be used. The array would be dimensioned, and then the names would be entered as follows:

$NML(0,0): = WAREHO	$NML(0,1): = USE
$NML(1,0): = INVENT	$NML(1,1): = ORY
$NML(2,0): = MANAGE	$NML(2,1): = MENT

Notice that each of the array locations in the first column has six letters. Since the null has been overwritten, BASIC will read the first two array locations before stopping. (BASIC continues through all locations until a null is found.) So, entering PRINT $NML(0,0) would result in WAREHOUSE being printed. Similarly, PRINT $NML(1,0) and $NML(2,0) would cause the full names to be printed. PRINT $NML(0,1) would result in USE being printed.

As this example shows, overwriting the null causes dramatic changes in the way variables are read. Be careful when using six characters instead of five for character strings either alone or in arrays.

⚠ CAUTION

Regardless of the size given to a string array, any characters may be entered into that variable. Any characters beyond the size of the variable will overwrite other memory areas, which may cause unpredictable operation.

3.3 BASIC Commands

The BASIC commands are functional only in the PROGRAM mode and thus cannot be used in a program. They are used to direct system functions. The commands interact with the system to initiate immediate action. Do not confuse commands with statements: Commands allow operator control; Statements are steps in a program.

The command keywords are: CONTINUE, LIST, LOAD, NEW, RUN, SAVE, and SIZE. CONTINUE, LIST and SIZE may be abbreviated to their first three letters. The other commands must be completely written. To enter commands, type them in, then press [Return]. The following sections describe each of these commands.

NOTE: Several statements can also be used as commands in PROGRAM mode. These include GOTO, PRINT, SYS, TIME and UNIT.

3.3.1 CONTINUE Command

The CONTINUE (CON) command restarts a program after a break. If the break occurred because of an error or an ESCAPE entry from the keyboard, CON will cause execution to restart on the interrupted line. If the break occurred because of a STOP statement, CON will cause execution to restart at the line following the interrupted line. You cannot use CON to proceed beyond an END statement, nor can you use it to restart execution after you have edited any part of the program.

Note that CON differs from RUN in that RUN reinitializes all variables and returns to the beginning of the program. CON always starts from the point at which a break occurred and does not affect the values of any variables in the program.

3.3.2 LIST Command

The LIST (LIS) command displays all or part of a current program. If no line number precedes the LIS command, display begins at the lowest numbered line and ascends. When a line number is given, the statements from that point to the end of the program are displayed. The given line number need not be a number in the program: the LIS command begins at the line number of the program equal to or greater than the given line number. The listing of a program may be stopped at any time by pressing the Esc key; however, the program listing will not stop until the RAM buffer memory is emptied. If there is a long program listing in the buffer, there may be a long delay before the listing stops.

NOTE: When a program is listed, both square brackets and parentheses are used. Square brackets appear only around dimensioned variables and functions; parentheses are used at all other times. You do not need to enter square brackets when programming; parentheses are automatically changed to square brackets when the program is listed.

3.3.3 LOAD Command

The LOAD command transfers a program from a disk into the Programmable BASIC Module memory. Once the VPU has been initialized, the Operating System Disk can be removed and the program disk inserted. Entering LOAD causes the program to be transferred from the disk, through the VPU buffer memory, and into the memory of the Programmable BASIC Module.

When LOAD is first entered, the VPU screen displays "DISK LOAD IN PROGRESS." Once the transfer of data begins, the display changes to "DATA TRANSFER IN PROGRESS." When the LOAD operation is complete, "DISK LOAD COMPLETED" is displayed on the bottom of the VPU screen.

NOTE: The program being loaded will over-write the statements that have the same line numbers. Statements with different line numbers will not be affected. While this may be useful in merging two or more programs, it is best to avoid the problems that this may cause by using the NEW command before loading a new program.

Any errors that occur during loading will halt the procedure at the point of error. All statements before this position will be successfully transferred. However, you have to reload the program to get the complete program. "DISK LOAD ERROR" appears on the bottom of the VPU screen to indicate the error that caused LOAD to stop.

The time it takes to LOAD a program depends on the baud rate selected for the VPU. This time may vary from 90 seconds to 19 minutes.

NOTE: The Esc (abort) key is disabled during LOAD operations, so there is no way to interrupt a LOAD operation without powering down the VPU.

It is recommended that you do not use the LOAD command for programming devices other than a VPU.

3.3.4 NEW Command and Retentive Memory

The NEW command deletes the current program, sets the RET array to its default value of 0, and clears all variable spaces, stacks, and pointers. When the command is complete, the Programmable BASIC Module responds with the following:

```
MEMORY CLEARED
SERIES 505 PROGRAMMABLE BASIC MODULE *READY
```

Once the command has finished, a new program may be loaded into the cleared space.

NEW may also be used to set the RET array to a value different from 0. RET may be dimensioned from 0 to 4095 6-byte elements with the NEW command. Any change in the RET array must be done before a new program is written, because using the NEW command destroys the current program. To dimension RET to a new size, write the size needed after the word "NEW." The size may be expressed as either a number or an expression. Each number, or the number to which the expression evaluates, reserves 6 bytes of retentive memory space (remember, since RET is an array, memory positions begin at zero). For example, to reserve 3 elements (18 bytes) of retentive memory, *and* destroy the current program and memory contents, if any, enter the following command:

```
NEW 2
```

3.3.5 RUN Command

The RUN command is used to begin the execution of a program. It clears all variable spaces, stacks, and pointers and begins execution at the lowest line number in the program.

3.3.6 SAVE Command

The SAVE command transfers a program from BASIC memory, through the VPU buffer memory, and onto a formatted disk. The program remains in BASIC memory after the SAVE operation.

When the SAVE operation is entered, "DATA TRANSFER IN PROGRESS" is displayed on the bottom of the VPU screen. During the transfer of data, "DISK SAVE IN PROGRESS" is displayed. Once the program has been written to the disk, "DISK SAVE COMPLETED" is displayed.

Any errors in the operation cause the SAVE operation to stop. "DISK SAVE ERROR" appears on the bottom of the VPU screen to indicate why the operation stopped. Once the error is corrected, you must restart the SAVE operation from the beginning of the program.

The time it takes to SAVE a program depends on the baud rate selected for the VPU. This time may vary from 90 seconds to 19 minutes.

NOTE: If you interrupt a SAVE operation by pressing the Esc (abort) key, the program already on the disk will be destroyed, and the SAVE operation will stop.

Do not use the SAVE command for programming devices other than the VPU.

3.3.7 SIZE Command

The SIZE (SIZ) command is used to determine the current program size, variable space allocated, and free memory (in HEX bytes). For example, after a "NEW 0" command (memory and program cleared, with RET set to 6 bytes), entering SIZ would result in the following:

```
PRGM:    014H BYTES
VARS:    0CH BYTES
FREE:    06FE6H BYTES
```

"PRGM" refers to the current user program size, "VARS" to the variable space allocated; and "FREE" to the amount of remaining memory space.

3.4 BASIC Statements

Statements form the sequential list of instructions called a program. All statements except DEF, DATA, END, READ, RESTOR, STOP, and TAB may be used without line numbers in PROGRAM mode. When a statement is entered without a line number, it initiates immediate action. For convenience the statements are listed in alphabetical order.

NOTE: In the descriptions that follow, angle braces (< >) are used around certain components of the format statements. The braces and elements within them provide information about what is required at that location in a statement format. Braces should not be entered when using the particular statement in a program; they are used only to help you read the format for the statement. All other components of the format for a statement should be entered as shown.

3.4.1 DEF Statement

A DEF statement defines user functions. These functions are executed only when referenced at other places in the program. DEF must appear before the function is used, and usually these statements appear as a group at the beginning of a program. Once defined, the function operates like any other mathematical function that is included as a part of BASIC.

The format for a DEF is:

```
<line number> DEF FN <LTR> (dummy variables) = <exp>
```

The line number is required, and also, DEF is not used on a multi-statement line and it is not followed by a tail-remark (!).

The name of the function must always begin with "FN" and end with an uppercase "LTR", which is one letter between A and Z. The letter chosen can only be used for one particular function. This requirement necessarily limits the number of functions to 26.

The equation or expression for the function is "exp." It is any valid BASIC expression. A function, however, cannot include itself as a part of the definition. For example, the following statement is invalid:

```
10 DEF FNI(A) = 2*FNI(A)
```

Dummy variables are variables used only in this function. There must be at least 1 but not more than 3 dummy variables in each function. Each dummy variable is a single uppercase letter that is meaningless except within the function. The letters are "dummies" and are replaced with the variables used when the function is executed. The names of the dummy variables may be the same as other variables in the program; there will be no conflict when the program is executed. For example, the following statement defines a function:

```
10 DEF FNQ(A,B,C) = (−B+(B^2−4*A*C))/2*A
```

When this function is executed, the variables used at the time of calling will replace A, B, and C. For example:

```
100 X(0)=FNQ(DAT(0),DAT(1),DAT(2))
```

will use the FNQ function previously defined with DAT(0) replacing A, DAT(1) replacing B, and DAT(2) replacing C.

Other valid DEF statements are the following

```
20 DEF FNA(X,Y) = X/Y+5
30 DEF FNB(A,B,C) = A/B+C−15
40 DEF FNN(I) = N*2/SQR(Z)
```

3.4.2 DIM Statement

The DIM statement initializes the space available for an array. This must be included before the array is referenced or an error results. You cannot change the dimensions of an array later in a program. Also, the DIM statement cannot be used to set the size of RET (the retentive memory array). RET is dimensioned by using the NEW command.

The format for the DIM statement is:

```
<line number> DIM ARRAYNAME(size 1,size 2,....,size n)
```

The values in the parentheses are the maximum subscript values allowed in each dimension. These values may be numeric constants, simple variables, other dimensioned variables, or function calls. If a function returns a real value, only the integer portion will be used. When using the subscripts to find a particular location, the starting point is zero and continues to the end value assigned. Each address in an array occupies 6 bytes of memory. For example, an array dimensioned to 2 would contain 3 elements and occupy 18 bytes of memory.

The following is an example of a DIM statement:

```
10 DIM CAT(10),B(1,2)
```

This initializes the string array "$CAT" to 11 elements ($CAT(0) through $CAT(10)) and the numeric array "B" to a 2 X 3 matrix containing 6 elements: B(0,0),B(0,1),B(0,2),B(1,0),B(1,1), and B(1,2). In terms of bytes, "$CAT" has 66 bytes of available storage and "B" has 36 bytes of storage. Note that the string array does not have a dollar sign preceding it in the DIM statement. This and the CALL (See 3.4.18) statement, which is used with the PLC interface subroutines and the FIND statement, are the only statements in which the dollar sign is omitted. All other references to a string array must use the dollar sign when the array is to contain character data.

A more detailed description of the bit-level representation of arrays is given with the BIT function.

3.4.3 ERROR Statement

The ERROR statement transfers execution to a particular location when an error occurs. The statement has the following format:

```
<line number> ERROR <line number>
```

The second line number is the line number to which execution will go when an error is detected.

This statement works in conjunction with the SYS(1) and SYS(2) functions. SYS(1) will contain the number of the error encountered, and SYS(2) will contain the number in which the error occurred. See Appendix D for a list of error messages. The statement to which execution is transferred may be a subroutine designed to investigate the error. The error would be investigated and execution restarted at the line after the statement containing the error.

Each ERROR statement can be used only once. After an error triggers the ERROR statement, another ERROR statement must be included to reset the error flag. You may include as many ERROR statements in a program as you need.

NOTE: Since the ERROR statement uses the memory stacks in the module, a stack overflow error cannot be detected with the ERROR statement.

The following program illustrates the use of the ERROR statement and an error subroutine:

```
100 ERROR 1000
........
1000 REM Read data error handler
1010 REM If PLC does not respond, report alarm
1020 IF SYS(1)=51! THEN PRINT "comm error with PLC"
1030 ERROR 1000 ! Reset error statement
1040 RETURN
```

Press Return to terminate the error subroutine.

One example of how to use the ERROR statement follows:

Problem: The Programmable BASIC Module automatically changes itself from RUN to PROGRAM mode when the relay ladder logic (RLL) program is modified. When the RLL is modified, the PLC is either taken to PROGRAM mode, or a "pause" in the RUN mode is allowed. Either way, during this time there is no communication to special function modules. Therefore, if the Programmable BASIC Module issues a PLC comm subroutine during this time, the PLC will not respond, and a PLC comm error will be issued by the Programmable BASIC Module. The comm error then causes the module to fall out of RUN mode.

Solution: The ERROR statement in BASIC can be used to capture the error and allow the module to continue in the RUN mode. The ERROR statement works in conjunction with SYS(1) and SYS(2) to tell you what the error encountered was, and on what line it occurred. You may choose to ignore the error or initiate some action based on the error type. In this example, it may be decided to print that a communication error occurred and to continue without retrying the communication.

3.4.4 ESCAPE/NOESC Statement

This statement either allows or prevents the Esc key on the keyboard from halting program execution. It is most often used to protect critical operations from interruption (by disabling the Esc key) and then allowing the operation of the Esc key during non-critical operations. The default value is ESCAPE; NOESC must be entered at the beginning of a program if that program is to run without the possibility of interruption. A sample statement would be:

```
10 NOESC ! The ESCAPE key is now disabled
100 ESCAPE ! The ESCAPE key is now enabled
```

If the Esc key is pressed while a NOESC statement is in effect, the ESCAPE keystroke is stored until an ESCAPE statement is encountered, at which time program execution is halted.

⚠ WARNING

When "NOESC" is selected, there is no way to halt program execution except by powering down the module. The NOESC statement must be used only when absolutely necessary. To avoid personal injury or damage to equipment, before restoring power to the module: 1) Ensure that all personnel are clear of machinery areas controlled by the module application program. 2) Ensure that all equipment is prepared for start-up operation.

3.4.5 FIND Statement

The FIND subroutine is used to locate the address of a variable in BASIC memory. It is used in conjunction with the MEM and MWD functions. FIND locates the address of a variable in memory and then NEM or MWD can be used to read or alter the data at that address. The format for FIND is:

```
<line number> CALL "FIND",<variable name>,<variable name>
```

The first "variable name" is the variable for which the location is desired. The second variable name is where the location of the sought variable will be stored once the variable is found. Since this statement involves the use of CALL, no dollar sign is used with character variables.

NOTE: The FIND statement, the PLC interface subroutines, and the DIM statement are the only statements that do not use a dollar sign. All other references to character variables must use a dollar sign.

If you wish to find the location of VAR(0), enter the following statements:

```
10 DIM VAR(10)
.........
100 ADR=0
110 CALL "FIND",VAR(0),ADR
```

After this is executed, the first location of the address for VAR(0) will be in ADR. If the sought variable is not found, error message 40 (undefined variable) appears. This error message can also arise if the variable in which the sought variable address is to be stored is undefined. A variable is defined if it appears on the left side of the equal sign (=) before it appears on the right side of the equal sign.

3.4.6 FOR/NEXT Loop Statement

FOR and NEXT appear together and indicate the start and end of a repeating instruction block. The block is a loop in which a specific variable is increased or decreased and then operated upon. When the variable reaches a preset value, the loop ends and execution passes to the statement following the NEXT statement. The formats for these two statements are:

```
<line number> FOR <var>=<exp> TO <exp> STEP <exp>
<line number> NEXT <var>
```

In the FOR statement, "var" is the variable that will be altered each time through the loop. This variable must be a simple variable (not an array) and must be the same as that used in the NEXT statement. Also, the variable may be used elsewhere after the FOR-NEXT loop is completed.

The different "exp"s (expressions) can be any valid BASIC expressions. The first expression is the beginning value for the variable and the second expression is the last value for the variable. When the last value is reached or exceeded, the loop stops and execution passes to the statement following NEXT. When the loop ends, the value of the variable will be the last value used in the loop plus the STEP value. The final expression, after the word STEP, is the amount added to the variable each time through the loop. The amount added may be negative or positive. If STEP <exp> is omitted, the default value is 1.

For example, the following are all valid FOR-NEXT statements:

```
10 FOR X=0 to 3 STEP D
(instructions for this loop)
50 NEXT X

60 FOR X4=(17+COS(Z))/3 TO 3*SQR(10) STEP Y4
(instructions for this loop)
110 NEXT X4

120 FOR A=8 TO 3 STEP -1
(instructions for this loop
170 NEXT A

200 FOR X=1 TO 3000
(instructions for this loop)
250 NEXT X
```

Notice in this example that the values may be incremented or decremented. The direction of the steps must coincide with the direction between the beginning and ending values. If the step value is positive, the ending value should be greater (more positive) than the beginning value. Likewise, if the step moves in a negative direction, the end value should be less than (more negative) than the beginning value. For example, if statement 50 had been written as:

```
50 FOR X=8 to 3 STEP 2
```

the statement would be ignored, because the step value is positive and the end value is less than (a negative direction) the beginning value.

FOR-NEXT loops may be nested up to 10 deep provided that each loop has its own specific variable. Each loop is indented to aid readability when the program is printed. The following is a typical FOR-NEXT loop.

```
10 FOR A=1 TO 2
20   FOR J=3 TO 7 STEP 2
30     FOR Z=-1 TO -6 STEP -1
40     X=A*J*Z
50     PRINT X
60     NEXT Z
70   NEXT J
80 NEXT A
90 END
```

When FOR-NEXT loops are nested, the right-most (deepest) is executed entirely, then the next loop left is incremented, and the process repeats. With the above example, A will be set to 1, J will be set to 3, and Z will vary from −1 to −6. When Z reaches −6, J will increment to its next value (5), and Z will again vary from −1 to −6. When J finally reaches 7 and Z reaches −6, A will increment to 2, and the entire range of values for J and Z will begin again.

You should not transfer execution into a FOR-NEXT statement.

⚠ **WARNING**

Be careful when putting a "FOR" and "NEXT" on the same statement line. An infinite loop could occur, which could not be stopped except by powering down the module. To avoid personal injury or damage to equipment, before restoring power to the module: 1) Ensure that all personnel are clear of machinery areas controlled by the module application program. 2) Ensure that all equipment is prepared for start-up operation.

3.4.7 INPUT Statement

The INPUT statement allows values for variables in the program to be entered from a keyboard. The NKY (See 3.5.13) function may be preferable for some applications. Up to 80 characters may be entered per string or character variable before writing into variable memory. A question mark is the prompt for numeric values, and a colon appears if a character string is required. A double question mark (??) appears if a character string is entered when a number is required. All values entered are echoed on the programming device, and the module continues prompting until all values are entered. Also, all material enclosed in double quotes and inserted into the INPUT statement is written as is when BASIC prompts for values.

If the value entered for a variable is incorrect, you may correct it before entering it by pressing the Del key if you are using a VPU. This will backspace and remove the incorrect character.

The format for an INPUT is:

```
<line number> INPUT<value 1>;<value 2>;...;<value n>
```

For example, consider the following section of a program:

```
40 INPUT X
50 INPUT $A,$B
60 INPUT $Y,Z
70 PRINT X,$A,$B,$Y,Z
80 STOP
```

If this were executed, the module would prompt for data as follows:

```
?  (prompt for a value for X) 256
:  (prompt for a character string) CAT
:  DOG
:  HI
?  80A
?? (the last entry was invalid; re-type:) 80
```

Then, the module would print the values that were entered. The BASIC system also allows formatting of the entered data. There are four symbols used to format input: a semi-colon (;), a pound sign (#), a percent sign (%), and a question mark (?).

Semi-colon The semi-colon cancels the automatic carriage return which occurs after every value is entered. This is often used in conjunction with descriptive phrases to help an operator enter a correct value. For example, the following program:

```
10 INPUT "VALUE FOR X",X;
20 PRINT "X SQUARE=";X*X
```

would print:

```
VALUE FOR X? 12 (Return to enter value) X SQUARE= 144
```

In this example, the entire output occurs on 1 line. The question mark given as a prompt may be omitted by placing a semi-colon before the variable. In the above example, a semi-colon rather than a comma between "VALUE FOR X" and "X" would have caused the question mark to be omitted.

Pound and Percent Signs The pound and percent signs are used to restrict the number of characters that may be entered for a given variable. The pound sign (#) specifies the maximum number of characters that may be entered. The percent sign (%) specifies the exact number of characters to be entered. The format for each is the same:

```
<line number> INPUT <#number>,<var. 1>,<var. 2>,...,<var. N>
or
<line number> INPUT <%number>,<var. 1>,<var. 2>,...,<var. N>
```

For example, consider the following:

```
10 INPUT #3,A,B
20 INPUT %5,C
```

This example allows up to 3 characters to be entered for A and B and requires that exactly 5 characters be entered for C. The module continues prompting until all 5 characters have been given for C; it will not reset if the carriage return is pressed. Once the 5 characters are entered, BASIC automatically initiates a carriage return and begins execution on the next line of the program.

Question Mark The question mark is used to move execution to specific parts of the program when an invalid character or a control character is entered. To discover whether a control or invalid character has been entered, the system function "SYS(0)" must be read. If "SYS(0)" is equal to −1, then an invalid character has been entered; otherwise, the value found in "SYS(0)" will be the HEX-ASCII code for the control character encountered. The format for the question mark is:

```
<line number> INPUT ? <line number>,<variable>
```

For example:

```
10 INPUT ? 100,N
```

would prompt for a numeric input for N. If a character string is entered (or a control key is pressed), the execution jumps from line 10 to line 100, as in a GOSUB (See 3.4.19) or ERROR statement, where an error subroutine (which must end with a RETURN statement) can be located.

3.4.8 LET Statement

The LET statement assigns a value to a variable. The variable to be assigned is set equal to an expression containing constants and variables separated by operators. The variable to be assigned can appear on both sides of the equation. The format for a LET statement is:

```
<line number> LET <variable> = <expression>
```

The word LET is optional, which means that a simple equation is a valid LET statement. All of the following are valid LET statements:

```
10 LET A = A+10-D
20 Z = X+Y
30 $CAT = 'MOUSE'
```

3.4.9 PRINT Statement

The PRINT statement causes the values of all specified variables to be written to an output device. As with INPUT, the PRINT statement can be left in free format or formatted to create specific output patterns. Free format will first be described and then formatting codes will be discussed.

Free Format. PRINT writes any variable, whether it be numeric, character string, or HEX-ASCII code. Also, anything enclosed in double quotes in a PRINT statement is written as is. HEX characters are separated from the rest of the text by angle braces and may appear anywhere in a character string. For example, if <0A> <0D> is the HEX code for a carriage return and line feed, then the following command:

```
10 PRINT "GO THE NEXT LINE <0A> <0D> AND CONTINUE PRINTING"
```

prints:

```
GO THE NEXT LINE
AND CONTINUE PRINTING
```

NOTE: If you are using the VPU, you should not print the HEX-ASCII codes <12>, <13>, and <14>, because they are used as VPU control characters and will cause problems in the operation of the VPU.

A numeric value is printed by including the numeric variable in the PRINT statement. Strings contained in string variables may also be printed in this way. More than one variable may be included in each PRINT, with each variable separated by a comma or semi-colon. If a comma is used, the values will be placed 5 values to a line and each in a 15-space character field. Semi-colons compress the spacing by inserting one blank space between the values printed. Therefore, it is possible to print more than 5 values to a line using semi-colons. Both semi-colons and commas may be used within the same PRINT statement.

Semi-colons also serve as shorthand substitutes for the word "print" when entering a series of PRINT statements. The semi-colons will be converted to PRINTs when the program is stored in module memory. The following example illustrates the use of semi-colons and commas:

```
10 $NAM="SMITH"
20 S1=95
30 S2=85
40 S3-90
50 S4=(S1+S2+S3)/3
60 ; "SCORES FOR " ;$NAM; " WERE";S1;S2;S3;
70 ; " DIAGNOSTICS FOLLOW:"
80 ;
90 ; "THE HIGH SCORE= ";S1,"AVERAGE= ";S4
95 END
```

This program would print:

```
SCORES FOR SMITH WERE 95 87 90 DIAGNOSTICS FOLLOW:

THE HIGH SCORE= 95 AVERAGE= 90
```

There are several things to note in this example:

- The semi-colon at the end of line 60 suppresses the carriage return and line feed;
- There is one space between all printed words and numbers, except between 95 and AVERAGE. This occurs because a comma is used instead of a semi-colon;
- A semi-colon replaces the word "PRINT" in statements 60, 70, 80, and 90. The semi-colons are converted to "PRINTs" when the program is stored and listed; and,
- PRINT with nothing after it (as in line 80) produces a blank line.

This completes the basic PRINT capabilities provided by the Programmable BASIC Module.

Formatting Codes It is possible to tailor an output to your exact needs by using format codes. Print formatting is essentially done by inserting a pound sign (#) within the PRINT statement along with a HEX formatting character or a decimal formatting string. Since formatting codes do not replace the BASIC PRINT capabilities, commas and semi-colons retain their functions in formatted statements.

For HEX print formatting, there are three forms for the PRINT statement:

```
<line number> PRINT <#;> <expression>
<line number> PRINT <#,> <expression>
<line number> PRINT <#> <expression>
```

If "#;" is used, the expression is converted to a HEX value and printed as a single byte with no preceding or trailing blanks or zeros. Also, the terminating "H" is omitted. If the HEX value is larger than one byte, only the least significant portion is printed.

If "#," is used, the HEX value is printed as two bytes (a full word) with no preceding or trailing blanks or zeros. The terminating "H" is also omitted.

If "#" is used, the HEX value is written in free format. As many spaces as are needed to write the entire HEX value are used. A zero is inserted as the preceding character, and an "H" terminates the value.

The following example illustrates these three forms of HEX formatting:

```
10 PRINT #;10;
20 PRINT #,10;
30 PRINT #10
```

This prints:

```
OA OOOA OAH
```

For decimal formatting, there are two forms for the PRINT statement:

```
<line number> PRINT <#> <"string constant">;<expression>
<line number> PRINT <#> <string variable>;<expression>
```

Each line must be formatted because a specified format does not carry over from one PRINT statement to the next. More than one format specification may be given in one PRINT statement.

The string constant is enclosed in double-quotes and is composed of nines or zeros and one of the seven format characters listed below. A string variable may be used instead of the string constant. It will contain a specific format code similar to that used in the string constant. In effect, it replaces a long format code with a single variable, which may be useful if you have many formatted PRINT statements. If the specified format for a number is too small, a series of asterisks (*) is printed across the field where the number would have gone. To print the number, change the format specification to include the size of the number.

Nines and zeros serve as digit holders in the string constant. Zeros also force a zero to be printed in all non-significant digit positions. Examples of nines and zeros are given in the discussions on the special symbols. The special symbols are summarized in Table 3-1. A period is used to specify the location of the decimal point in a string constant. For example:

```
10 PRINT #"99.0";15.575;128.64
```

would write:

```
15.6 ****
```

That is, 15.575 would be rounded to fit into the "99.0" format specification, and asterisks would be printed for 128.64 because it is too large to fit into the specified format.

^ (carat) A carat translates to a decimal point when printed. The decimal point will always be in the same place in the output as specified in the format string. For example:

```
10 PRINT #"999^00"12056;
20 PRINT #"999^00"200
```

would print:

```
120.56 2.00
```

A number is always printed with a decimal point two digits from the right, regardless of the number entered. The zeroes used in this example hold the digits (as a nine would) and also places zeroes in the right-most two positions if no numbers are specified to go there.

Since the field in the example is specified to exactly five characters, an error results if six characters are entered.

, (comma) A comma is used in a character string to force a comma to be placed in the printed number. The position of the comma in the format string is where it appears in the output. (When there are no digits to the left of the comma, the comma is omitted.)

For example:

```
10 DIM A(3)
10 LET $A(0)="99,999.00"
20 PRINT #$A(0);3529.871
30 PRINT #$A(0);100.3
```

prints:

```
3,529.87
100.30
```

$ (dollar sign) A dollar sign, when used, is included in the printed output. It is also a digit holder like a nine or zero and will float to the position next to the left-most digit or decimal point. It is deleted if the number to be printed is the same size as the specified field. For example:

```
10 PRINT "$$$.00"25.32;
20 PRINT #"$$$.00".50;
30 PRINT #"$$$.00"135.62
```

would write the following:

```
$25.32 $.50135.62
```

Note that the last number is too large for the specified field, which results in the omission of the space and dollar sign.

S (letter S) This symbol is used to print a signed value. A minus sign will be inserted for a negative number, and a blank will be used to indicate a positive number. The "S" is also a digit holder like a nine or zero and will float to the position next to the left-most digit. If the "S" is not used, all numbers are printed as positive values. For example:

```
10 PRINT #"SSS0.00"208.7;
20 PRINT #"SSS0.00"-20.73;
30 PRINT #"9990.00"-36.81
```

prints:

```
208.70 -20.73 36.81
```

E (letter E) This symbol places a minus sign following the right-most digit. It is similar to the "S" above but cannot be used as a digit holder. For example:

```
10 PRINT #"990.00E"32.356;
20 PRINT #"990.00E"-356.9
```

prints:

```
32.36 356.90-
```

<> (braces) Braces are used together to print negative numbers without minus signs appearing in the print-out. Instead, negative numbers are enclosed in angle braces and positive numbers are left as is. The left brace (<) is both a sign holder and a digit holder. The right brace (>) is only a sign holder. The left brace will also float to the position next to the left-most digit. The right brace will always appear after the right-most digit.

For example:

```
10 PRINT #"<<<,<<<.00>"1250.73;
20 PRINT #"<<<,<<<.00>"-2568.9;
30 PRINT #"<<,00>"-0.345
```

prints:

```
1,250.73 <2,568.90> <.35>
```

Table 3-1 Formatting String Characters

Char.	Function	Example
.	Decimal point specifier	PRINT #"999.99"25.32; b 25.32
^	Translates to decimal	PRINT #'999 ^ 00"1000; b10.00
,	Suppressed if before significant digit	PRINT #"99,999.99"100; bbb100
9	Digit holder	PRINT #"9999"123; b123
0	Digit holder or forces zero	PRINT #"9990.99".23; bbb0.23
$	Digit holder & floats $	PRINT #"$$$.99"8; b$8.00
S	Digit holder & floats sign	PRINT #SSS.99" −6; b−6.00
E	Sign holder after decimal	PRINT #990.99E" −150.75;150.75−
<	Digit holder before decimal & floats on negative number	PRINT #"<<<.00>" 500;500.00
>	Appears after decimal if negative	PRINT #"<<<.00>" −50; 50.00

Note: b indicates blank.

3.4.10 RANDOM Statement

The RANDOM statement plants the seed for pseudo-random number generation. It is used in conjunction with RND function. The seed may be set to either a constant value or an arbitrary value. The format for this statement is:

```
<line number> RANDOM <expression>
```

"Expression" may be any valid BASIC expression, provided it evaluates to a number between −32767 and 32767. If no expression is given, 0 is the default value. A sequence of numbers between 0 and 1 is generated by RND because of the constant seed. The sequence of numbers restarts every time a RUN command is entered.

3.4.11 REM Statement

The REM statement is used to insert remarks into a program. The line containing a REM is ignored during execution and is only seen when the program is listed. Because of this, REM should not be used as part of multi-statement line. If a comment is needed after a statement, use a tail-remark (!). REM statements, as well as tail-remarks, should be concise since they affect the size and rate of execution of a program. The format for a REM statement is:

```
<line number> REM <comment>
```

For example:

```
100 REM Read values for temperature
```

3.4.12 STOP/END Statements

Both STOP and END terminate the execution of a program. Both are also required to have line numbers. STOP is used at all logical termination points in the program, and END occurs only once as the last statement of the program. Since STOP is logical in use, a CONTINUE can be used to restart the program from the point at which a STOP appears. This is not possible with an END statement. The line number of the STOP or END statement that causes the program to cease executing is sent to the device connected to port 1 of the module.

3.4.13 TAB Statement

The TAB statement is used to advance to a particular column on a line and begin printing there. The columns allowed range from 0 to 127. (If your device has lines shorter than 127 characters, specifying a column beyond the end of the line causes BASIC to wraparound to the next line. For example, if your line is 80 characters long, specifying column 100 causes BASIC to begin printing in the 20th column of the next line.) The format is:

```
<line number> PRINT TAB(<column number>);<expression>
```

The column number must be evaluated to be an integer value that specifies the horizontal column position when printing begins. For example, to print the word "field" starting in the 20th column, you would use the following statement:

```
20 PRINT TAB(20); "field"
```

There is no limit to the number of TAB statements in a line except for the line length itself. The TAB settings will reset on a carriage return. Also, TAB cannot be used to go to a column that has already been passed.

3.4.14 TIME Statement

The TIME statement is used to set, display or store the 24-hour clock. It also keeps track of the day, month, and year. TIME may be read or initialized anywhere in the program. TIME is intended to be used to keep track of long periods of time; for short periods of time, use the TIC function. There are three formats for the TIME statement:

Format 1 <line number> TIME <YR>,<MO>,<DY>,<HR>,<MN>

This form is used to initialize the internal clock to a specific time. The time is entered in the order of year, month, day, hour (based on 24-hour time), and minute. (You do not need to enter the number of seconds because this will be automatically set to zero when the statement is executed. Also, any values omitted will be interpreted as zeros upon execution.) For example, to set the clock to December 20, 1986, 2:10 pm, you would enter:

```
10 TIME 86,12,20,14,10 ! Time is set to Dec. 20, 1986
20 REM 2:10 pm each time line is encountered
```

To set the time once, verify that the battery is active. Then type in TIME in the command mode (without line number) following the format outlined above.

Format 2 <line number> TIME

This statement reads the time currently in the BASIC clock. It prints the time in the order of year, month, day, hour, minute, and second. For example:

```
100 TIME
```

could result in this being printed:

```
86:12:21:14:15:30
```

This is interpreted as 15 minutes, 30 seconds, past 2 pm on December 21, 1986.

Format 3 <line number> TIME <string variable>

This form of the TIME statement places the current time into the designated string variable. The current time is formatted as in (1) and (2). For example:

```
10 DIM TIM(2) ! Dimension to 3 elements
.......
100 TIME $TIM(0) ! Loads the TIME into $TIM(0)
```

places the time (YR:MO:DY:HR:MN:SC) in $TIM(0). Note that any array used for storing the time must be dimensioned to 3 and that the colons are part of the time value as stored in memory. Both colons and numbers are stored, rather than just numbers.

3.4.15 UNIT Statement

The UNIT statement is used to choose which port will send or receive data. The format for this statement is:
<line number> UNIT <number>

The number may be 0, 1, 2 or 3. It may also be an expression which evaluates to one of these three numbers. Each number signifies which ports will be used for inputs or outputs. The following chart shows what each number means:

Number	Action
0	BASIC uses neither port for input or output
1	BASIC uses port 1 for input or output
2	BASIC uses port 2 for input or output
3	BASIC uses ports 1 and 2 for output, and port 1 for input

The default value for UNIT is 1 (inputs are sought at port 1 and outputs are directed to port 1). Note that if UNIT 0 or UNIT 2 is chosen, the Programmable BASIC Module will still accept commands from port 1 but will not echo the keystrokes back to the programming device. (Remember that programming is allowed only through port 1.)

NOTE: Pressing the $\boxed{\text{Esc}}$ key does not reset the UNIT statement. Thus, if you press $\boxed{\text{Esc}}$ while UNIT 0 or UNIT 2 is in effect, keystrokes after ESCAPE are not shown. To see the keystrokes on the device, type UNIT 1 and press $\boxed{\text{Enter}}$ (the keystrokes are not shown on the device connected to port 1).

3.4.16 Internal Data Statements

BASIC allows a data list to be included in a program. The statements READ, DATA, and RESTOR are used to place and obtain the data at different points in the program. All three statements must have line numbers. Each statement is described below:

READ The READ statement assigns values from the internal list to a variable or array element. READ gets the data by sequentially reading the internal data as requested by the program. If a READ statement is used and the data list has been read through to the end, an error results.

RESTOR The RESTOR statement is used to move the read pointer either to a specific statement number in the list of data statements or to the first data statement in the program. If a number (or an expression which evaluates to a number) is entered after RESTOR, that number must be a line number in the program (or an error results). The line number entered is the line number of the DATA statement that the next READ statement uses. (If the given number is not a DATA statement, the next DATA statement in the program after the given line number is used.) If no line number is given, the next READ statement begins at the first position of the first DATA statement used in the program.

DATA The DATA statement contains the list of internal data. Values are separated by commas, and all string variables are enclosed in quotes. DATA statements may appear anywhere in a program except as part of a multi-statement line. DATA statements usually appear as a group near the beginning or end of a program.

The formats for these three statements are:

```
<line number> READ <variables>
<line number> RESTOR <optional line number>
<line number> DATA <data>
```

The following example shows how these three statements work:

```
10 READ A,B,C,D ! Gets the first 4 values from 80
20 H = A*B*C*D
30 READ E,F,G ! Gets the 5th, 6th, and 7th values from 80
40 I = E*F*G
50 RESTOR ! Resets data pointer to first data element 80
60 READ X,Y,Z ! This obtains the first three values in 80
70 PRINT A;B;C;H;I;X;Y;Z
80 DATA 1,2,3,4,5,10,20
90 END
```

The output from this program would be:

```
1       2       3       24       1000    1       2       3
```

In this example, note how READ moved sequentially through the data string and did not return to the beginning of the string for the second READ statement. The RESTOR statement reset the data pointer to the beginning of the data list, which resulted in A,B,C being the same as X,Y,Z. If the RESTOR had not been used, an error would have occurred when the third READ statement was used.

3.4.17 Branch Statements

There are a variety of ways to transfer execution to different parts of a program, or to prevent the transfer of execution. IF-THEN-ELSE, GOTO, and ON are the statements used in BASIC to alter the sequential flow of a program. These statements are explained in the following paragraphs.

IF-THEN-ELSE This statement is used to branch a program in two directions depending on a conditional statement. The two branches are the "IF-THEN," which appears on one statement line and contains the conditional statement, and the "ELSE," which appears on a line following the IF-THEN and initiates the second branch.

The function of the IF-THEN statement is to compare two expressions and arrive at a true or false result. If the statement is true, the expression following the THEN is executed. If the statement is false, execution skips the rest of the line and passes to the line immediately following the IF-THEN line. The format for the IF-THEN is:

```
<line number> IF <expression> THEN < BASIC statements>
```

"Expression" may be any variable along with logical and relational operators. For example, all of the following are valid IF-THEN statements:

```
20 IF A=0 THEN GOTO 100
30 IF SQR(J)=4 THEN K=J*J/I::PRINT J,K
35 REM J & K printed when "IF" is true
40 IF BIT(A,1) THEN BIT (B,1)=1
50 IF X<6 OR X>2 THEN PRINT "IN RANGE"
```

The ELSE statement is placed after an IF-THEN statement and is executed when the IF-THEN is false. ELSE should appear on a line by itself; it cannot be used as part of a multi-statement line. If the IF-THEN is true, the ELSE statement is ignored. More than one ELSE may follow an IF-THEN statement. Each ELSE is executed when the IF-THEN is false. The format for an ELSE is:

```
<line number> ELSE <BASIC Statements>
```

The following short program illustrates an IF-THEN-ELSE:

```
100 IF A=0 THEN PRINT "DATA EXHAUSTED"
110 ELSE PRINT "MORE DATA TO RUN"
```

These two statements give information to an operator depending on the status of the data to be run. When A is 0, the entire IF-THEN statement of line 100 will be executed. The ELSE statement of line 110 will be ignored. For all other values of A, only the IF part of line 100 will be read, and then execution will pass to line 110.

GOTO The GOTO statement transfers execution to a specified line of the program. Execution then proceeds sequentially from the specified line. As shown below, there is no embedded blank between GO and TO:

```
<line number> GOTO <line number>
```

For example:

```
10 GOTO 500
```

skips all statements between 10 and 500 and resumes execution at line 500.

ON The ON statement specifies a variable that triggers a GOTO or GOSUB statement. It cannot be used as part of a multi-statement line, nor can it be followed by a tail-remark (!). The format for an ON is:

```
<line number> ON <variable> then GOSUB <line number>
or
<line number> on <variable> then GOTO <line number>
```

The "variable" may also be an expression, in which case the expression would be evaluated, and then, depending on the answer, execution would be transferred elsewhere in the program by the GOTO or GOSUB. It is possible to indicate more than one location for the GOTO or GOSUB transfer. The value that arises from the variable or expression is used as a counter to find a specific number in the list of line numbers after the GOTO or GOSUB statement. If the variable indicates a number greater than the number of available line numbers, then the ON statement will be ignored.

For example, the following ON statement:

```
10 ON J THEN GOTO 15,20,35
```

would do the following:

When J=0, the statement would be ignored;
When J=1, execution would go to line number 15;
When J=2, execution would go to line number 20;
When J=3, execution would go to line number 35; and,
When J=4, the statement would be ignored.

3.4.18 CALL (PLC Interface Statement)

CALL is the statement used to invoke subroutines that obtain information from, or send information to, the PLC. There are five of these subroutines available in BASIC: IOREAD, IOWRITE, PCREAD, PCWRITE, and SRTC. Note that in all of these subroutines the dollar sign is omitted for character strings. For instance, when invoking PCREAD, a character string would be included without the usual prefix of a dollar sign. These subroutines (and the DIM and FIND statements) are the only elements in BASIC where the dollar sign is omitted. All other references to a character array must include a dollar sign.

NOTE: When using PCREAD, PCWRITE, or SRTC, be aware that PLC scan time will be slightly affected. The increase in scan time is not more than 2 ms, which is negligible in most applications. This can be avoided by implementing the lock-out bit as described under IOREAD/IOWRITE.

IOREAD/IOWRITE These two subroutines transfer data directly from or to the I/O registers on the module (data type must be integer). The PLC input points are numbered 1 to 4, and the output points are 5 through 8. In addition, a complete I/O address will have the particular channel, base, and slot that the BASIC module occupies. For example, a module inserted in the first slot of base 0 would have the following input and output points:

I/O WRITE				I/O READ			
1	2	3	4	5	6	7	8
WX9	WX10	WX11	WX12	WY13	WY14	WY15	WY16

The format for both subroutines is the same:

```
<line number> "IOWRITE",<number>,<array>,<number>
or
<line number> "IOREAD",<number>,<array>,<number>
```

The first "number" before the array is the input or output point to be used; the "array" is the location to which or from which the data is to be transferred; and the final number after the array is the number of consecutive words to be transferred.

```
10 CALL "IOREAD",5,DAT(0),4
20 CALL "IOWRITE",1,DAT(0),4
```

Statement 10 transfers 4 words to $DAT(0) from output registers 5 through 8 and sends 4 words from $DAT(0) to input points 1 through 4 in statement 20.

Be sure that the array in the statement is dimensioned large enough to accept the words being sent.

The following program shows how IOREAD and IOWRITE subroutines may be used in a program:

```
10 REM Dimension data arrays to at least size of
20 REM points to be transferred.
30 DIM DAT(3) ! Dimension the array for storing values to 3
40 DIM VAL(3) !Dimension the array from which data is sent
50 REM Clear data array before reading
60 DAT(0)=0
70 DAT(1)=0
80 DAT(2)=0
90 DAT(3)=0
100 CALL "IOREAD",5,DAT(0),4 ! Obtains data from PLC
110 REM Enter values to be sent in array
120 VAL(0)=100
130 VAL(1)=200
140 VAL(2)=300
150 VAL(3)=400
160 CALL "IOWRITE",1,VAL(0),4 ! Sends values to PLC
170 END
```

After this program is executed, the array DAT(0) contains the values present in the output points 5 through 8, and the PLC will have the values 100, 200, 300, and 400 for the input points 1 through 4.

Lock-Out Bit Communications between the Programmable BASIC Module and the PLC employing the PCREAD, PCWRITE, or SRTC statements extend the scan time of the PLC by allowing this communications window. If it is necessary to prevent this scan extension, the module must be prevented from requesting PCREAD, PCWRITE or SRTC communications. This can be done by assigning a "pseudo" lock-out bit in the normal I/O update communication allowed to the module. This involves setting a bit in a WY word of the module.

The module operating system does not recognize a lock-out bit. The bit, however, must be mutually understood by the module and the PLC RLL program. Simply setting the selected bit does not automatically impede PCREAD, PCWRITE, or SRTC communications. A subroutine in the BASIC module program must be executed prior to any extended scan communications to determine if the bit has been set.

For example, a Programmable BASIC Module resides in slot 1 of base 1, occupying WX1, WX2, WX3, WX4, WY5, WY6, WY7, WY8 in "normal" I/O. The least significant bit of WY8 has been selected to be the lock-out bit. Therefore, if the PLC RLL program loads a 1 into WY8, the scan extension communications should not be allowed by the BASIC program.

NOTE: The WY location used as the lock-out bit is shared by the module and the PLC. The PLC recognizes the bit as the least significant bit (bit 16), while the module recognizes it as the least significant bit of an integer format (bit 32).

The following program section should then appear in the Programmable BASIC Module program:

```
200 GOSUB 520 ! Check to see if lock-out bit is set
205 IF FLG=1 THEN GOTO 220 ! Lock-out bit set, skip PLC
    comm
210 CALL "PCREAD",ADR(1),DAT(1)4 ! Scan extending comm
220 REM Continue without obtaining data from PLC
.
.
520 REM check for lock-out bit
525 FLG=0
530 CALL "IOREAD",8,LOK,1 ! LSB of WY8 is lock-out bit
540 IF BIT(LOK,32)=1 THEN FLG=1 ! 32 is least significant bit
550 RETURN ! Bit is not set, communications is allowed
```

NOTE: When using a bit as a lock-out bit, it is critical that the bit is not altered by any data passed through the word. For this reason, it might be advantageous to reserve the word for use as only a lock-out bit, should a lock-out bit be useful in your application.

PCREAD/PCWRITE These two subroutines transfer data from or to a variable in the PLC. Variables in the PLC may be any of the following:

Variable		Address Example
V	variable	$ADR = "V100"
WX	analog word input	$ADR = "WX6"
WY	analog word output	$ADR = "WY14"
X	discrete input	$ADR = "X440"
Y	discrete output	$ADR = "Y3"
C	internal control relay	$ADR = "C210"
TCC	timer/counter current value	$ADR = "TCC6"
TCP	timer/counter preset value	$ADR = "TCP18"
DSC	drum step current value	$ADR = "DSC127"
DCC	drum count current value	$ADR = "DCC12"
DCP	drum count preset value	$ADR = "DCP1201"

DCC returns two words. First, the step drum is on 0 to 15; second, the current value of the step. The current count counts down from preset.

DCP is applicable to event drums only and requires additional addressing to write to it. DCP12 refers to the drum number; 01 is the step in the drum to be looked at or altered.

NOTE: If your module is operating in a TI560™/TI565™ PLC system, there are some limitations on accessing memory. The following upper limits are placed on the data accessible from the PLC through the PCREAD and PCWRITE statements.

- V memory V65535
- K memory .... none accessible
- DCC memory DCC127
- S memory .... none accessible
- DCP memory D127S16

Once the variable name has been determined and correctly formatted, the format for both subroutines is as follows:

```
<line number> CALL"PCWRITE",<variablename>,<array>,<number>
<line number> CALL"PCREAD",<variablename>,<array>,<number>
```

The "variable name" is the string variable from which data is to be obtained or to which data is to be written from the PLC. The "array" is where the data will be placed in the BASIC memory or where the data to be sent to the PLC is located. "Number" is how many consecutive words or data points are to be transferred. The PLC addresses, as well as the BASIC variables, must be sequential to be accessed in a single scan. Consider the following example.

```
05 $ADR(0) = "V200"
10 CALL "PCREAD",ADR(0),DAT(0),14
20 CALL "PCWRITE",ADR(0),DAT(0),14
```

The first statement gets 14 sequential points starting at the PLC variable represented by $ADR(0) and puts them in DAT(0) in the BASIC memory. Statement 20 takes the 14 sequential points and puts them sequentially in the PLC memory starting at V200.

For the subroutines to execute properly, the following conditions must exist:

- The variable name must be dimensioned to at least 2;
- The variable name must be assigned to a valid PLC address;
- The array must be dimensioned correctly for the incoming data.

The following example shows a program containing PCREAD and PCWRITE subroutines:

```
5 REM Dimension data arrays to at least size of
10 REM points to be transferred.
15 DIM ADR(2) ! Dimension address array to 2
20 DIM DAT(13) ! Dimension data array
25 DIM SET(1) ! Dimension write array
30 DIM VAL(13) ! Dimension data array
35 LET $ADR(0)="V100"
40 LET $SET(0)="C64"
45 REM Clear out entire data array before reading values
50 FOR I=0 TO 13
60 DAT(I)=0
70 NEXT I
80 CALL "PCREAD",ADR(0),DAT(0),10 ! Gets values from PLC
90 FOR I=0 TO 9
100 PRINT DAT(I) ! Prints the values obtained from the PLC
110 NEXT I
120 REM Enter values to be written into data array
130 VAL(0)=0 ! Set C64 to off
140 VAL(1)=1 ! Set C65 to on
150 CALL "PCWRITE",SET(0),VAL(0),2
160 END
```

If PLC addresses V100 through V109 contain the values 100,200,300,..., 1000, the output will be these 10 values at line 100. Also C64 will be off and C65 will be on after the execution of the program.

NOTE: Care should be taken when using PCWRITE to send data to word memory areas (WX or WY). Unlike discrete memory points (X, Y, or C), word memory areas can be overwritten even if they are forced.

The PCREAD and PCWRITE subroutines permit up to a maximum number of consecutive data points to be transferred between the Programmable BASIC Module and the PLC during a single PLC scan. The maximum number of data points transferable follows:

Type of Data Point	Consecutive Points Per Scan	
	PCREAD	PCWRITE
DISCRETE: C, X or Y	26	18
Word: V, WX, WY, TCC TCP, DSP, DCP, DSC	30	28
DCC	20 values (40 words) per scan	N/A

The above values are valid only if the PLC is in the RUN mode.

Only one PCREAD or PCWRITE subroutine can be called per scan, regardless of the amount of data to be transferred.

If a call to PCREAD or PCWRITE is made requesting more data than can be transferred in a single scan, the remaining data will be transferred on subsequent scans.

SRTC The SRTC (Send and Receive Task Codes) subroutine allows a BASIC program to send operational codes (rather than values) to the PLC. The codes may be sent only to the PLC; they cannot be sent through a Network Interface Module to the TIWAY network. SRTC sends specific task codes to the PLC in a HEX-ASCII format. The HEX-ASCII format (A–F, 0–9) must be enclosed in a variable that will be used in the CALL statement. Byte count and message delimiters will be added by the subroutine. The format is:

```
<line number> CALL "SRTC",<string variable>,<string variable>
```

The first "string variable" contains the HEX-ASCII code to be read, and the second string variable is where the task code response is to be written. For example, the following:

```
100 CALL "SRTC",TCC(0),TCR(0)
```

takes the task code in $TCC(0) and places the response to the code in $TCR(0).

To ensure that no values are overwritten, both string variables must be properly dimensioned. Since 61 characters is the maximum possible number for a HEX code, a dimension of 10 prevents values from being overwritten.

Please see your Siemens Industrial Automation, Inc. distributor for further information on these task codes and this subroutine.

⚠ WARNING

Use of SRTC can cause unpredicable machine behavior. Because it can change the PLC program, SRTC has the potential to endanger personnel and equipment if used incorrectly. To avoid the risk of personal injury or damage to equipment, be sure you understand how SRTC works. Some task code requests may cause the module to time out before the PLC can respond.

3.4.19 Subroutine Statements

GOSUB, POP, and RETURN are used for program subroutines in BASIC. There are also subroutines that are used for errors (see 3.4.3) and statements that simply transfer execution (see the GOTO and ON statements in 3.4.17).

GOSUB GOSUB is used for entry into a subroutine. It has the following form:

```
<line number> GOSUB <line number>
```

"Expression" is either a number or a BASIC expression which evaluates to a number. The number is the line number to which execution passes.

When a GOSUB statement is reached, the next statement after the GOSUB is placed in a wait stack, and execution passes to the line number specified in the GOSUB statement.

RETURN A RETURN statement ends the subroutine and returns execution to the statement at the top of the wait stack. All subroutines must end with a RETURN, and a RETURN should not appear without being coupled to a GOSUB or ERROR statement.

The following is a simple example of these two commands.

```
10 X = 2
20 GOSUB 90
30 PRINT X;Z
40 END
...........
90 Z = 2*X-1
100 X = X*Z
110 RETURN
```

results in:

```
6 3
```

POP Subroutines may be nested up to 20 deep. Sometimes the wait stack can become filled or you may wish to escape the current subroutine and go back to the last routine. For such cases, use a POP statement to remove the top line number in the wait stack. POP will not return to the removed statement; it only removes the line number at the top of the stack, and then execution passes to the statement after POP. Incorrect use of a POP statement could result in a stack overflow error.

NOTE: When using GOSUB statements, terminate the main program with an END statement. Failure to do so may result in a stack overflow error.

3.5 BASIC Functions

BASIC supports many pre-defined mathematical and string functions. These functions are of the form:

```
<line number>....<function name(argument)>....
```

The "function name" is a 3-digit letter designation for a particular function. "Argument" is the variable, string, or expression on which the function will operate. The dots indicate that functions may be used in conjunction with any other BASIC expressions. In the following sections, each of the functions used in BASIC is described.

3.5.1 ABS Function

The ABS (absolute value) function operates only on negative numbers. These numbers are converted to positive values with ABS (positive values are not affected).

```
20 PRINT SQR(ABS(X))
```

3.5.2 ASC Function

The ASC (ASCII conversion) function obtains the decimal ASCII value of the first character of a particular string. ASC is the inverse operation of the byte replacement operator (%). Note that, if you use the byte replacement operator, you must terminate the characters being replaced by putting a null at the end as shown in the following program:

```
10 $I="B" ! Assign value to $I
20 J=ASC($I) ! J = decimal ASCII value of $I
30 K=J+020H ! Increment value of J by 20H; B+20H=b
40 $L=%K%00 ! Return decimal ASCII to alphanumeric rep.
50 M=ASC($L) ! M = decimal ASCII value of $L
60 PRINT $I;J;$L;M
70 STOP
```

This program prints:

```
B 66 b 98
```

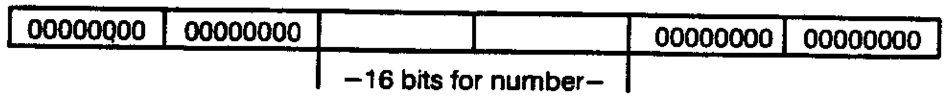
3.5.3 ATN Function

The ATN (arctangent) function returns the value in radians for a ratio representing the tangent. (Radians may be converted to degrees by multiplying the value in radians by 180/Pi.)

3.5.4 BIT Function

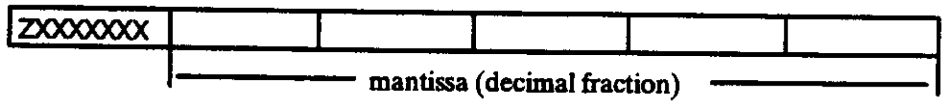
The BIT (bit modification) function is used to read or modify the bits of a variable. If you wish to alter bytes or words, use the MEM or MWD functions. Each variable is composed of 6 bytes or 48 bits of memory. With 48 bits, accuracy to roughly 11 significant digits and numbers between +/−1E74 and +/−1E−74 may be expressed. The internal representation of these bits is in either integer or floating point (real) format.

With each box below containing 8 bits, integer format is represented as follows:



The binary string is interpreted by converting the string to a decimal number.

With each box below containing 8 bits, floating point format is represented as follows:



Z = bit for sign (1 = negative, 0 = positive)
Xs = exponent (Excess-64 notation)

An example for floating point format is the following:

01000010 01010000 00000000 00000000 00000000 00000000

To interpret this number, start with the exponent. "1000010" is part of the first 8 bits which is in "Excess-64 notation." This notation is used to express both positive and negative exponents of 16. To derive the exponent, you would interpret the 7 bits to a decimal number and subtract 64. For the example, "1000010" is equivalent to decimal 66, which, when 64 is subtracted from it, yields an exponent of 2. The exponent is applied to 16, which is equal to 256.

After the exponent is derived, the mantissa is interpreted. The mantissa is a fraction; it contains the value to the right of the decimal point. Each bit represents 2 raised to a negative power. For the example, the mantissa "01010000" is interpreted as 2^−2+2^−4, which is equal to 0.3125.

The final result is obtained by multiplying the interpreted mantissa by the interpreted exponent. In this example, the final result is 256 multiplied by 0.3125, which equals 80.

After the number is derived, you would obtain the sign of the number by checking the first position of the binary string. For the example, this is 0, which means that the number is positive, and 80 is the decimal equivalent of the binary string.

The structure of arrays is the same as above with each position in the array taking up 6 bytes of memory . In other words, each position of an array is arranged at the bit level in either the floating point or integer format. For example, the following is a representation of an array dimensioned to (2,1) starting at HEX address C200H:

Position		Address
(0,0)	6 Bytes:	C200, C202, C204
(0,1)	6 Bytes:	C206, C208, C20A
(1,0)	6 Bytes:	C20C, C20E, C210
(1,1)	6 Bytes:	C212, C214, C216
(2,0)	6 Bytes:	C218, C21A, C21C
(2,1)	6 Bytes:	C21E, C220, C222

Each byte of the array contains one ASCII character, which translates to 6 characters per array position. The sixth character is most often a null, which is used to indicate the end of a word. For example, if the word "dog" were stored in position (0,0) of the above array, the address would look like this:

D	O	G	00000000	XXXXXXXX	XXXXXXXX
—— C200H ——		—— C202H ——		—— C204H ——	

BIT has the following format:

```
BIT(<var>,<position>)
```

Bit positions range from 1 (most significant) to 48 (least significant). Recall integer values are stored in bits 17 through 32. If you use a number greater than 48, the next memory address will be read or modified. For example, the following program reads the variable "A" and writes its bit representation:

```
10 INPUT A,
20 FOR I=1 TO 48
30 PRINT BIT (A,I);
40 IF I=1 THEN PRINT "–";
50 IF I=8 THEN PRINT " ".
60 IF I=16 THEN PRINT " ";
65 IF I=24 THEN PRINT " ";
70 IF I=32 THEN PRINT " ";
80 IF I=40 THEN PRINT " ";
90 NEXT I
100 GOTO 10
```

If this were executed, it would prompt for a value (1) and print the following:

```
?1 0–00000000 00000000 00000000 00000001 00000000 00000000
?1.0–10000001 00010000 00000000 00000000 00000000 00000000
```

Note the differences in representation for the integer value 1 and the floating point value 1. BIT can also be used to change the structure of an address. For example, if you wanted to change a number from positive to negative, BIT would be used to change the first position of the address from a 0 to a 1. The following program shows this change:

```
10 A=1.23
20 BIT(A,1)=1 ! Changes the sign of the number
30 PRINT A
```

If this were executed, this would be the output:

```
-1.23
```

⚠ WARNING

Be careful when using the BIT function. If you rewrite part of the module memory rather than variable memory, the module may perform unpredictably. To avoid personal injury or damage to equipment, before restoring power to the module: 1) Ensure that all personnel are clear of machinery areas controlled by the module application program. 2) Ensure that all equipment is prepared for start-up operation.

3.5.5 COS Function

The COS (cosine) function determines the cosine value of an angle entered in radians. The angle cannot be entered in degrees. (To obtain radians from degrees, you must multiply the value in degrees by Pi/180.)

3.5.6 EXP Function

The EXP (exponential) function is used to raise the value of "e" (the base of natural logarithms) to the power indicated by the argument. The argument is the exponent of "e." The argument used with "e" should be 87 or less. For example, the following would raise "e" to the fifth power:

```
10 Z=5
20 PRINT EXP(Z)
```

3.5.7 INP Function

The INP (integer part) function returns the signed integer portion of the entered value. The value is truncated rather than rounded when converted to integer form. The entered value should not exceed 6.87E10 or erroneous answers result. For example, the following program:

```
10 INPUT X,Y,Z
20 PRINT INP(X);INP(Y):INP(Z)
30 END
```

prompts for values for X, Y, and Z:

```
? 1.145 −2.53 5
```

and prints:

```
1 −2 5
```

3.5.8 LEN Function

The LEN (length) function is used with character strings to find the number of non-null characters in the string. 0 is returned if no characters are found in a string. For example:

```
10 $I="ABC"
20 PRINT LEN($I);
30 PRINT LEN ("DEF G")
40 END
```

prints:

```
3 5
```

3.5.9 LOG Function

The LOG (logarithm) function is the inverse of the EXP function: The value entered is the number for which the natural logarithm (e) is required. The entered value must be positive; a negative number results in an error. For example, the following program:

```
10 L=5280
20 PRINT LOG(L)
30 STOP
```

prints:

```
8.57168
```

3.5.10 MCH Function

The MCH (match) function is used only with character strings. It compares two character strings and returns a value for the number of characters in exact agreement between the two strings. Zero will result if no characters are in agreement. For example, the following program:

```
10 $C="ABCD"
20 M=MCH("ABF",$C)
30 PRINT M
40 END
```

prints:

```
2
```

3.5.11 MEM Function

After using the FIND statement to locate a variable address in BASIC memory, the MEM (byte modification) function may be used to read or modify the data at that memory address. It reads in bytes: if you wish to read in bits or words, refer to the BIT function or the MWD function for instructions on reading other lengths of addresses. The argument which follows MEM is the address to be read or altered. For example, the following:

```
M=MEM(0C200H)
```

places the byte in address 0C200H into M, while:

```
MEM(0C200H)=07FH
```

replaces the byte at address 0C200H with 07FH (or 127 in decimal form).

⚠ WARNING

Be careful when using the MEM function. If you rewrite part of the module memory rather than variable memory, the module may perform unpredictably. To avoid personal injury or damage to equipment, before restoring power to the module: 1) Ensure that all personnel are clear of machinery areas controlled by the module application program. 2) Ensure that all equipment is prepared for start-up operation.

3.5.12 MWD Function

After using the FIND statement to locate a variable address in BASIC memory, the MWD (word modification) function can be used to modify or read a word (2 bytes) of data at that memory address.

This function is similar to the MEM function, except MWD works on a word rather than just one byte.

The argument which follows MWD is the address that will be read or modified. Note that, since all variables are stored starting at even address boundaries, the argument for MWD must be even. For example, the following:

```
M=MWD(0C200H)
```

reads the word starting at address 0C200H and places it in M. The following:

```
MWD(0C200H)=07FFFH
```

replaces the word starting at address 0C200H with 07FFFH (or 32,767 in decimal form).

⚠ WARNING

Be careful when using the MWD function. If you rewrite part of the module memory rather than variable memory, the module may perform unpredictably. To avoid personal injury or damage to equipment, before restoring power to the module: 1) Ensure that all personnel are clear of machinery areas controlled by the module application program. 2) Ensure that all equipment is prepared for start-up operation.

3.5.13 NKY Function

The NKY (key) function reads characters out of the input buffer of either port without interrupting the execution of the program. This is useful in keyboard polling applications where characters need to be entered without halting program execution such as in interaction with bar code readers. The format for NKY is the following:

```
NKY (<number>)
```

NKY reads a character from the port designated as input by the UNIT statement. If the number used with NKY is 0, the number read from the input buffer is returned. If no number was in the buffer, a 0 is returned. If the value given with NKY is not 0, the HEX equivalent of the given number is compared with the HEX equivalent of the number in the input buffer. If the two agree, a 1 is returned; if they do not match, a 0 is returned.

The following program illustrates the use of the NKY function:

```
2010 REM Keyboard polling: Echo keystroke
2020 N=NKY(0)
2030 IF N=0 THEN RETURN
2040 $N1=%N%00 ! Convert to ASCII
2050 PRINT $N1; ! Print the last keystroke
2070 GOTO 2020

3080 REM polling for a specific input
3090 N=NKY(41) ! A is the target value
4000 IF N=0 THEN RETURN ! A is not found
4010 ELSE PRINT "A WAS INPUT"
4020 GOTO 2090
```

The NKY function is often used with the % operator. This operator converts the HEX or decimal value of a character into the alphanumeric representation.

```
10 LET A=65 ! Decimal A
20 A1=%A%0 ! Convert
30 PRINT A1

RUN

A
```

3.5.14 RND Function

The RND (random) function is used to generate a pseudo-random number between 0 and 1. It is used in conjunction with the RANDOM statement. RANDOM specifies the string of numbers to be read, and RND sequentially steps through these numbers. This sequence of 32,767 numbers is repeated until the seed value of RANDOM is changed.

3.5.15 SIN Function

The SIN (sine) function determines the sine value of an angle entered in radians. The entered value cannot be in degrees (radians may be obtained by multiplying the value in degrees by Pi/180).

3.5.16 SRH Function

The SRH (search) function is used with character strings. It finds the location of one string within a second string. If the search is unsuccessful, 0 will be returned. For example:

```
10 $C="ABCDEFG"
20 S=SRH("BCD",$C)
30 PRINT S
40 END
```

prints:

```
2 (the location where the sought-after string begins)
```

3.5.17 SQR Function

The SQR (square root) function returns the square root of the specified value. The specified value must be positive or an error will result.

3.5.18 SYS Function

The SYS (system interrogation) function reads 13 system parameters. The parameters are read by enclosing a number from 0 to 12 in parentheses after SYS. The 13 parameters and their associated SYS functions are explained below:

SYS(0) This function works in conjunction with the question mark option of formatted INPUT statements. It stores a value of −1 if an invalid character is entered or the HEX representation of the control character encountered.

SYS(1) This function works in conjunction with the ERROR statement. It contains the error code number for the error encountered. See Appendix D.

SYS(2) This function works with the ERROR statement. It stores the line number in which an error occurred.

SYS(3) This function reads the task code error after an error 63 (PLC task code communication error) is encountered. If you have questions, contact your Siemens distributor.

SYS(4) This function will store a 1 if the battery is in place and capable of backing up the internal clock and memory. If the battery is weak or missing, a 0 will be stored.

SYS(5) This function reads the number of free bytes remaining in the print buffer memory of port 1.

SYS(6) This function reads the number of free bytes remaining in the print buffer memory of port 2.

SYS(7) This function reads the number of free bytes remaining in the input buffer memory of port 1.

SYS(8) This function reads the number of free bytes remaining in the input buffer memory of port 2.

SYS(9) This function reads a 1 if port 1 is clear to send is active, or a 0 otherwise.

SYS(10) This function is the same as SYS(9) except it reads the clear to send status for port 2.

SYS(11) This function keeps track of receiver errors (parity, overrun, framing) at port 1 since the last power-up or self-test.

SYS(12) This function keeps track of receiver errors at port 2 since the last power-up or self-test.

The RS232C/423 receivers are always enabled. Pins 5 and 6 must be active to enable the transmitter. The status of pin 5 may be checked with SYS(9) or SYS(10).

NOTE: If you are missing characters while using the 19.2k baud rate (as indicated by SYS(11) or SYS(12)), reduce the baud rate or pad the data to reduce the effective data transfer to 9600 baud or less.

Delays in output may be avoided by first checking SYS(5) or SYS(6) (depending on which port is used for output) for space remaining in the buffer. If the buffer is full, requests to print may be set aside through a program subroutine until the buffer empties.

If the input buffer becomes full, input characters will be ignored until there is room in the buffer for more characters. You may determine how much room is in each buffer by reading SYS(9) or SYS(10), depending on which port is being used for input. Under certain conditions, input characters may be missed when the baud rate is set to the maximum rate (19.2K bps). Depending on which port is being used for output, SYS(11) or SYS(12) keeps track of transmission errors.

3.5.19 TIC Function

The TIC (delta time) function is used to obtain the length of time between two events. One TIC is equal to 40 milliseconds. The resulting value is the difference between the time at sampling and the expressed value in the TIC function (to obtain seconds from the TIC value, divide by 25). For example:

```
T=TIC (0)
```

places the current TIC count in T, and

```
D=TIC(T)
```

calculates the elapsed time since the last count was stored in variable T. Multiple and overlapping TICs are allowed in a program.

NOTE: TIC does not use the same time base as TIME, which means the two functions may not be exactly equal over long periods of time.

3.5.20 String Manipulations

In addition to the specific functions LEN, MCH, and SRH, character strings may be manipulated in other ways. Strings may have a specific character or series of characters picked out; an element may be added or deleted from a string; or two strings may be put together. The following paragraphs describe each of the operations.

Picking a Single Character out of a String The first character of the string is assigned a value of 1, and numbering continues sequentially to the end of the string. These numbers are referred to as index numbers and are separated from the string name by a semi-colon.This operation can be demonstrated by returning to a modification of an earlier example. The names in the example are stored in the following array:

$NML(0,0):	WAREHO	$NML(0,1):	USE
$NML(1,0):	INVENT	$NML(1,1):	ORY
$NML(2,0):	MANAGE	$NML(2,1):	MENT

If the following were entered:

```
10 $A=$NML(0,0;4),1
20 PRINT $A
```

Then "E" would be printed because this is the 4th letter in position (0,0) of array $NML.

Picking a Series of Characters from a String Either a comma or a semi-colon is placed after the final parenthesis of a statement and is followed by a number (the number of characters to be picked). If a comma is used, a null is placed at the end of the picked series of characters; using a semi-colon instead of a comma deletes the null. Remember that the null is important because it is the flag to stop BASIC from reading any further .If the above example is continued, then:

```
10 $A=$NML(0,0;4),5
20 PRINT $A
```

would print "EHOUS" because those are the five characters starting from the 4th letter of position (0,0) in the array. A null exists after the "S" in $A, although this could be omitted in a semi-colon were used instead of a comma.

NOTE: You must assign the result of a character pick to a new variable and then print the new variable. If you try to combine the two, you will get all the string from the designated character to the terminating null.

Character pick can be used to remove a particular element from the TIME function. To do this, assign the current time to a variable such as:

```
05 DIM $TIM(3)
10 TIME $TIM(0)
```

Examine the format of the data stored to determine what characters are needed. Note that the colons are also stored in the variable:

```
$TIM(0)=YR:MO:DY:HR:MN:SC
```

Now the string must be dissected:

```
20 $YR=$TIM(0;1),2
30 $MTH=$TIM(0;4),2
40 $DAY=$TIM(0;7),2
50 $HR=$TIM(0;10),2
60 $MIN=$TIM(0;13),2
70 $SEC=$TIM(0;16),2
```

It is recommended that you do character picks from left to right. Using the correct format for character pick is especially important when dissecting the TIME element. If it is required to have the time or date as numeric data (rather than string), use the format explained in changing strings to real numbers.

Adding an Element to a String A slash (/) is used to insert a character or series of characters into a string. The index system described is also used in this operation. For example, the following program:

```
10 DIM A(10),B(10)
20 $A(0)="ABCDEFG"
30 $A(0;4)=/"..."
40 PRINT $A(0)
50 STOP
```

prints:

```
ABC...DEFG
```

Deleting an Element from a String This has the same form as inserting, except that a number is used after the slash instead of the characters to be inserted. This number refers to the number of characters to be deleted from the string. For example, if the above program had statement 30 replaced with:

```
30 $A(0;4)=/3
```

then the output would be:

```
ABCG
```

Combining Separate Strings Various strings can be combined into one large string by using a plus sign (+) between strings to be added. For example, the following array:

```
$A(0)=$A(0)+"FG"+$C+"Z"
```

would contain all of its original contents along with the contents in C and the letters FGZ. The order of the letters would be as they appeared in the concatenating statement.

Changing Strings to Real Numbers If you have a string which encloses a number, then you may convert that string to a real number by using the following format:

```
<var> = <var>, <var>
```

The first "var" is the variable that contains the real number. The second var is the string variable containing the number you wish to convert. The final var is the variable which contains the first non-numeric character encountered in the string variable. (If there are no non-numeric characters in the string variable, this will be set to a null.) The following example illustrates how a number contained in a string is changed to a real number:

```
10 N="1234",E
20 N1="12DE",E1
30 PRINT N;$E
40 PRINT N1;$E1
50 STOP
```

When this is executed, the output is:

```
1234
12 D
```

Converting a Number to a String Variable This reverse operation is accomplished by assigning a real number to a string variable. Print formatting can be used to store decimal values into a string. For example, the following two statements take numbers and store them as characters:

```
110 $A(0)=2*25
120 $B(0)=#"99.00",25*2
```

If these were printed, the numbers would look like:

```
50
50.00
```

Inserting Non-printable Characters Into Strings There are two ways to insert non-printable characters into a string. The first is to enclose the HEX-ASCII representation of the non-printable character in angle braces. This would then be placed in a string constant at the point where the non-printable character would have gone. The following program demonstrates this method:

```
10 DIM A(8)
20 $A(0)="This rings the bell.<07>"
30 PRINT $A(0)
```

When this program is executed, the following is printed:

```
This rings the bell.
```

and the bell sounds.

The second method for replacing non-printable characters is to use the byte replacement operator. This operator is symbolized by the sign (%). The byte replacement operator causes a low order byte (i.e., the byte stored at an even address) of an integer constant or variable to be copied to the specified character position of a string variable. The byte replacement operator must be followed by a null or an error results. The following program uses the byte replacement operator:

```
10 DIM A(7)
20 $A(0)="This rings the bell."
30 $A(0;21)=%7%0 ! Attach bell character and terminator
40 PRINT $A(0)
```

While these two methods achieve the same result, the manner in which they work is different. Specifically, in the first method the string is stored exactly as it is written, but in the second method the bell character is stored as itself (rather than as a HEX-ASCII representation which must be interpreted before it can be used).

NOTE: The null character cannot be printed in either of these ways.

3.6 BASIC Operators

Operators perform prescribed functions between BASIC variables and constants. The entire string of operators and variables or constants is termed a BASIC expression. There are three types of BASIC operators: arithmetic, logical, and relational.

3.6.1 Arithmetic Operators

The arithmetic operators supported in BASIC are:

+	Addition
–	Subtraction
^	Exponentiation*
*	Multiplication
/	Division
+/–	Signed values

3.6.2 Logical Operators

There are two sets of operators supported by BASIC: "logical" operators, which operate bit-wise on HEX values, and Boolean operators, which manipulate true and false values.
The "logical" operators are:

LNOT	1s complement of a HEX value
LAND	Bit-wise AND of two HEX values
LOR	Bit-wise OR of two HEX values
LXOR	Bit-wise exclusive OR

For example, if A=0AAAAH, B=05555H, and C=0BBBBH, then:

```
LNOT(A) = 5555            A LOR B = FFFF
A LAND B = 0              A LXOR C = 1111
```

* The limit to answers given by exponentiation is E+/–37.

3.6.3 Boolean Operators

Boolean operators are applied to the TRUE or FALSE conditions set by the relational operators. They may also be applied to variables in a program, where all non-zero values are treated as TRUE and all zero values as FALSE. The Boolean operators return a 1 for TRUE or a 0 for FALSE after the evaluation of the expression to which they are applied.

NOT Returns a TRUE value if the expression it is applied to evaluates to FALSE (0); otherwise returns a FALSE value.

AND Returns a TRUE value if both expressions it is applied to evaluate to TRUE (not 0); otherwise returns a FALSE value.

OR Returns a TRUE value if either of the expressions it is applied to evaluate to TRUE (not 0); otherwise returns a FALSE value.

3.6.4 Relational Operators

Relational operators are placed between two arithmetic expressions and return values of 1 for TRUE and 0 for FALSE. These values may, in turn, be used with Boolean logical operators. The relational operators supported in BASIC are:

=	Exactly equal
< >	Not equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
==	Approximately equal (+/− 5.96E−08)

3.6.5 Order of Operator Evaluation

The evaluation of BASIC expressions occurs from left to right and in the following hierarchical order:

1. Expressions in parentheses (inner-most parentheses first)
2. Exponentiation and negation
3. Multiplication and division
4. Addition and subtraction
5. Less than or equal (<=) and not equal (< >)
6. Greater than or equal (>=) and less than (<)
7. Equal (=) and greater than (>)
8. Approximately equal (==) and LXOR
9. NOT and LNOT
10. AND and LAND
11. OR and LOR
12. Assignment (as in LET statements)

3.7 Editing a Program

Once a program is entered into BASIC memory, changes may be made to the program to correct faults or add instructions. This section describes general editing procedures for any terminal and also contains a description of the editing keys on the VPU. Regardless of the terminal used, RUN must be used after editing to re-start execution of the program. Also, an edit line may be up to 100 characters in length. If your particular terminal does not have a line length of the same size, be sure it can wrap around before attempting to create 100-character lines. (The VPU wraps around to give a 100-character line.) BASIC uses square brackets and parentheses interchangeably. You do not have to enter square brackets; parentheses are automatically changed to square brackets where appropriate.

3.7.1 General Editing Procedures

NOTE: Any editing change may be aborted by pressing the [Esc] key before pressing [Return] to enter the change.

Adding a Line To add an instruction, find the place where it is to go in the program, note the line numbers of the statements on either side of the desired location, and type a new line with a line number between the two existing line numbers. When the program is executed or listed, the new statement will be read at the proper time because, when the statement is entered, the program is rearranged into ascending numeric order.

Modification of Existing Line To alter a particular statement, place the line to be altered into EDIT mode. To do this, type the line number and then press [CTRL] and [E] at the same time. After this, you may use the following symbols to position the cursor on the incorrect character and then overwrite it with a correct character. There are also symbols to create or delete characters and spaces within a line. These editing symbols are shown in Table 3-2.

Table 3-2 VPU Editing Symbols

Action/Symbol	Result
Carriage return	enter line
CTRL J (line feed)	enter line and generate next
n CTRL E	edit line number n
CTRL F	move cursor forward
CTRL H	move cursor backward
Space	replace with space
CTRL D n	delete n characters (n=1−9)
CTRL I n	insert n spaces (n=1−9)

CTRL refers to the Control Key, which must be held down while a particular letter is pressed. For example, CTRL F means that you must hold the Control Key down while pressing the F key to move the cursor one space forward.

The carriage return or line feed enters the current line regardless of the cursor position. In addition, line feed automatically generates the next line number by adding 10 to the line being edited.

For delete and insert operations, the number is added after the CTRL D or CTRL I is completed. For example, if you want to add a third variable to the following statement:

```
10 A(J+1)=SQR(B(1,1)-B(1,2)     )
```

First put the line in EDIT mode by typing 10 and then CTRL E. Place the cursor under the last parenthesis by pressing CTRL F or CTRL H. Then press CTRL I together, and finally 7 (the number of spaces needed for a third element). This moves the last parenthesis over as shown below:

```
10 A(J+1)=SQR(B(1,1)-B(1,2)     )
```

After this, insert the variable. Pressing carriage Return or Line Feed completes the operation by entering the edited line into memory. Note that if Line Feed is used, the next line will automatically be put on the screen with an increment of 10 for the line number. The Return key only enters the edited line. The new line looks like the following:

```
10 A(J+1)=SQR(B(1,1)–B(1,2)– B(1,3))
```

Deleting a Line An entire line may be deleted by writing the line number to be deleted and pressing Return. This in effect overwrites the line with a null line, which is ignored during execution. To duplicate a line, display it, then backspace to and change the line number. Press Return to give you two statements: one at the old location and one at the new location.

3.7.2 Editing with a VPU

In addition to the general editing procedures described in the last section, the VPU has keys specifically devoted to editing. If you are using a VPU as your editing device, it is simpler to use these keys rather than the general procedures shown above. The editing keys are located on the keypad to the right of the main set of keys. Some keys are labeled as symbols. The editing keys and their families are listed in Table 3-3.

Table 3-3 VPU Editing Keys

VPU Key	Action
n ROLL DOWN	Display line n for editing
RETURN	Enter displayed line
(down arrow)	Enter line and display next line
INS n	Insert n characters (n=1–9)
DEL	Delete 1 character
[] (box) n	Delete n characters (n=1–9)
––––––––>	Move cursor forward
<––––––––	Move cursor backward
ESC or HELP	Discard changes

Appendix A
Summary of BASIC Language

A.1 Introduction A-2

A.2 Table of BASIC Functions A-3

A.3 Table of BASIC Statements A-4

A.1 Introduction

This appendix lists the commands, statements, and functions of the BASIC language. After the name, the action is summarized. In the chart for BASIC statements, the abbreviation <ln> stands for line number. The summary is not meant to be a complete explanation; turn to the appropriate section for complete information on the command, statement, or function.

Table A-1 BASIC Commands

Command	Action	Reference
CONTINUE(CON)	Restart execution from where it stopped	3.3.1
LIST (LIS)	Display part or all of a program	3.3.2
LOAD	Transfer program to BASIC memory	3.3.3
NEW	Destroy program and reset RET and values	3.3.4
RUN	Start execution from beginning of program	3.3.5
SAVE	Transfer program from BASIC memory	3.3.6
SIZE (SIZ)	List space available in BASIC memory	3.3.7

A.2 Table of BASIC Functions

Table A-2 BASIC Functions

Command	Action	Reference
ABS	Converts negative numbers to positive	3.5.1
ASC	Obtains ASCII value of a character	3.5.2
ATN	Calculates arctangent	3.5.3
BIT	Reads or modifies a variable in bits	3.5.4
COS	Calculates cosine	3.5.5
EXP	Calculates the natural antilogarithm	3.5.6
INP	Converts real numbers to integers	3.5.7
LEN	Obtains the length of a character string	3.5.8
LOG	Calculates the natural logarithm	3.5.9
MCH	Compares two strings for agreement	3.5.10
MEM	Reads or modifies memory in bytes	3.5.11
MWD	Reads or modifies memory in words	3.5.12
NKY	Obtains value from input buffer	3.5.13
RND	Steps through sequence of random numbers	3.5.14
SIN	Calculates sine	3.5.15
SRH	Finds a character in a string	3.5.16
SQR	Calculates square root	3.5.17
SYS	Reads system parameter	3.5.18
TIC	Calculates time intervals	3.5.19

Table A-3 BASIC Statements

Statement	Description
CALL	See the individual routines: FIND, IOREAD, IOWRITE, PCREAD, PCWRITE, and SRTC.
DATA	<ln> DATA <var.1> , <var.2> ,...,<var.N> Source of data stored in a program; used with READ and RESTOR. See Section 3.4.16. NOTES: Must have a <ln> and it cannot appear on a multi-statement line.
DEF	<ln> DEF FN <LTR> (dum.vars.) = <expression> Initializes user−defined functions. See Sec. 3.4.1. NOTES: Must have a <ln> ; cannot appear on a multi-statement line nor be followed by a tail-remark (!)
DIM	<ln> DIM ARRAYNAME (size1, size2,...,sizeN) Assigns memory space for an array. See Sec. 3.4.2. NOTES: Retentive memory cannot be set with this; use the NEW command.
ELSE	<ln> ELSE <BASIC expression> Executes only when the preceding IF−THEN is false. See Sec. 3.4.17. NOTES: Cannot be used on multi-statement line.
END	<ln> END Final termination of program and execution. See Sec. 3.4.12. NOTES: Must have a <ln>.
ERROR	<ln> ERROR <destination line number> Transfers execution upon error. See Sec. 3.4.3. NOTES: Works only once; a second ERROR must be used if further errors are to be detected. RETURN must be used at end of error subroutine.
ESCAPE	<ln> ESCAPE Enables the ESCAPE (abort) key. See Sec. 3.4.5.
FIND	<ln> CALL"FIND", <var. >, <var.> Looks in BASIC memory for the address of a specific variable. See Sec. 3.4.5.
FOR	<ln> FOR <var.> = <exp.> TO <exp.> STEP <exp.> Begins repeating instruction block; used with NEXT. See Sec. 3.4.6. NOTES: Default value of Step is 1.

Table A-3 BASIC Statements (continued)

Statement	Description
GOSUB	<ln> GOSUB <destination line number> Transfers execution to program subroutine. See Sec. 3.4.19. NOTES: RETURN must be use at end of subroutine.
GOTO	<ln> GOTO <destination line number> Transfers execution to a specified line. See Sec. 3.4.17.
IF–THEN	<ln> IF <exp.> THEN <expression or statement> Branches execution in two directions depending on whether IF is true or false. See Sec. 3.4.17. NOTES: When the IF is false, execution passes to next line; rest of the line is not read.
INPUT	<ln> INPUT <var.1> , <var.2 >,..., <var.N> Allows data entry from the keyboard. See Sec. 3.4.7.
IOREAD	<ln> CALL "IOREAD", <no.> , <array> , <no.> Reads P/C data directly from I/O registers. See Sec. 3.4.18.
IOWRITE	<ln> CALL"IOWRITE" , <no.> , <array> , <no.> Writes data directly to P/C I/O registers. See Sec. 3.4.18.
LET	<ln> LET <var.> = <exp.> Assigns a value to program variable. See Sec. 3.4.8.
NEXT	<ln> NEXT <var> Terminates repeating instruction block; used with FOR. See Sec. 3.4.6.
NOESC	<ln> NOESC Disables ESCAPE (abort) key. See Sec. 3.4.4.
ON	<ln> ON <var.> THEN GOSUB or GOTO<destination> Transfers execution when specified value encountered. See Sec. 3.4.17. NOTES: Cannot be used on a multi-statement line nor can a tail–mark (!) follow it.
PCREAD	<ln> CALL"PCREAD", < var.> , <array> ,< no.> Reads up to 26 consecutive data points per scan from a P/C variable. See Sec. 3.4.18.
PCWRITE	<ln> CALL"PCWRITE", <var.> , <array> ,< no.> Sends up to 30 consecutive data points per scan to a P/C variable. See Sec. 3.4.18.
POP	<ln> POP Removes top statement from the "wait" stack where it had been placed by a GOSUB. See Sec. 3.4.19.

Table A-3 BASIC Statements (continued)

Statement	Description
PRINT	<ln> PRINT <var.1>, <var.2 >,..., <varN> Writes data to output device in either free format or formatted forms. See Sec. 3.4.9.
RANDOM	<ln> RANDOM <exp.> Sets seed for random number generation. See Sec. 3.4.10.
READ	<ln> READ <var.1> , <var.2> ,... <varN> Obtains data from program data list; used with DATA and RESTOR. See Sec. 3.4.16. NOTES: Must have a <ln>.
REM	<ln> REM <remark> Contains comments that are ignored during execution. See Sec. 3.4.11.
RESTOR	<ln> RESTOR <optional line number> Resets data pointer; used with DATA and READ. See Sec. 3.4.16 NOTES: Must have a <ln>.
RETURN	<ln> RETURN Transfer execution from a subroutine to the line after ERROR or GOSUB. See Sec. 3.4.19.
SRTC	<ln> CALL"SRTC", <var.> , <var.> Sends task codes to the P/C. See Sec. 3.4.18.
STOP	<ln> STOP Logical end to execution. See Sec. 3.4.12. NOTES: Must have a <ln>.
TAB	<ln> PRINT TAB(column), <exp.> Writes data to specific columns. See Sec. 3.4.13. NOTES: Must have a <ln>.
TIME	<ln> TIME <YR> , <MO >, <DY >, <HR >, <MN> (set time) <ln> TIME (read time) (ln> TIME <var.> (store time) Accesses internal clock. See Sec. 3.4.14. NOTES: For short spans of time, use TIC.
UNIT	<ln> UNIT <expression> Sets ports for input and output. See Sec. 3.4.15. NOTES: Default value is 1.

Appendix B

VPU Hex-ASCII Codes

B.1 VPU Character Codes **B-2**

B.1 VPU Character Codes

The following list gives the HEX-ASCII codes for the VPU. These codes may be placed in braces and inserted in place of the character in a program. For devices other than a VPU, please consult that device manual for the appropriate HEX-ASCII codes.

Table B-1 VPU Character Codes

Key or Sequence	ASCII Character	Representation Decimal	Representation Hexadecimal
CTRL F1	NULL	00	00
CTRL A	SOH	01	01
CTRL B	STX	02	02
CTRL C	ETX	03	03
CTRL D	EOT	04	04
CTRL E	ENQ	05	05
CTRL F	ACK	06	06
CTRL G	BEL	07	07
CTRL H	BS	08	08
CTRL I	HT	09	09
CTRL J	LF	10	0A
CTRL K	VT	11	0B
CTRL L	FF	12	0C
ENTER	CR	13	0D
CTRL N	SO	14	0E
CTRL O	SI	15	0F
CTRL P	DLE	16	10
CTRL Q	DC1	17	11
CTRL R	DC2	18	12
CTRL S	DC3	19	13
CTRL T	DC4	20	14
CTRL U	NAK	21	15
CTRL V	SYN	22	16
CTRL W	ETB	23	17
CTRL X	CAN	24	18
CTRL Y	EM	25	19

Table B-1 VPU Character Codes (continued)

Key or Sequence	ASCII Character	Representation Decimal	Representation Hexadecimal
CTRL Z	SUB	26	1A
ESC	ESCAPE	27	1B
CTRL 1	FS	28	1C
CTRL 2	GS	29	1D
CTRL 3	RS	30	1E
CTRL 4	US	31	1F
SPACE	SPACE	32	20
SHIFT 1	!	33	21
SHIFT '	"	34	22
SHIFT 3	#	35	23
SHIFT 4	$	36	24
SHIFT 5	%	37	25
SHIFT 7	&	38	26
'	'	39	27
SHIFT 9	(	40	28
SHIFT 0	)	41	29
SHIFT /	*	42	2A
SHIFT –	+	43	2B
,	,	44	2C
SHIFT ARROW RT	–	45	2D
.	.	46	2E
/	/	47	2F
0	0	48	30
1	1	49	31
2	2	50	32
3	3	51	33
4	4	52	34
5	5	53	35
6	6	54	36
7	7	55	37
8	8	56	38
9	9	57	39

Table B-1 VPU Character Codes (continued)

Key or Sequence	ASCII Character	Representation Decimal	Hexadecimal
SHIFT ;	:	58	3A
;	;	59	3B
SHIFT ,	<	60	3C
=	=	61	3D
SHIFT .	>	62	3E
SHIFT 8	?	63	3F
SHIFT 2	@	64	40
SHIFT a	A	65	41
SHIFT b	B	66	42
SHIFT c	C	67	43
SHIFT d	D	68	44
SHIFT e	E	69	45
SHIFT f	F	70	46
SHIFT g	G	71	47
SHIFT h	H	72	48
SHIFT i	I	73	49
SHIFT j	J	74	4A
SHIFT k	K	75	4B
SHIFT l	L	76	4C
SHIFT m	M	77	4D
SHIFT n	N	78	4E
SHIFT o	O	79	4F
SHIFT p	P	80	50
SHIFT q	Q	81	51
SHIFT r	R	82	52
SHIFT s	S	83	53
SHIFT t	T	84	54
SHIFT u	U	85	55
SHIFT v	V	86	56
SHIFT w	W	87	57
SHIFT x	X	88	58
SHIFT y	Y	89	59

Table B-1 VPU Character Codes (continued)

Key or Sequence	ASCII Character	Representation Decimal	Representation Hexadecimal
SHIFT z	Z	90	5A
CTRL ,	[	91	5B
CTRL /	\	92	5C
CTRL .	]	93	5D
SHIFT 6	^	94	5E
CTRL =	(UNLN)	95	5F
...		96	60
A	a	97	61
B	b	98	62
C	c	99	63
D	d	100	64
E	e	101	65
F	f	102	66
G	g	103	67
H	h	104	68
I	i	105	69
J	j	106	6A
K	k	107	6B
L	l	108	6C
M	m	109	6D
N	n	110	6E
O	o	111	6F
P	p	112	70
Q	q	113	71
R	r	114	72
S	s	115	73
T	t	116	74
U	u	117	75
V	v	118	76
W	w	119	77
X	x	120	78
Y	y	121	79

Table B-1 VPU Character Codes (continued)

Key or Sequence	ASCII Character	Representation	
		Decimal	Hexadecimal
Z	z	122	7A
CTRL ;	{	123	7B
CTRL 8	|	124	7C
CTRL '	}	125	7D
CTRL 0	~	126	7E
DEL	DEL	127	7F

Appendix C Reserved Words

C.1 Reserved Words List **C-2**

C.1 Reserved Words List

The following reserved words cannot be used as variable names or any part of a variable name:

ABS	FIND	MEM	RUN
ASC	FOR	MWD	SAVE
ATN	GOSUB	NEW	SCH
BIT	GOTO	NEXT	SIZE
CALL	IF	NKY	SRH
CON	IOREAD	NOESC	SIN
CONTINUE	IOWRITE	ON	SRTC
COS	INP	PCREAD	SQR
CRB	INPUT	PCWRITE	STEP
CR	LEN	POP	STOP
DATA	LET	PRINT	SYS
DEF	INPUT	PRO	TAB
DIM	LEN	RANDOM	THEN
ELSE	LET	READ	TIC
END	LIS	REM	TIME
ERROR	LIST	RESTOR	UNIT
ESC	LOAD	RET	
ESCAPE	LOG	RETURN	
EXP	MCH	RND	

Appendix D
Error Format

D.1 Introduction D-2

D.2 Error Messages D-3

D.1 Introduction

BASIC checks for syxtax errors both when the statements are entered and when the program is executed. If you attempt to enter a line with a syntax error in it, entry of the statement is prevented. The line you attempt to enter is displayed with the cursor over the character that caused the error and a message describing the error. For example, attempting to enter the following:

```
100 PRINT SIN(A(0)
```

causes the following:

```
***UNMATCHED PARENTHESES***
100 PRINT SIN(A(0)
```

If errors are detected during execution, execution is halted if no ERROR statement is in effect. When execution halts, the error message and line number containing the error are written to the output device. You may also read SYS(1) and SYS(2) to obtain the error message number and the line number respectively. The following is an example of what is written when an execution error is encountered and no ERROR statement is in effect:

```
***SUBSCRIPT OUT OF RANGE*** AT 100
```

D.2 Error Messages

Table D-1 contains is a list of error messages and their code numbers.

Table D-1 Error Messages

Code Number	Error Message
01	Syntax error—invalid BASIC grammar
02	Unmatched parentheses—check for left and right pairs
03	Invalid line number—line number does not exist
04	Illegal variable name—violates one of the variable name rules: • Nothing • One or two more capital letters • A numeric value from 0 to 127
05	Too many variables—limit of 128 unique variable names. To clear this error: • Reduce the number of variable names used • Store program to disk • Clear module memory via the NEW command • Reload program
06	Illegal character—use of lowercase letter in commands
07	Expecting operator—syntax error
08	Illegal function name—functions must be named
09	Illegal function argument function can only deal with numeric data dummy parameters for a function can be only one letter variable names
10	Storage overflow—out of memory
11	Stack overflow—more than 20 nested subroutines
12	Stack underflow—return without GOSUB
13	No such line number—trying to edit a nonexistent line
14	Expecting string variable—make sure the $ precedes the variable name and that the data really is a string
15	Invalid screen command—invalid direct command
16	Expecting dimensioned variable—check syntax of DIM statement

Table D-1 Error Messages (continued)

Code Number	Error Message
17	Subscript out of range—subscript larger than that dimensioned
18	Too few subscripts—dimensioning a single subscripted array and addressing it as a double subscripted array: • More commonly caused by memory mismanagement that overwrites existing memory • Putting more than five characters in a string and not allowing for overrun • Violating the format for character manipulation, especially with the use of TIME
19	Too many subscripts—dimensioning a triple subscripted array and addressing it as a double subscript
20	Expecting simple variable—applying a subscript to an undimensioned variable
21	Digits out of range—assigning unit to more than 4, or the retentive array (using NEW command) to more than 4095
22	Expecting variable—syntax error
23	READ out of data—all data has been read; either more data is needed or data should be restored
24	READ type differs from DATA type—if reading data to string variables, each data element must be enclosed in quotation marks and separated by commas.
25	Square root of a negative number—imaginary numbers are not supported
26	Log of a non−positive number—not supported
27	Expression too complex—nested too deep
28	Division by zero—infinite numbers not supported
29	Floating point overflow—numbers > 1x10 E−/5 in excess 64 notation are illegal. Misreading a string value into a floating-point variable may cause this error.
30	Fix error—most generally flagged on a PCWRITE statement, but caused prior to that. An error in trying to convert a value to integer to transmit to the PLC. Can be corrected in most cases by taking the integer part (INP) of a variable before attempting a PCWRITE.
31	FOR without NEXT—loop end parameter omitted

Table D-1 Error Messages (continued)

Code Number	Error Message
32	NEXT without FOR—loop start parameter omitted
33	EXP function has invalid argument—the largest allowable exponential value is 87. Anything larger will flag an error.
34	Un-normalized number—a normalized number will be stored in excess 64 notation with the fraction as large as possible, and the exponent as small as possible. Should the interpreter find a bit pattern that does not agree with this convention, an error will occur. This is likely to occur when a numeric variable attempts to read a string location.
35	Parameter error—probably relates to user–defined function parameters
36	Missing assignment operator—syntax calls for an "="
37	Illegal delimiter—possibly caused by violating rules governing the use of multiple statement separator (::)
38	Undefined function—using a user–defined function that has not been defined
39	Undimensional variable—all subscripted arrays must be dimensioned. The $ that precedes string variables is omitted in the dimension statement.
40	Undefined variable—a value must be assigned to a variable before it can appear on the right side of an equation
41	Not used
42	Not used
43	Not used
44	Not used
45	Not used
46	Not used
47	Unknown command name—caused by making a call to a nonexistent subroutine or function
48	Invalid response from the PLC—check hardware. Are connections good? Properly grounded system? Problem with SF communication?

Table D-2 Error Messages (continued)

Code Number	Error Message
49	Program cannot be continued—if program is stopped (either through the use of a STOP statement or use of the ESC key), and the program is edited, it cannot resume execution via the CONTINUE command. If the program is not edited, it usually can continue.
50	Number of points requested is too large—excessive number of data elements requested by PCREAD or PCWRITE
51	No response from PLC—the SFIC watchdog timed out. Are connections between base and PLC good? Is PLC in RUN? Is P/C GOOD light on?
52	Illegal PLC address specified—legal P/C addresses are: V, C, X, Y, WX, WY, TCC, TCP, DSC, DCC, DCP. They must be assigned to a string variable; the $ on the variable name does not appear in the call to PCREAD or PCWRITE
53	Illegal non−HEX data encountered—encountered in using SRTC subroutine when data is not in correct HEX format (letters greater than F)
54	No response to the store and forward—time out on SRTC call
55	Store and forward buffer is full—PLC buffer full
56	Store and forward error—catch all error messages for incorrect use of task code 43
57	Fix error—same as error code 30 but is specific to special function communication fix errors
58	Illegal I/O point specified—used with IOREAD and IOWRITE, the only legal I/O points are numbered 1 through 8
59	All requested data not read—read was started, but not completed. May be error of PLC or BASIC
60	Unknown subroutine specified in CALL—the only legal call subroutines are: PCREAD, PCWRITE, IOREAD, IOWRITE, SRTC, FIND
61	Subroutine error—catch−all error for above subroutines
62	Communications parity error—serial port error; a bit was dropped
63	PLC task code error encountered—miscellaneous task code errors