Series 30/40 BASIC Guide

DOC-IWS-280
Revision B
September 1996

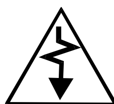
Nematron Corporation
5840 Interface Drive
Ann Arbor, Michigan 48103
Phone: 734-214-2000
Fax: 734-994-8074

Nematron®

Open minds. Open systems. Real solutions.

Important Information

Note: This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at his own expense.



Electrical Shock Hazard! Do not operate the unit with its back cover removed. There are hazardous voltages inside. Servicing of the equipment should only be done by qualified and authorized personnel.

This publication is issued on condition that it is not copied, reprinted, or disclosed to a third party, either wholly or in part, without the prior written consent of Nematron Corporation.

Nematron Corporation assumes no liability or responsibility for the loss or damage, direct or indirect, arising from the use of this product. Nematron Corporation reserves the right to change this product's specifications without notice.

This document is based on information available at the time of its publication. While efforts have been made to be accurate, the information contained in this guide does not purport to cover all details or variations in hardware and software. This guide may describe features which are not present in all hardware and software systems.

Nematron and FloPro are registered trademarks, and NemaSoft, PowerVIEW, Industrial Control Computer, Industrial Workstation, and AutoNet are trademarks of the Nematron Corporation. All other brand or product names are trademarks or registered trademarks of their respective companies.

Release Date	Revision
July 1993	Original release
September 1996	Addition of IWS-40

©1993, 1996 Nematron Corporation, All Rights Reserved. Printed in U.S.A.

Contents

Chapter 1 Getting Started

Symbols ... 1-1
Hardware and Software Requirements ... 1-3

Chapter 2 Display and Keyboard

Display ... 2-1
- Character Set ... 2-1
- Contrast Adjustment ... 2-1

Keyboard ... 2-1
- Keyboard Faults ... 2-1
- Keyboard Inserts ... 2-1

Chapter 3 Hardware Installation

Chapter 4 Software Installation

Power-up Sequence ... 4-1
Downloading Firmware ... 4-1
Required Equipment ... 4-2
- Downloading ... 4-2

Version Numbers ... 4-4

Chapter 5 On-Line Configuration

Accessing the On-Line Configuration Menu ... 5-1
Main Configuration Menu ... 5-1
- Selecting a Menu Item ... 5-1
- Selecting Parameters ... 5-1
- Changing Parameters ... 5-1
- Exiting ... 5-1

Clock ... 5-2
- Date Format ... 5-2
- Date ... 5-2
- Time ... 5-2

Keyboard/Display ... 5-2
- Auto-Repeat ... 5-2
- Repeat Delay ... 5-2
- Repeat Rate ... 5-2
- Line End Action ... 5-3
- Screen End Action ... 5-3
- Cursor ... 5-3

Communications ... 5-4
- Baud Rate ... 5-4
- Parity ... 5-4
- Data Bits ... 5-4
- Stop Bits ... 5-4

Transmit Handshake 5-4
RTS Control 5-4
Receive Handshake 5-5
Convert Parity Error 5-5
Parity Error Char 5-5
Utility 5-6
Communications Test 5-6
Upload 5-7

Chapter 6 Service

Changing the Fuse 6-1
Memory Write Disable 6-2
Troubleshooting 6-3

Chapter 7 Concepts

Command/Run Modes 7-1
Console 7-1
Ports 7-1
Statements 7-2
Numbers and Constants 7-2
Variables 7-3
Scalar Variables 7-3
Array Variables 7-3
Memory Allocation 7-3
Variable Names 7-4
Floating point Variables 7-4
Integer Variables 7-4
String Variables 7-4
Built-in Variables 7-4
Operators and Expressions 7-4
Operators 7-4
Expressions 7-5
Relational Expressions 7-5
Memory Usage 7-5

Chapter 8 How to Use BP Software

Starting up BP 8-1
Starting up the Workstation 8-2
Summary of BP Functions 8-3
Creating a Program 8-4
Line Editing 8-4
Command Editing 8-4
Sending a Program to the Workstation 8-5
Receiving a Program from the Workstation 8-5
Configuring BP 8-6
Color Configuration 8-6
Serial Port Configuration 8-7
Display Configuration 8-7
Save and Restore Defaults 8-7
Additional BP Functions 8-7
DOS Gateway 8-7
Data Capture 8-7
History Window 8-7
Hex Display 8-8

Keyboard Macros ... 8-8
Troubleshooting ... 8-9

Chapter 9 Application Suggestions

Special Keyboard Functions ... 9-1
Halting a Program; [Ctrl]-C ... 9-1
On-Line Configuration; [F3] ... 9-1
Restore COM1 Defaults; [F4] ... 9-2
Program Development Techniques ... 9-2
Storing Programs in Permanent Memory ... 9-3
Auto-Starting Programs ... 9-3
Program Protection ... 9-3
Preserving Data During a Power Loss ... 9-4
Printing to the Display ... 9-4
Time-Driven Functions ... 9-4

Chapter 10 Commands and Statements

ABS ... 10-1
ASC ... 10-2
ATN ... 10-3
BIT ... 10-4
CALL ... 10-5
CBY ... 10-6
CHR$ (right side) ... 10-7
CHR$ (left side) ... 10-8
CLEAR ... 10-9
CLOCK ... 10-10
CLS ... 10-11
CONT ... 10-12
COPY ... 10-13
COS ... 10-14
CR ... 10-15
CSRLIN ... 10-16
DATA ... 10-17
DATE$ ... 10-18
DBY ... 10-19
DEL ... 10-20
DIM ... 10-22
DIR ... 10-24
DO . . . UNTIL ... 10-25
DO . . . WHILE ... 10-27
DUMP ... 10-28
END ... 10-29
ERR and ERL ... 10-30
EXP ... 10-31
FOR . . . NEXT ... 10-32
FREE ... 10-33
GOSUB . . . RETURN ... 10-34
GOTO ... 10-35
HEX$ ... 10-36
HVAL ... 10-37
IBY ... 10-38
IF . . . THEN . . . ELSE ... 10-39
IN# ... 10-40
INKEY$ ... 10-41

INPUT 10-42
INPUT$ 10-47
INSTR 10-48
INT 10-49
INV 10-50
LD@ 10-51
LEFT$ 10-55
LEN 10-56
LET 10-57
LIST 10-58
LOCATE 10-60
LOG 10-61
MID$ (right side) 10-62
MID$ (left side) 10-63
MTOP 10-64
NEW 10-65
NOT 10-66
ON ERROR GOTO 10-67
ON . . . GOSUB 10-69
ON . . . GOTO 10-70
ON TIME = . . . GOSUB 10-71
OPEN COM 10-72
PH0. and PH1. 10-80
PI 10-81
PLEN 10-82
POP 10-83
POS 10-84
PR# 10-85
PRINT 10-86
PRINT USING 10-88
PUSH 10-90
RAM 10-90
REACT 10-92
READ 10-93
REM 10-94
RENUM 10-95
RESTORE 10-96
RESUME 10-97
RETI 10-98
RETURN 10-99
RIGHT$ 10-100
RND 10-101
ROM 10-102
RUN 10-103
SDIM 10-104
SGN 10-105
SIN 10-106
SPC 10-107
SQR 10-107
ST@ 10-108
STOP 10-109
STR$ 10-110
TAB 10-111
TAN 10-112
TIME 10-113
TIME$ 10-114
TROFF 10-115
TRON 10-116

VAD.......... 10-116
VAL.......... 10-117
VARPTR.......... 10-118
VERSION.......... 10-119
XBY.......... 10-120

Chapter 11 Operators

Precedence.......... 11-1
Addition (+).......... 11-2
Subtraction or negation (–).......... 11-3
Multiplication (*).......... 11-4
Division (/).......... 11-5
Exponentiation (^).......... 11-6
Equal (=).......... 11-7
Not equal (<>).......... 11-8
Less than (<).......... 11-9
Greater than (>).......... 11-10
Less than or equal to (<=).......... 11-11
Greater than or equal to (>=).......... 11-12

Chapter 12 Logic

Truth Tables.......... 12-1
AND.......... 12-2
OR.......... 12-3
XOR.......... 12-4
INV.......... 12-5
INV AND.......... 12-6
INV OR.......... 12-7
INV XOR.......... 12-8

Chapter 13 CALLs

CALL 12; clear to the end of the display.......... 13-2
CALL 13; clear to the end of the line.......... 13-2
CALL 30; turn on COM1's RTS line.......... 13-3
CALL 33; turn off COM1's RTS line.......... 13-3
CALL 38; enter the on-line configuration menu.......... 13-4
CALL 39; enter the monitor.......... 13-4
CALL 40 and 41; return characters in COM1 buffer.......... 13-5
CALL 82; print variable list.......... 13-6

Appendices

A. Speed-up Hints.......... A-1
B. Differences between Series 30/40 BASIC and GW-BASIC.......... B-1
C. Memory Map.......... C-1
D. Reference.......... D-1
  Command Summary.......... D-1
  Operators.......... D-6
  CALLs.......... D-7
  Workstation Key Codes.......... D-7
  CTRL Characters Sent to Workstation.......... D-8
  Ports.......... D-8
E. Error Codes.......... E-1

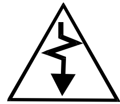
Chapter 1

Getting Started

This guide describes the details of the BASIC language in the Series 30/40 Workstations. To get the most from this guide, you should already be familiar with the BASIC language.

Symbols

Certain symbols in this guide help make you aware of critical information, as shown below:



This symbol emphasizes that hazardous voltages, currents, temperatures, or other conditions which cause **personal injury** exist in this equipment or may be associated with its use.

This symbol appears when **equipment damage** may occur if care is not taken.

Note A note gives information that pertains to a specific firmware release or one form of the hardware only.

The following table shows the meaning of various symbols used throughout this guide. Words enclosed in square brackets usually represent keys on the keyboard as shown in the first example below:

Symbol	Refers to
[Enter]	Key labeled "Enter" (or "Return") on the keyboard.
UPPER CASE	Words that you type exactly as shown.
[Filename]	Name of a screen file entered by the user.
lower case	An entry that varies based on your needs.
nnn	Number that you enter .
var	Variable name of any type, such as "A" or "A$."
nvar	Numeric variable such as "A" or "A%."
svar	String variable, such as A$.
const	Constant number, such as "123.45" or "5."
expr	Any expression that returns a value, including simple variables such as "A" as well as such expressions as "A/B" or "A$ + B$."
aexpr	Any number, variable, or combination that returns a numeric value in the range of a floating point number.
iexpr	Any number, variable, or combination that returns a numeric value in the range of an integer number.
rexpr	Any number, variable, or combination that returns a 0 or non-zero value; typically includes a relational operator such as "<" or ">=."
sexpr	Any string or combination that returns a string.
{ }	Braces indicate an optional entry.

Screens on your computer are enclosed in boxes as shown in the example below:

```
NEMATRON CORPORATION
Series 30/40 BASIC
  Version 5.50

READY - RAM 1
>
```

Workstation screens are enclosed in double boxes, as shown in the example below:

```
Welcome to V5.50 B
Series 30/40 BASIC
```

This guide indicates something you are supposed to type by italicizing it: *Type This.*

Hardware and Software Requirements

You need the following equipment to write programs for a Series 30/40 Workstation with BASIC:

- **Series 30/40 Workstation**

- **PC-compatible with:**
  Minimum 640K of memory
  Floppy disk drive
  Hard disk drive with 1.5 megabytes of available space
  Serial communications port (COM1)
  Monochrome or color monitor
  PC system software (DOS 2.0 or later)

- **IWS-SETUP-BP-30**
  One 3 1/4" setup diskette. Contact the factory if you require the setup program on a 5 1/4" diskette.
  Cable to connect your PC's 9-pin serial port to your Series 30/40 Workstation’s COM1 port. If your PC has a 25-pin serial port, a standard 9 to 25 pin adapter will allow you to use the download cable.

Chapter 2

Display and Keyboard

This chapter describes how to control the Workstation's display and how the keyboard operates.

Display

The Series 30 Workstation displays two lines with 20 characters on each line of a liquid crystal display (LCD) with LED backlighting. Because of the 100,000 hour half-life of the backlight, the Workstation has no "screen saver" function.

The Series 40 Workstation comes with a vacuum fluorescent display (VFD) with two lines of 20 characters each.

Character Set

The character set on both models includes the entire standard ASCII character set, but is limited only to U.S. characters and eight programmable characters.

Contrast Adjustment (IWS-30 only)

To adjust the contrast of the display on the IWS-30, rotate the knob on the back of the Workstation.

Keyboard

The keyboard on the Series 30/40 Workstation is a sealed-membrane type with stainless steel domes that provide tactile feedback.

Keyboard Faults

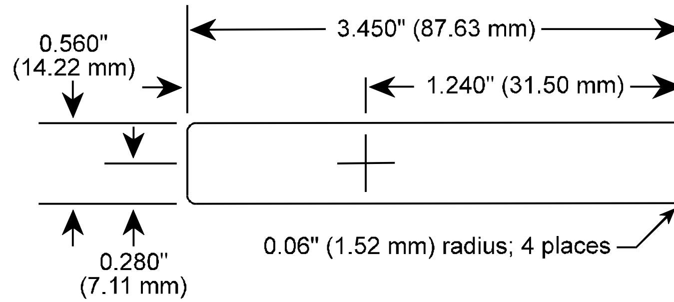
If the Workstation appears to ignore all key presses, the keyboard may be faulty. If the Workstation senses any invalid combination of keys pressed simultaneously, it accepts no further key presses until all keys are released. This provides some protection against a stuck key.

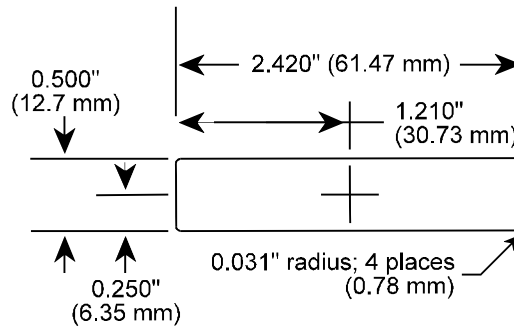
Keyboard Inserts

You can change the key and logo legends by replacing inserts that slide in behind the keyboard. To replace the inserts, you must first remove the back cover.

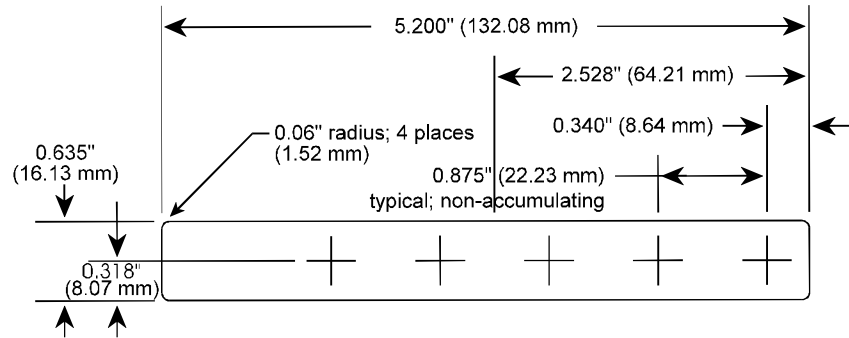
For a modest tooling charge and additional cost per unit, Nematron can provide custom inserts to meet your needs. As an alternative, you can make your own inserts according to the following instructions.

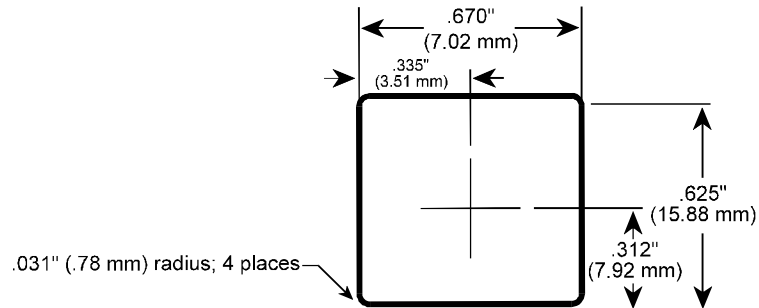
Logo Insert You can install your own logo in place of Nematron's logo. The drawing on the to shows the insert, which you must cut as shown; the cross mark would appear at the center of the window. The drawing on the bottom shows the window, which indicates the maximum size you can print your logo.





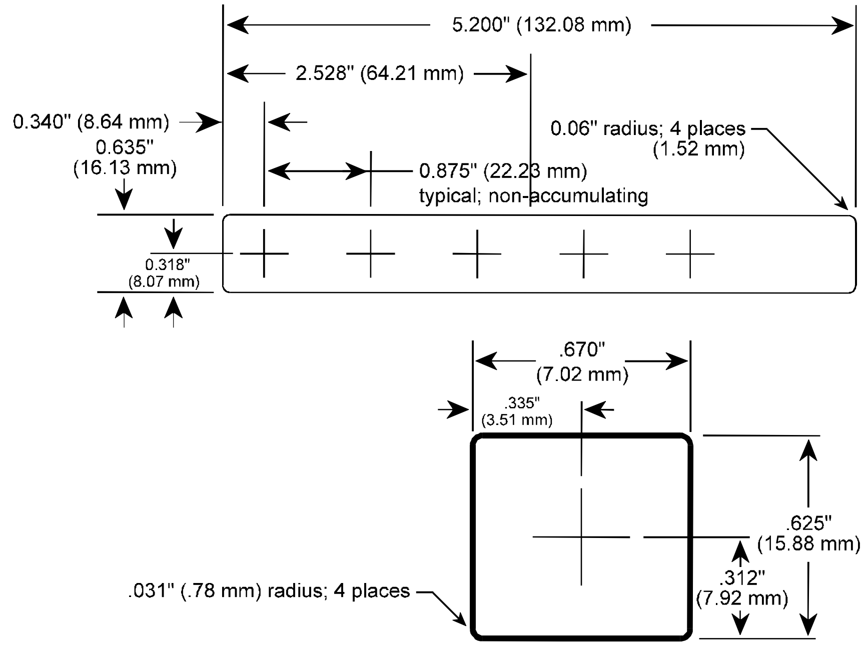
Top Row Insert The drawing on the top shows how to cut an insert for the top row of keys. The cross marks indicate the center of each key's legend. The drawing on the bottom shows the window for each key, which indicates the maximum size you can print your legends.





Bottom Row Insert

The drawing on the top shows how to cut an insert for the bottom row of keys. The cross marks indicate the center of each key's legend. The drawing on the bottom shows the window for each key, which indicates the maximum size you can print your legends.



Chapter 3

Hardware Installation

For information on installing and connecting the Series 30/40 units, refer to the Series 30/40 Installation Guide (DOC-IWS-271; Rev. C), which is included with the 30/40 Workstation.

Chapter 4

Software Installation

This chapter describes how to install the software included with IWS-SETUP-30/40 on your hard disk drive and how to download firmware to your Workstation.

Power-Up Sequence

After you apply power, the Workstation checks for valid firmware memory. If the firmware is valid, the unit displays its model number and performs a brief self-test that includes memory, keyboard and display.

To speed up the display of the self-test, press [x]. At the end of the self-test, the unit begins normal operation, which varies depending on the firmware.

If firmware memory is empty or invalid, the Workstation performs an exhaustive test of the memory. During this test, which lasts about one minute, the Workstation displays the following:

```
Now testing Flash.
Please wait . . .
```

At the conclusion of the test, the Workstation displays the following message (if the firmware appears faulty, the message starts with “Invalid” instead of “Empty”):

```
Empty firmware;
you must download.
```

Downloading Firmware

This section describes how to change the firmware in your Series 30/40 Industrial Workstation. You may have to change your firmware either to change its type or to upgrade the current version.

Three types of firmware are available

- PLC Workstation (IWS-SETUP-30)
- Industrial Computer with BASIC (IWS-SETUP-BP-30)
- Industrial Terminal (all Series 30/40 models leave the factory with this firmware installed), which is included with the IWS-SETUP-30 kit.

Because the unit stores its firmware in “flash” memory, each hardware model supports all three types of firmware. To change firmware, you simply plug in a PC-compatible and transmit a firmware file.

When a new Workstation leaves the Nematron factory, it has the Industrial Terminal firmware already downloaded. If you want to program your unit in BASIC, then you must download new firmware.

Required Equipment

In order to download firmware, you must have the following equipment and materials available:

- One 3 1/2” diskette included in the IWS-SETUP-30/40 package.
- PC-compatible with at least 256K RAM, one serial port, one hard disk drive, and one floppy disk drive.
- Cable to connect your PC to the Workstation; this is part number CBL-C2, and is included in the IWS-SETUP-30 package.
- Series 30/40 Industrial Workstation.

Downloading

The following instructions describe how to download new firmware to your Series 30/40 Industrial Workstation. Before you begin, you should ensure that you have the necessary equipment available.

1. If your new Workstation displays the following screen after completing its power-up self-test, then proceed to step 2:

```
First-time power-up!
Download or hit key.
```

If your Workstation does not display this screen, then you must hold down the [↑] and [↓] keys simultaneously while powering up the Workstation.

Note Unlike other hidden keyboard commands that the system can accept at any time during the power-up self-test, the system looks for the keyboard command to enter the download mode only at the moment of power-up. In other words, you must hold down [↑] and [↓] even *before* you apply power.

This immediately brings up the following message:

```
Ready to accept
firmware download.
```

2. If you have already loaded the IWS-SETUP-BP-30 software onto your hard drive, then skip to step 4. Otherwise, create a directory on your hard drive for the software and change the current path to that directory.

   For example, if you want the directory to be “IWS” then you would type *MD IWS* and press [Enter] to create the directory. Then you would type *CD IWS* and press [Enter] to change the current path to that directory.

3. Insert the diskette labeled IWS-SETUP-BP-30 into your PC's floppy disk drive and type *Copy a:*.* d:\IWS* (d = disk drive where you created the subdirectory\IWS) and press [Enter]. This copies the files to your hard drive.
4. Connect the download cable between the 9-pin COM1 or COM2 port on your PC and the COM1 port on your Workstation.

5. Type DOWNLOAD followed by the firmware file you want to download to your PC. Listed below are your filename choices:

Filename	**Firmware Description**
H0.ROM	Allen-Bradley PLC-5 and PLC-2 (RS-232); GE Micro Allen-Bradley Micrologix 1000 and SLC-5/03 and 5/04 (RS-232)
H1.ROM	Allen-Bradley PLC-2 (programming port)
H2.ROM	Allen-Bradley 100/150 and Modbus
H3.ROM	Allen-Bradley SLC 500-RS-485 (available separately)
H4.ROM	GE Fanuc Series Ninety; Omron SP Series
H5.ROM	GE Fanuc Series 1, 3, 5, 6; Texas Instruments 305 and 435; Koyo DL330, DL340, DL430, and DL440; Square D 100 to 700
H6.ROM	Omron; Simatic/TI 500/505 Series
H7.ROM	Siemens 90U to 115U; Westinghouse 700-1250
H8.ROM	Mitsubishi A, F, and FX Series
H9.ROM	Hitachi H Series; IDEC Micro 3
HA.ROM	IDEC Micro-1, FA-1, FA-2, and FA-3; Toshiba M, EX, and T2
HB.ROM	BASIC
HC.ROM	Telemecanique
HTM.ROM	Terminal

For example, to set up your unit for BASIC, type *DOWNLOAD HB*.

To download using COM2 on your PC, type -2 before the filename; for example, *DOWNLOAD -2 HB*.

As of this guide's publication, the most recent release of DOWNLOAD was version 1.81. The version number is automatically displayed when you run the program.

Version Numbers

As of this guide's publication, the most recent release of BP was version 3.36, and the most recent release of firmware was version 5.50. If you call Nematron for assistance, you should be able to provide the version numbers you are using. As of this guide's publication, the most recent release of DOWNLOAD was version 1.81. The version number is automatically displayed when you run the program.

To find the version of BP software you have, just look at the top line of the screen:

```
=======================╡ BP Version 3.36│ ====================
```

To find the version of firmware you have, cycle power and watch the display during the power-up self-test:

```
BASIC       V5.50 B
Model IWS-30/40
```

Chapter 5

On-Line Configuration

This chapter describes how to access and use the built-in configuration program. The configuration program allows you to set up the operation of your Workstation. For example, you can set up the communications parameters for your serial port.

Accessing the On-Line Configuration Menu

To access the Configuration Menu, hold down [F1] and [↵] simultaneously during the power-up self-test or press [F3] at any time when BASIC is in the command mode (i.e., no program is running).

Main Configuration Menu

When you enter the on-line configuration program, the Workstation displays the following screen:

```
F1-Clock  F2-Kbd/Dsp
F3-Comm   F4-Utility
```

Selecting a Menu Item To select an item from a menu, simply press the corresponding function key.

Selecting Parameters To scroll from selection to selection without changing anything, press [↑] or [↓]. If you make a change and then press [↑] or [↓], the unit does not record the change and instead goes to the previous or next selection.

Changing Parameters To change a parameter, you must press [+] or [–] to choose the desired parameter, and then press [↵]; the system then displays the next selection.

Exiting Pressing [x] at any time returns to the previous menu (and the Workstation ignores any change on the current screen). To exit the configuration program entirely, press [x].

Clock

Date Format *U.S. (MM/DD/YY), Int'l (DD.MM.YY)*
Selects the format the Workstation uses to print the date. The U.S. format is *MM/DD/YY*, where the month comes first; for example, June 24, 1994 is 06/24/94. The Int'l format is *DD.MM.YY*, where the day comes first; for example, June 24, 1994 is 24.06.94

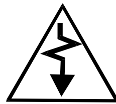
Date Selects the current date; note that the Workstation rejects months and days that are invalid. Because the Workstation has no battery, it loses the date every time you turn off power.

Time Selects the current time; note that the Workstation uses a 24-hour clock, so in the afternoon, you must enter the current time plus twelve. Because the Workstation has no battery, it loses the time every time you turn off power. And while you have power applied, the Workstation's clock is accurate only to within a few minutes every day.

Keyboard/Display

Auto-Repeat *Enabled, Disabled*

Selects whether holding down a key causes it to repeat continually (after a brief delay). We recommend you leave this *Disabled*.



Leave the auto-repeat option *Disabled* if you have assigned any keys to a machine control function. Please consider whether a stuck key could indirectly harm personnel or equipment.

Repeat Delay *1 to 255*

This is the amount of time your operator must hold down a key before it repeats. Note that the resolution of this parameter is 50 milliseconds, which means that a value of 20 equals one second. If you must enable the auto-repeat option, we recommend a repeat delay of 20.

Repeat Rate *1 to 255*

This is the amount of time the unit waits between repeats while a key is held down. The resolution of this parameter is 50 milliseconds, so a value of 2 means the unit repeats every 100 milliseconds, or 10 times each second. If you must enable the auto-repeat option, we recommend a repeat rate of 1 or 2.

Line End Action *None, Auto-CR, Auto-CRLF*

This selection allows you to choose where the cursor moves after it prints to the last column on a line. Following is how these selections control the cursor's movement:

None	The cursor remains at the end of the current line.
Auto-CR	The cursor moves to the first column of the current line.
Auto-CRLF	If the cursor is not on the bottom line, it moves to the first column of the bottom line; if the cursor is on the bottom line, the cursor movement depends on your choice for the Screen End Action selection.

When the cursor is sent backwards, the same concepts apply. For example, when going backwards from the first column, *Auto-CR* moves the cursor to the last column of the current line, while *Auto-CRLF* moves the cursor to the last column of the previous line. Of course, *None* leaves the cursor in the first column. We recommend that you set this parameter to *Auto-CRLF*.

Screen End Action *None, Wrap, Scroll*

If you choose *Auto-CRLF* for the Line End Action selection described previously, this selection allows you to choose where the cursor moves after it prints to the last column on the bottom line. Listed below is how each selection affects the cursor's movement:

None	Cursor remains at the end of the bottom line.
Wrap	Cursor moves to the first column of the top line.
Scroll	The Workstation moves the bottom line to the top and clears the bottom line. Then it moves the cursor to the first column of the bottom line.

When the cursor is sent backwards, the same concepts apply. For example, when going backwards from the first column of the top line, *Wrap* moves the cursor to the last column of the bottom line, while *Scroll* moves the top line down and places the cursor at the last column of the top line (which is now blank). Of course, *None* leaves the cursor in the first column. We recommend that you set this parameter to *Scroll*.

Cursor *None, Block, Underscore*

Selects the cursor type. We recommend that you choose the Block cursor.

Communications

Baud Rate *110, 30/400, 600, 1200, 2400, 4800, 9600, 19200*
Selects the speed at which this port transmits and receives. We recommend 9600, but in any event you must ensure that the device connected to this port operates at the same speed.

Parity *None, Odd, Even*
Selects whether the Workstation sends and receives an extra bit that helps guard against lost bits. Selecting *None* disables this feature, while *Even* specifies that of the bits received, an even number of them must be high (and conversely for *Odd*). We recommend you enable parity if the data communicated is critical. In any event, this must match the setting of the device connected to this port.

Data Bits *7, 8*
Selects the number of data bits in each byte transmitted. This must match the setting of the device connected to this port. The combination of 7 data bits, no parity, and 1 stop bits is invalid. In that instance, we recommend you select 2 stop bits instead.

Stop Bits *1, 2*
Selects the number of stop bits transmitted after each byte. This must match or exceed the setting of the device connected to this port. In other words, selecting *2* always works, but selecting *1* usually works. The combination of 7 data bits, no parity, and 1 stop bits is invalid. In that instance, we recommend you select 2 stop bits instead.

Transmit Handshake *None, CTS, XON/XOFF*

Selects the type of "handshaking" that the Workstation respects when transmitting. Selecting *None* tells the Workstation to transmit immediately. This works fine if the other device's receiver is *always* ready to receive. Selecting *CTS* tells the Workstation to transmit only if its CTS input is asserted. Choose this if the other device has an output that it asserts when its receiver is available. This is often called "hardware handshaking."
Selecting *XON/XOFF* tells the Workstation to stop transmitting when it receives an XOFF (ASCII code 19, or [Ctrl]-S) and to resume when it receives an XON (ASCII code 17, or [Ctrl]-Q). This is often called "software handshaking" and is typically used only for Terminals.

RTS Control *Always On, On During Xmit, On to Receive, On at Xmit*

Selects the function of the RTS handshaking line. In most RS-232 applications, you should select *Always On* and connect RTS to the CTS input of the Workstation or the other device.
For virtually all RS-422 and RS-485 applications, you should select *On During Xmit*. The Workstation enables its RS-422/RS-485 transmitter only when RTS is on, which is crucial when there are multiple transmitters on the same pair of wires.
Choosing *On to Receive* is the same as choosing *RTS* for the Receive Handshake selection that follows.

Receive Handshake *None, RTS, XON/XOFF*

Selects the type of "handshaking" that the Workstation asserts when receiving. This is especially useful to prevent your host from overflowing the Workstation's receive buffer, which can cause weird display problems. If the other device supports "hardware handshaking" on its transmitter, which usually means that it doesn't transmit unless its CTS input is asserted, you can select *RTS* and connect the Workstation's RTS output to the other device's CTS input. (This is the same as choosing *On to Receive* for the RTS Control selection previously described.)

If the other device supports "software handshaking," you can select *XON/XOFF* so that the Workstation matches the other device.

Convert Parity Error *Enabled, Disabled*

This selects whether the Workstation automatically translates characters received with the wrong parity into some other character. Typically, you would assign a character such as "~" that is normally not displayed, so that the operator knows that there was a parity error.

Parity Error Char *0 to 255*

Selects the ASCII code of the character returned in place of characters received with incorrect parity. You should read the description above under "Convert Parity Error" for more information.

Utility

Selecting the Utility function from the main configuration menu brings up the following menu:

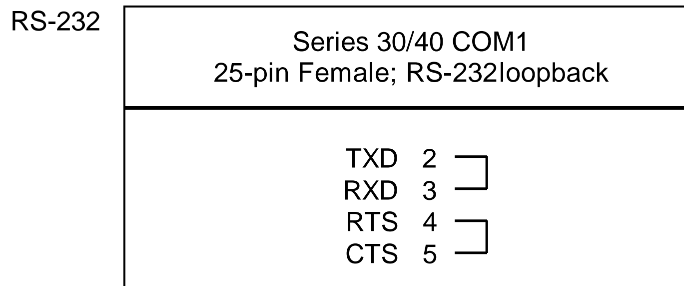
```
    Utility Menu
F1-Test    F2-Upload
```

Test

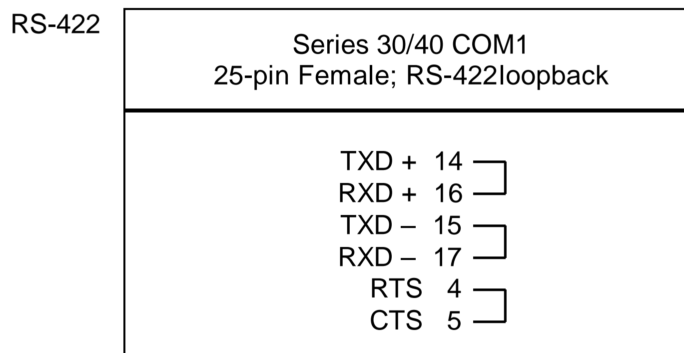
This section describes how to perform a "loopback" test on the Workstation's COM1 port so that you can check for a hardware failure. This test is useful either to confirm or rule out that communication problems are related to hardware problems.

Before you begin this test, you must plug a "loopback" connector into the COM1 port. This connector is a female DB25 connector that you have modified according to one of the following diagrams:

RS-232



RS-422



To perform this test, you must first gain access to the on-line configuration menu by cycling power and pressing [F1] and [↵] simultaneously during the power-up self-test. To run the test, you first press [F4] and then [F1].

After you start the test, the Workstation sends test data out through the transmitter and checks for identical data coming back through its receiver. If everything works properly, the Workstation displays the following screen:

```
Test Port:
COM1: Pass
```

If the COM1 port does not work properly, the Workstation displays one of the following messages:

Break	Indicates that a “break” character was received; probably indicates a failed transmitter.
Compare	Indicates that the Workstation received different characters than expected; the probable cause is a hardware failure.
Framing	Indicates that the Workstation received a character with a framing error; the probable cause is a hardware failure.
No CTS	Indicates that the CTS input appears not to be asserted; this is caused either because the jumper between RTS and CTS is missing from the loopback connector or because the RTS output or CTS input has failed.
Overflow	Indicates that the Workstation received too many characters; the probable cause is a hardware failure.
Overrun	Indicates that the Workstation was unable to process characters as fast as they arrived; the probable cause is a hardware failure.
Parity	Indicates that the Workstation received a character with a parity error; the probable cause is a hardware failure.
Timeout	Indicates that the Workstation received no characters at all; this is caused either because the jumper between RXD and TXD is missing from the loopback connector or because the transmitter or receiver has failed.

Upload

This allows you to enable uploading of new firmware to the Workstation. After selecting this option, you can follow the instructions on page 5-2 to download firmware.

```
Ready to accept
firmware download.
```

Chapter 6

Service

This chapter describes service procedures and helps you troubleshoot some of the simple hardware problems that can occur. The material in this chapter is also contained in DOC-IWS-271, *Series 30/40 Installation Guide.*

Changing the Fuse

The Series 30/40 Workstation requires one 1/2 Amp Slo-Blo Pico™ fuse (Littelfuse® part number 473.500); Nematron offers fuses under part number COS-FUS-30 which contains ten fuses.

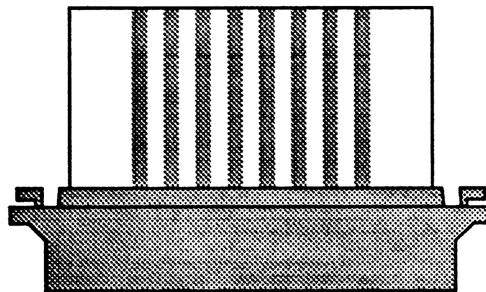
The fuse holder is accessible only by removing the back cover and printed circuit board. Follow these steps to replace the fuse:

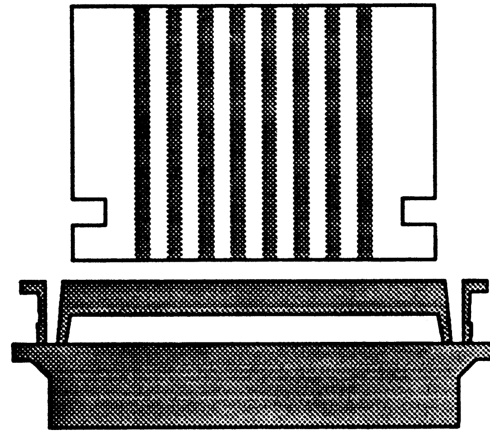
1. Disconnect power to the unit. Disconnect any communications cable plugged into COM1. Remove the unit from the panel.

2. Correct the condition (usually a power supply overload) that caused the fuse to blow.

3. Remove the four screws that hold on the back cover and slip off the back cover.

4. Locate the ribbon of plastic that connects the keyboard to the printed circuit board, unlatch the black cap of the keyboard connector, and remove the keyboard tail. The following two illustrations show the keyboard connector in the latched and unlatched positions:





5. Remove the four standoffs that hold on the printed circuit board and remove it as well.

6. Locate fuse F1 near the power supply terminal block. Using a continuity tester, check the fuse to make sure it's the problem.

7. Using needle-nose pliers, grab the center of the fuse and pull straight up. Install a new fuse.

8. Install the printed circuit board back on the posts; be sure that the two spacers between the display and the printed circuit board remain in place.

9. Insert the keyboard tail into the keyboard connector, and latch the connector.

10. Install the standoffs, replace the back cover, and replace the screws.

Memory Write Disable

After you download your firmware and application file, you can disable any further changes to the firmware by moving a jumper inside your unit.

To move this jumper, follow these instructions:

1. Disconnect power to the unit. Disconnect any communications cable plugged into COM1. Remove the unit from the panel.

2. Remove the four screws that hold on the back cover and slip off the back cover.

3. The jumper is located on the component side of the printed circuit board near the LCD display module. The jumper is labeled "E2" and has two positions: "RO" and "RW". The "RO" position is "Read-Only"; moving the jumper to that position protects the memory from any intentional or accidental changes. The "RW" position is "Read/Write"; moving the jumper to that position allows you to download new firmware or a new application.

4. Replace the back cover and the screws.

Troubleshooting

Virtually all apparent problems are caused by improper communications connections or inadequate grounding.

Problem noted	Possible cause(s)	Remedy
No response to some keypresses	Loose keyboard connector; faulty keyboard	Unlatch keyboard connector, ensure keyboard tail is fully seated, and re-latch connector; return unit to factory for new keyboard.
Firmware download reports that Flash is write-protected, empty or invalid	Write-protection jumper is in "RO" position; using old version of DOWNLOAD; faulty firmware	Move write-protection jumper to "RW" position; use V1.6 or later of DOWNLOAD; or return unit to factory for new firmware.
Communications problems	Bad cable; port failure	Verify cable; use proper wire for RS-422; check serial port with our loopback test as described in Chapter 5.
Unit resets randomly or exhibits other intermittent problems	Loose or shorted power cable; short in communications cable	Check, repair and replace cable(s).
	Workstation not connected to earth	Connect unit to earth ground.
	Voltage potential between earth grounds of Workstation and other devices connected to its communications ports	Connect all devices to the same earth ground.
	Loose integrated circuit inside	Open unit, remove board and press chip into socket.
	Faulty Workstation power supply	Return unit to factory.
	Low line power voltage	Raise voltage.
	Faulty Workstation	Return unit to factory.

Chapter 7

Concepts

Command/Run Modes

BASIC operates in two modes, the Command, or direct mode, and the Run, or deferred mode. Some statements can only execute when BASIC is in the Command mode; others can execute only in the Run mode, while the remainder can execute in both modes.

In Command mode, BASIC immediately executes one or more statements after you press [Enter]. Examples of statements that execute only in Command mode are NEW and DEL.

In Run mode, BASIC executes numbered program statements. Examples of statements that can execute only in RUN mode are STOP and RESUME.

Console

Your IBM-compatible PC acts as a programming "console" when connected to the Workstation's serial port. Through the console, you can enter commands and edit your program.

The default Workstation serial port that connects to your console is COM1; you can assign a different serial port as the console port with the REACT C command. For example, REACT C2 assigns port 2 as the console. See the description of the REACT command for more information.

Ports

You must connect your Workstation's COM1 port to your PC when you're programming, and to some other device when your program is running. This can be awkward, so we recommend that you consider using an IWS-117 or IWS-127 for program development.

When communicating, the display and keyboard are also considered to be a port. The following table summarizes the port names and numbers:

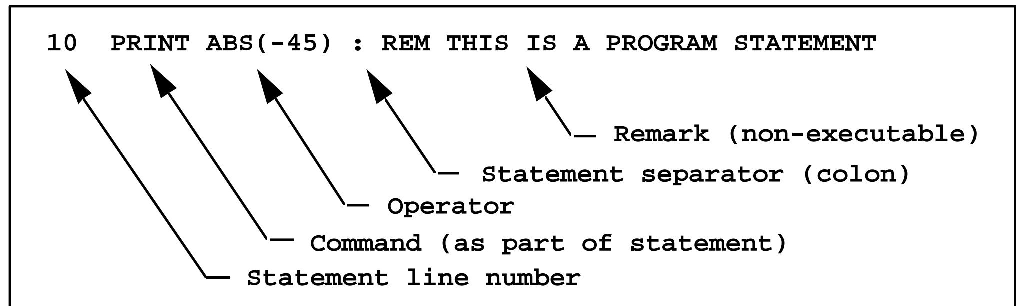
Name	Number	Description
COM0	0	Keypad and display
COM1	1	COM1 serial port

Statements

A BASIC program consists of numbered lines. Each line ends with a carriage return and contains one or more statements. Each statement on the same line must be separated by a colon (:).

The following rules apply to BASIC statements:

- Every line in a program must have a statement line number ranging between 0 and 65535. A good programming practice is to number line numbers by 10 (i.e. 10, 20, 30, etc.), because BASIC doesn't care if there are gaps between line numbers.
- A statement number can appear only once in a program.
- You do not have to enter statements in numerical order, because BASIC automatically stores them in ascending order.
- A statement may contain no more than 250 characters.
- BASIC ignores blanks (spaces) except between quotation marks.
- A single line can contain more than one statement; a colon (:) separates each statement.



Numbers and Constants

There are two types of numbers that BASIC can handle: integers and floating point numbers. Integers range from -65,535 to 65,535, while floating point numbers range from ±1E -127 to ±.99999999 E + 127.

Floating point numbers contain eight significant digits; BASIC continually truncates results that return more than eight significant digits. BASIC can accept and display numbers in three formats:

Format	Example
Decimal	34.98435
Hexadecimal	0A6EH
Scientific notation	1.2745 E+3

Hex numbers must begin with a digit between 0 and 9; to enter a hex number that starts with a letter (A through F), you must precede the number with a 0. For example, you must enter the hex number A00H as 0A00H. When a BASIC operation requires an integer, BASIC either rejects any number that exceeds the range and returns an error or truncates any fractional portion so it fits the integer format.

The word "constant" simply refers to a group of digits or a string of characters. For example, 123 is a constant, while A is a variable. Constants can range in value from ±1E -127 to ±.99999999 E + 127.

Variables

Variables are named locations that hold changeable values. Different types of variables exist for different types of values; for example, small integers, fractions or large numbers, and strings of characters. Also, a variable can hold one value ("scalar variable") or many values ("array variables").

Each type of value that a variable can hold is listed in the table below:

Variable Type	Format	Example
Floating point	Variable name	A
Integer	Variable name followed by "%"	A%
String	Variable name followed by "$"	A$

The first three variable formats identify different variables, even if the preceding letters are the same. For example, the variable names A, A%, and A$ all refer to *different* variables.

Array variables are different from scalar variables of the same name; for example, A( ), A, A%( ), A%, A$( ) and A$ all refer to different variables!

Scalar Variables

Simple variables that represent a single value are called "scalar" variables. For example, the variable "A" is a scalar variable that represents a single number.

Array Variables

BASIC supports single-dimensioned variable "arrays," where one variable name refers to several separate variables, each identified by a number that follows the name. The format of an array variable is "A(n)" where "n" is a number that refers to a specific variable in the array. Chapter 11's description of the DIM statement contains additional information about array variables.

Memory Allocation

BASIC allocates RAM memory space to variables in a "static" manner. This means that each time the program refers to a new variable, BASIC allocates 8 bytes of memory to that variable (strings require more; see the description of VARPTR in Chapter 11).

A BASIC program cannot de-allocate memory allocated to specific variables. For example, if BASIC executes a statement like A = 4, you cannot tell BASIC later that the variable A no longer exists. The only way to clear the memory allocated to variables is to execute a CLEAR statement, which eliminates all variables.

Note BASIC evaluates references to scalar variables faster than array variables. To improve your execution time, use scalar variables for intermediate results, then assign the final result to an array variable.

Variable Names

A variable is a named data location that a BASIC program can examine and change. A variable name must start with a letter, but it may include both numbers and letters. Only the first two characters of the variable name are significant. For example, "A" is a valid variable name, as is "A1234;" however, BASIC considers "A1234" to be the same as "A1."

Variable names cannot include any words that BASIC reserves for its own use. Among the reserved words are all of the statements listed in Chapter 11. For example, TOP is not a valid variable name because it contains the word TO, which BASIC uses in a FOR statement. As another example, STOP is not a valid variable name because it is a BASIC statement.

Floating point Variables

Unless otherwise indicated, variables refer to floating point numbers. For example, A, A1, BAD, and STARTING are all valid floating point variable names. Floating point numbers range in value from ±1E -127 to ±.99999999 E + 127, but with a resolution of only 8 digits. For example, when BASIC multiplies 1.2345678 by 11, it truncates the result of 13.5802458 to 13.580245.

Integer Variables

Integer variable names end with a % character; for example, A% is an integer variable, not a floating point variable. Integer numbers range in value from -65,535 to 65,535. (Integers in most BASICs have a range of only -32,768 to 32,767.)

Integers offer a speed advantage; most operations involving integers execute as much as 10% faster than the same operations using floating point numbers. Unlike most BASICs, Series 30 BASIC does not offer any savings in memory consumption by using integer variables instead of floating point.

String Variables

String variable names end with a $ character; for example, A$ is a string variable. String variables have a default length of 10 characters; your program can set up a different default string length for all strings as well as specify a length of up to 254 characters for any specific string. Consult the description of SDIM in Chapter 11 for more information.

Unlike most BASICs, Series 30/40 BASIC allocates memory to strings in a "static" manner. Once you create the string, either with the SDIM statement or simply by referring to it, you cannot change its maximum length.

Built-in Variables

Built-in variables contain values that BASIC assigns itself, depending on the circumstances. For example, ERR contains the number of the last error that occurred. Note that all characters in the variable name are significant for built-in variables. In other words, ER is not the same variable as ERR.

Operators and Expressions

This section describes how to combine multiple mathematical operations into a single expression.

Operators

Operators manipulate one or two operands and return a result. Typical dyadic (two-operand) operators include add (+), subtract (-), multiply (*), and divide (/). Typical unary (one-operand) operators are exponentiation (^), SIN (sine), COS (cosine), and ABS (absolute).

Expressions

An expression is a mathematical formula that includes a combination of operators, constants, and variables. Expressions can be simple or complex. A "stand-alone" variable such as "A" is an expression in its simplest form. A complex expression might be SIN(A)*(SIN(A)+COS(A))*COS(A)/2.

Relational Expressions

Relational expressions compare two expressions and return a true/false result. The numeric value of true is 65,535; the numeric value of false is 0. A relational expression can be any arithmetic expression, although typical relational expressions use the relational operators shown below:

=	Equal
<>	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The relational operators also work with strings; string1 is considered "less than" string2 if it is either shorter than string1 or if the first character that doesn't match has a lower ASCII value. For example, "TO" is less than "TAP" and less than "TS."

Memory Usage

Page C-13 of the Appendix contains a drawing that shows how BASIC uses the Workstation's memory. You may want to refer to this drawing as you read the following text.

While you're creating a new program, BASIC stores your program at the bottom of available RAM; as your program increases in size, BASIC uses RAM memory at increasing addresses. While you're editing your program, the entire RAM is available.

During program execution, BASIC stores normal variables starting at the top of available RAM (as specified by MTOP, whose default value is the highest RAM address available); as you add variables, BASIC stores those variables at decreasing addresses in memory. BASIC stores array variables starting at the end of your program and works up.

BASIC can run out of memory during editing if your program expands beyond the available RAM. If this happens, then your program is certainly too big, because BASIC allocates memory for variables only while your program is executing. BASIC can also run out of memory while your program is running, if it detects that normal variable memory has overlapped your program or array variable memory.

In either case, BASIC issues a MEMORY FULL error and you must slim down your application.

After you have completed your application, you must save it in the Workstation's permanent memory by copying it to ROM. When the Workstation powers up, it copies the program to RAM and runs out of RAM. Please see the description of the ROM command in Chapter 11 for complete information.

Chapter 8

How to Use BP Software

IWS-SETUP-BP-30, the setup package for the Series 30/40 Workstation, includes a terminal emulation program called BP, along with several BASIC programs that contain interesting examples and useful utilities (described below). This chapter shows how to use BP to create, upload, modify, and download BASIC programs.

Starting Up BP

You should make a copy of your BP diskette and save the original as a backup.

Disk Files

In addition to the BP software itself, BP.EXE, the BP diskette contains the following files:

BP.CFG	Contains information about your configuration, including window colors, baud rate, and so on. You can change the configuration information by following the instructions on page 9-6.
IHEX30/40.LST	Allows you to dump the contents of memory to a disk file using the Intel hex format. This is especially useful for OEMs who use EPROM program memory.
BAR30/40.LST	Demonstrates the use of the display's programmable characters by displaying a bargraph with 1% resolution.

To run BP, you must first install the software as described in Chapter 4. After you install the software, type *BP* and press [Enter]; your PC displays the following:

```
WELCOME TO BP

BP is  communications software for the Series 30/40 Workstation with BASIC.

Version 3.36, dated November 17, 1993.
Copyright 1993, Nematron Corporation, Ann Arbor, Michigan.
All Rights Reserved.

(press any key to continue)
```

After you press any key you see the following:

```
================╡ BP Version 3.36 │COM1: 9600 N81│=========




1 Help 2 Edit  3 Cmd   4 DOS   5 Recv  6 Send  7 DCap  8 Dual  9 Setup 10 Exit
```

The normal function of BP is to emulate a "dumb" terminal connected to the Workstation. In other words, when you press a key on your PC's keyboard, the PC sends that key to the Workstation, and when the Workstation sends a character to the PC, the PC displays that character on the screen.

BP also has additional functions that allow you to send disk files to the Workstation, receive program listings from the Workstation and store them on disk, and edit a line in the BASIC program stored in the Workstation.

The display above is the standard terminal emulation screen; pressing one of the function keys [F1] through [F10] causes BP to perform one of its additional functions.

Starting up the Workstation

Before you can create, change, transmit, or receive programs, you must first connect a cable between your PC and your Workstation and turn on the Workstation. The cable is included with your IWS-SETUP-BP-30 package and connects between your PC's COM1 serial port and the Workstation's COM1 port. (To use your PC's COM2 port, you must change BP's configuration as described in this chapter.)

If your PC has a 25-pin serial port, you will need to purchase a standard 9-to-25 pin adapter to use with the cable that comes with IWS-SETUP-BP-30.

Turn on the Workstation and wait for it to complete its self-test. At the end of the test, it displays the following message:

```
 Welcome to V5.50
  Series 30/40 BASIC
```

When the Workstation completes its self-test, it checks to see if it has an "auto-start" program in memory. If it does, it runs the program. If not, then it sends the following message to your computer:

```
NEMATRON CORPORATION
Series 30/40 BASIC
  Version 5.50

READY - RAM 1
>
```

The ">" symbol indicates that the BASIC interpreter is waiting for you to enter a command. If you don't see that symbol, the Workstation is probably executing a program; try holding down

[F1] and [↵] simultaneously to halt the program (see Chapter 9 about Halting a Program for more information).

Summary of BP Functions

[F1]	Display Help screen
[F2]	Edit a line in a BASIC program.
[F3]	Edit a recent command.
[F4]	Temporarily exit to DOS.
[F5]	Receive current program and save on disk.
[F6]	Send program from disk to Workstation.
[F7]	Capture data from Workstation and save on disk.
[F8]	Toggle history buffer on and off.
[F9]	Set up communications and screen parameters.
[F10]	Exit BP and return to DOS.
[Alt]-H	Toggle hex mode on and off.
[Alt]-M	Enter macro editor.
[Alt]-S	Turn sound on/off

Creating a Program [F2]

Although you can write a program "on-line" by typing directly into the Workstation, the Workstation's editing capabilities are limited. To simplify this task, you should use a word processor that creates "TXT" (text) files. Once you finish writing a program, download it to the Workstation by following the instructions later in this chapter. Once you download your program, you can type simple changes and additions directly into the program.

Line Editing

BP has a convenient single-line editor that allows you to change a line in your program without retyping the entire line. To use the editor, press [F2] to call up the following:

```
========================| Edit |=====================| ins |=



Enter line number:
```

You can select a line to edit by typing in its line number and pressing [Enter]. BP then retrieves the line from the Workstation and displays it on the PC's display. At that point, you can change the line and then press [Enter] to send the line back to the Workstation. Listed below are some special editing keys and their corresponding functions:

[Enter]	Send the changed line to the Workstation.
[Esc]	Exit the edit function and leave the line unchanged.
[Tab] or [Ctrl]-F	Move to the next word.
[Shift]-[Tab] or [Ctrl]-A	Move to the previous word.
[Backspace]	Move left and delete the character under the cursor.
[Insert]	Toggle between insert and overtype modes; an indicator at the top right corner of the edit window displays "ins" when in insert mode and "ovr" when in overtype mode.
[Delete]	Delete the character under the cursor.
[Home]	Move the cursor to the start of the line.
[End]	Move the cursor to the end of the line.
[←], [→], [↑], [↓]	Move the cursor one position to the left, right, up, or down.

Command Editing

BP allows you to edit any line you recently typed, including both direct commands and program lines. To use the command editor, press [F3] to call up the following display
:

```
========================| Edit |======================| ins |=



1 Help  2 Edit  3 Prev  4 Next  5 Clear  6        7        8        9        10
```

When you enter the command editor, BP displays the last line you typed. You can select a previous line by pressing [F3] or you can create a new line by pressing [F5].

Sending a Program to the Workstation [F6]

To send a program stored on your personal computer's disk to the Workstation, press [F6] on the PC's keyboard to display the following screen:

```
====================== FILE TRANSMIT ======================

    Transmit File Name:
```

Enter the filename of your BASIC program, including the drive location and/pathname. For example, if you are uploading MYPROG.TXT which is located on a disk in drive A, enter the name of the file as *A:MYPROG.TXT*.

If BP finds your file, you then see the following:

```
====================== FILE TRANSMIT ======================

    Transmit File Name :  A: MYPROG.TXT
    Transmitting Line  :
```

BP displays a running count of the number of lines transmitted. If this number does not continually increase, there is a problem with your software configuration or hardware setup. We recommend you use the statement *OPEN "COM1:IB,CE,LF,TX0"* to set the communications parameters of your console port. You can abort the transmission by pressing [Esc].

Receiving a Program from the Workstation [F5]

After changing a program within the Workstation, you probably want to capture the program on disk. To do this, press [F5] to see the following screen:

```
====================== FILE RECEIVE ======================

    Receive File Name :  A: MYPROG.TXT
    Lines received    :
```

Enter the filename of your BASIC program, including the drive location and/or pathname. For example, if you are uploading MYPROG.TXT and you want to place the file on a disk in drive A, enter the name of the file as *A:MYPROG.TXT*.

BP displays a running count of the number of lines received. If this number does not continually increase, there is a problem with your software configuration or hardware setup. We recommend you use the statement *OPEN "COM1:IB,CE,LF,TX0"* to set the communications parameters of your console port. You can abort receiving by pressing [Esc].

Configuring BP [F9]

Pressing [F9] allows you to view and change the current settings for BP. When you enter the Setup function, you see the following screen. The function key legend at the bottom of the screen indicates the configuration choices available to you; to exit any of the configuration choices, press [Esc].

```
===================BP Setup  Menu====================
COLOR CONFIGURATION            | SERIAL PORT CONFIGURATION
                               |
TERMINAL WINDOW                | Port              COM1
HISTORY WINDOW                 | Baud Rate         9600
EDIT WINDOW                    | Parity            None
FILE TRANSFER                  | Word Length       Eight
HELP MENU                      | Stop Bits         One
ERROR MESSAGES                 | Handshaking       Pff

This text sample shows how BP  | DISPLAY CONFIGURATION
displays both the text and     |
background colors for the      | Split Row         10
window                         | Split Window      Off
you have currently selected.   |
   1 HELP 2  3  4  5  6  7 SAVE   8 Rest   9 Next   10
```

Color Configuration

If your PC has a color monitor, you can configure the colors of both the text and background for all of the windows that BP displays.

To configure window colors, press [F9] until that section of the screen is highlighted. Then press [↑] or [↓] to highlight the current window selection. The bottom left corner of the screen shows a sample of the current text and background colors for the selected window. You can change the text colors by pressing [F2] and [F3], and you can change the background colors by pressing [F5] and [F6].

Serial Port Configuration

You can set up the PC's serial port that BP uses to talk to the Workstation by pressing [F9] until that section of the screen is highlighted. Press [↑] or [↓] to select a communications parameter, and press [←] or [→] to choose among the various selections for each parameter. We recommend that you avoid using 19,200 baud for downloading, because it is not reliable.

Display Configuration

You can set up parameters for the History window by pressing [F9] until that section of the screen is highlighted. The Split Row parameter selects the size of the History window, and the Split Window parameter selects whether the History window is on or off when BP first begins. To choose between the parameters, press [↑] or [↓]. To choose among the various selections for each of these parameters, press [←] or [→].

Save and Restore Defaults

After you change your configuration settings, you can press [F7] to save them on disk so that BP remembers your settings the next time you use BP. (BP stores your configuration settings in the file titled BP.CFG.) If you want to restore your configuration settings to the factory selection, press [F8].

Additional BP Functions

DOS Gateway

You can exit BP temporarily to perform DOS functions or run another program and then return to BP where you left off. To exit BP, press [F4]; to re-enter BP, type *EXIT* and press [Enter] after the DOS prompt. If the file COMMAND.COM is not in the current directory or path, then BP displays the error message CAN'T SPAWN.

Data Capture

There may be occasions when you want to capture data on disk that the Workstation generates. To initiate data capture, press [F7] to display the following screen:

```
===================== DATA CAPTURE ======================

     Data Capture File Name:
```

After you enter the filename, the display changes to show the number of bytes (characters) that BP has received so far.

In order to conclude the data capture function, you can either press [Esc] or the Workstation can send ASCII code 1Ah, which is equivalent to a [Ctrl]-Z. You can do this with the statement *PRINT CHR$(1Ah)*.

History Window

In the terminal window, BP displays the last 23 lines that it received from the Workstation. BP saves the last several hundred lines, however, and you can review these by displaying and scrolling through the history window. To display the history window, press [F8]; to cancel the history window, press [F8] again.

To move around in the history window, press [Page Up], [Page Down], [↑], [↓], [Home], and [End].

Hex Display

BP can display every character it receives in “hex” format instead of the normal ASCII format. To select the hex mode, press [Alt]-H; when BP is in the hex mode, it displays the word “HEX” in the upper left corner of the screen. To cancel the hex mode, press [Alt]-H again

Keyboard Macros

BP has a “keyboard macro” capability that allows you to type a string of characters by pressing [Alt] and a single digit simultaneously. To set up a keyboard macro, you must press [Alt]-M to enter the macro editor.

To change a macro or enter a new macro, press [↑] or [↓] to select the macro you want to edit. Then press [←] or [→] to position the cursor, after which you simply type the keystrokes you want in that macro. Keys such as [Enter] are displayed as odd symbols such as a musical note.

To save your changes, press [F8]. To cancel any changes to your current macro, press [F5]; to cancel all changes, press [F9]. To erase the current macro, press [F3].

To use a macro, hold down [Alt] while pressing a digit. For example, [Alt]-0 invokes the first macro.

As delivered, several of the macros are already set up to perform useful functions. For example, [Alt]-0 opens COM0 according to our recommended settings. Of course, you can change these macros too.

Troubleshooting

If BP is unable to complete a function that you request, it displays an error message which usually appears in the middle of your screen. This section describes those error messages and offers possible solutions:

Can't open serial port

When you start running BP, it checks for availability of the serial port (COM1 or COM2) assigned for its use. The factory setting is COM1, but you can access the setup screen to change the serial port to COM2 by pressing [F9] while BP is running.

If BP's serial port appears unavailable, it displays the message "Can't open serial port" on the bottom of your screen and terminates execution.
There are three possible causes for this message:

1. BP may be set up to use COM2 when the PC has no COM2 port. This usually occurs when you set up BP to use COM2 on a PC with two serial ports, and then copy both BP.EXE and BP.CFG to another PC.

   Because BP immediately terminates when it is unable to communicate, you cannot access the setup function to change the serial port. Instead, you must delete BP.CFG from your disk and then run BP again. When BP fails to find BP.CFG, which contains its setup information, it generates a new setup file that sets the serial port to COM1.

2. You computer may already utilize the selected serial port for some other function, such as a remote file server. In that case, you must prevent your PC from logging on to the file server when it boots; this typcially requires you to chagne your AUTOEXEC.BAT file and re-boot.

3. Your serial port may be faulty; you'll have to repair it.

Can't spawn

When you press [F4] to exit BP temporarily in order to perform a DOS function, BP tries to run the COMMAND.COM file. If BP is unable to find COMMAND.COM, then it displays the message "Can't spawn."

This problem usually occurs because your path does not include the directory that contains COMMAND.COM. The usual solution is to change your path specification in your AUTOEXEC.BAT file to include the COMMAND.COM directory. An alternative is simply to copy COMMAND.COM to the same directory that contains BP.EXE.

Time-out

There are a few instances where BP waits for a response from your Workstation; if the Workstation does not respond within a reasonable period of time, BP displays the message "Time-out."

This usually occurs if you press [F2] or [F5] while you're in the middle of typing a line, instead of pressing the key at the ">" prompt.
The time-out error can also occur if you are downloading and the Workstation's console port (usually COM1) is *not* opened with the IB parameter. We recommend you use the statement *OPEN "COM1:IB,CE,LF,TX0"* to open your console port.

Garbled Programs or Downloading Problems

Make sure COM1 is opened properly: OPEN "COM1:IB,CE,LF,TX0". You can press the [F4] key on the Workstation to force COM1 to the proper default settings.

Chapter 9

Application Suggestions

This chapter describes some of the Workstation's special capabilities, including keyboard functions, auto-starting and terminating programs, protecting programs and data, controlling the display and LEDs, performing functions at regular time intervals, and communicating with intelligent devices.

Special Keyboard Functions

Halting a Program; [Ctrl]-C

Unless you provide otherwise, pressing [Ctrl]-C terminates execution of a running program. You can disable [Ctrl]-C entirely by opening each port without the CE parameter, or you can set up your program to handle [Ctrl]-C as an error via your own error-handling routine (enabled with the ON ERROR GOTO command).

The Workstation senses [Ctrl]-C from either the current input port or the Workstation's keyboard, provided both are configured with the "CE" parameter. The Workstation ignores a [Ctrl]-C received on a serial port that is not the current input port, regardless of that port's CE parameter.

If [Ctrl]-C is not enabled, then the only way to stop execution of your program is to cycle power. If your program is in RAM memory, however, any changes since you last copied the program to ROM will be lost. Moreover, if you have set up your program to run automatically with the REACT command, then you're absolutely stuck. You must download new firmware in order to erase your program.

On-Line Configuration; [F3]

You can select almost all of the Workstation's operating parameters either through the on-line configuration menus or through the OPEN statement. Typically, you should use the OPEN command to change parameters.

You can enter the Workstation's on-line configuration menus by pressing the [F3] key when the unit is *not* running a program. (Pressing [F3] won't work if another keypress remains in the buffer; you may have to press [F1]-[↵] or cycle power to empty the buffer, and then press [F3] again.) You should consult Chapter 6 for details.

Restore COM1 Defaults; [F4]

When BASIC is in the command mode (no program is running), you can press [F4] to reset the COM1 communication parameters to their default values of 9600,8,N,1. This simplifies program development because you can easily set COM1 to known parameters after your program stops running.

Program Development Techniques

We strongly recommend you develop BASIC programs on an IWS-117 or IWS-127 because those models offer three serial ports, which usually leaves one available for connection to your PC.

If you use the Series 30/40 Workstation, you must continually switch COM1 between your PC and another device. This can be inconvenient, not only because of the continual need to switch cables, but also because your PC and your other device may require different communications parameters. In that case, you need some way to switch your program back and forth between two sets of communications parameters.

In any event, you still face a debugging problem: if your program terminates with an error message, BASIC sends that error message to COM1, which will probably be connected to another device, so you won't see the message. You can force the Workstation to display the error message on its display instead of sending the message to COM1, but to do that you must set up your program to run automatically with the REACT R command, copy your program to ROM, and then cycle power to the Workstation. Once your program is in ROM, the only way to change it is to download firmware again.

At any time that your program isn't running, you can press [F4] on the Workstation's keypad to restore the COM1 communications parameters to their default values.

We suggest you implement an ON ERROR handler so that any error causes the program first to change the COM1 communications parameters to the default values needed for your PC, and then to terminate execution. Your program should also open COM0 with the CE parameter so you can press [F1]-[↵] at any time to halt program execution.

Unfortunately, your error-handler only sees errors that are "recoverable"; there are several non-recoverable errors that cause program execution to terminate immediately, without regard to the ON ERROR command. As a result, we suggest you postpone communications testing until late in your project and wait until then to implement an error handler.

Following are sections of a model program that switches COM1 between two sets of communications parameters:

```
10 ON ERROR GOTO 10000
20 OPEN "COM1:IB,ED,19200,TD100,CS0"
30 OPEN "COM0:IB,CE,BE,LE,LF,CL,SC,SS0"
.
.
.
10000 OPEN "COM1:IB,CE,LF,TX0,TD0"
10010 END
```

Storing Programs in Permanent Memory

You can only modify your program when it is in RAM, but when you're finished developing your program, you must copy the program to the Workstation's permanent memory.

The Workstation stores your program in the same Flash memory where it stores the BASIC firmware itself. When the Workstation powers up, it copies the program from Flash to RAM.

Because Flash cannot be changed without completely erasing every location, you must download new firmware if you want to make any changes to a program that you have already copied to Flash.

To copy a program to Flash, you would set up your REACT command and then use a command in the form "ROM = RAM 1", which copies the first program in RAM to Flash and makes the current REACT setting permanent. See the description of the ROM command in Chapter 10 for more information.

Auto-Starting Programs

You can set up the Workstation so that after the power-up self-test, it automatically executes the program stored in its memory. To do this, you must use the "REACT R" command *before* you copy your program to the Workstation's permanent Flash memory. This command tells the Workstation to run your program automatically. See the description of the REACT command in Chapter 10 for a complete description of the functions the Workstation can perform after power-up.

Program Protection

Many OEMs and systems houses hope to prevent their customers from modifying their programs, plus prevent their competitors from copying their programs. To do this, you must both prevent your program from halting plus lock it in the run mode if it does halt.

Most programmers set up an error-handling routine using the ON ERROR GOTO capability in order to "trap" errors. When any non-fatal error occurs, including [Ctrl]-C, the Workstation transfers program execution to the error routine. At that point, the error routine should provide a way to recover gracefully from any error condition.

However, if a fatal error such as SYNTAX ERROR or OUT OF MEMORY occurs, program execution terminates. (Fatal errors usually indicate serious programming errors, so they usually occur during the debugging process instead of after final installation.) In case of a fatal error, you can set up your program to re-run automatically by using the "P" parameter in your REACT command. For example, *REACT R,P* sets up your program to auto-start and protects it by locking it in the Run mode.

Note If you lock your program in the run mode, you should provide a secret way to terminate program execution so that you can provide field service. If your program is locked in the run mode, you can terminate execution only by erasing the program.

To erase the program, you must force the Workstation into the firmware download mode and then download new firmware. See Chapter 4 for instructions about downloading firmware.

Preserving Data During a Power Loss

Some applications require permanent storage of data such as recipe information or production statistics.

If the data never changes, then one simple way to store it permanently is by embedding data in your program with DATA statements and copying that data to variables by using READ statements. See Chapter 10 for details.

If you want to store data permanently but need the capability to change it while your program is running, you can use the ST@ command to write variables to memory and LD@ to retrieve them later. This technique has a significant limitation: although your program can write data to a permanent memory location, your program cannot change this data later. You can implement a technique to save changed data in unused locations that are at progressively higher addresses in memory, but eventually you could run out.

The reason for this limitation is that the only permanent memory that the Workstation provides is Flash, which is the same place that the Workstation stores its firmware and your program. Individual locations in Flash cannot be altered without erasing the entire Flash first, so every time your program stores data, it must use memory that was previously empty. For complete instructions and a sample application, please see the description of the LD@ command in Chapter 10.

If your application requires the capability to change memory without limitation, then you should use a Nematron Series 110 or Series 120 Workstation instead.

Printing to the Display

If you add “#0” immediately following a PRINT command, the Workstation sends the remainder of the PRINT statement to the display. You can use the LOCATE command to position the cursor on the screen, the CLS command to clear the screen, and CALLs 12 and 13 to clear part of the screen.

You can configure the system’s actions at the end of a line and at the end of a screen with the OPEN command. See the discussion of OPEN in Chapter 10 for more information.

Time-Driven Functions

The ON TIME control structure provides a way to interrupt normal program execution in order to perform a subroutine at a scheduled time.

You should see the discussion of ON TIME in Chapter 10 for detailed information.

Chapter 10

Commands

This chapter describes all of the commands and statements in the Series 30/40 BASIC language.

ABS

Returns the absolute value.

ABS(expr)

Where **expr** is any numeric constant, variable, function, or a combination of these.

Discussion

The absolute value of a number is always positive or zero. Like all numeric functions, ABS may appear on the right of an assignment statement, within a PRINT statement, and as part of an arithmetical expression.

Examples

```
>PRINT ABS(-5); ABS(6*(-5)); ABS(0); ABS(6)
5      30    0      6
>
```

ASC

Returns the ASCII value of the first (or specified) character of a string.

ASC(sexpr{,iexpr})

Where **sexpr** is any string expression and **iexpr** is an optional offset (between 1 and 254) into the string.

Discussion

Like all numeric functions, ASC may appear on the right of an assignment statement, within a PRINT statement, and as part of an arithmetical expression. The ASC operator returns the ASCII value of the first character of the string expression. ASCII values range between 0 and 255.

If the optional offset is included, then ASC returns the ASCII value of the selected character, where an offset of 1 selects the leftmost character of the string. This syntax is equivalent to ASC(MID$(sexpr, iexpr,1)).

Examples

```
10 A$ = "DUCKS"
20 PRINT ASC(A$)

>RUN
68

READY - RAM 1
>
```

```
>PRINT ASC("DUCKS")
68

>
```

```
10 A$ = "DUCKLINGS"
20 PRINT ASC(A$,4)
30 REM  In string A$, print ASCII of 4th letter

>RUN
75

READY - RAM 1
>
```

ATN

Returns the arctangent.

ATN(expr)

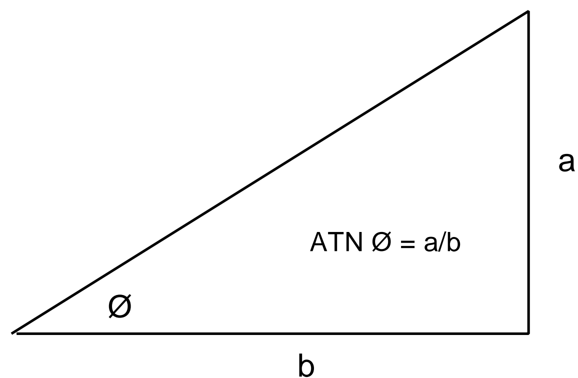
Where **expr** is a number or valid arithmetic calculation.

Discussion

Like all numeric functions, ATN may appear on the right of an assignment statement, within a PRINT statement, and as part of an arithmetical expression. The result of the ATN function is a value expressed in radians in the range of -PI/2 to PI/2 (-1.570796327 to +1.570796327).

Trigonometric Relationship

The arctangent is the resulting angle, expressed in radians, obtained by dividing length (a) by length (b). To convert radians to degrees, multiply the result by 57.29577.



Examples

```
10 INPUT A : INPUT B : REM Opposite and
adjacent sides
20 X = A/B : T = ATN(X) : L = T*57.29577
30 PRINT X,T,L

>RUN
?24
?24
1         .78539804          44.999985

READY - RAM 1
>
```

BIT

Reads from or writes to a bit within an integer.

BIT(iexpr1, iexpr2)

When used in an expression, BIT returns the status of bit number **iexpr2** (1–16) within **iexpr1** (0–65,535). Bit 1 is the least significant bit.

When used on the left side of an equal sign, **iexpr1** must be either a numeric variable whose value varies between 0 and 65,535 or a DBY or XBY location.

Discussion

The BIT statement reads from or writes to a specific bit within the 16 bit integer (an integer varies between 0 and 65535) or certain special integer variables, such as DBY() and XBY().

Examples

The following program defines variable A to be equal to 9, then prints out each of the variable's lowest 16 bits.

```
10 A = 9
20 FOR I = 16 TO 1 STEP -1
30 PRINT BIT(A,I);
40 NEXT

>RUN
0000000000001001
READY – RAM 1
>
```

In the above example, we know that if we force bit number 8 to a 1, the integer will have the value 137. The following program forces bit 8 to a 1 and prints out the resulting value.

```
10 A = 9
20 PRINT "The original integer is ";A
30 BIT(A,8) = 1
40 PRINT "The new integer is ";A
50 PRINT "The new bit pattern is ";
60 FOR I = 16 TO 1 STEP -1
70 PRINT BIT(A,I);
80 NEXT

>RUN
The original integer is 9
The new integer is 137
The new bit pattern is 0000000010001001
```

CALL

Goes to a built-in routine or machine language subroutine.

CALL iexpr

Where **iexpr** is either the address of a machine language subroutine or, if less than 128, the number of a "built-in" CALL.

Discussion

The CALL statement calls a machine language subroutine. Integers between 1 and 127 refer to built-in subroutines (described in Chapter 14); otherwise, the integer is the memory address of a machine language subroutine.

Examples

```
30 CALL 13 : REM Clear to end of line
1000 CALL 9000H : REM Call user-written subroutine
at 9000h
```

Note Some CALLs must be followed by a POP statement if the CALL statement returns a value.

```
60 CALL 40 :  POP B0: REM COM1 status
```

CALLs are described in detail elsewhere in this manual; however, a summary is listed below:

CALL	Description	Examples
12	Clears from cursor position to end of display.	CALL 12
13	Clears from cursor position to end of line on display.	CALL 13
30	Turn off the RTS line of COM1.	CALL 30
33	Turn on the RTS line of COM1.	CALL 33
38	Runs on-line configuration program.	CALL 38
39	Runs monitor.	CALL 39
40	Returns number of characters in port #1 receive buffer.	CALL 40 : POP var
41	Returns number of characters in port #1 transmit buffer.	CALL 41 : POP var
82	Print a list of all variables used.	CALL 82

CBY

Retrieves a byte from program memory.

CBY(iexpr)

Where **iexpr** is a valid integer between 0 and 65535.

Discussion

The CBY operator returns the data from the code address specified by the integer. Because the only code space is a Flash EPROM, you cannot use CBY to write data to code space.

Examples

```
10 A = CBY(1000) : REM A equals value at code
address 1000.
20 PRINT A

>RUN
171

READY - RAM 1
>
```

CHR$ (right side)

Returns a 1-character string with the ASCII value specified.

CHR$(iexpr)

Where **iexpr** is a number in the range of 0 to 255.

Discussion

CHR$ is the converse of the ASC( ) operator. It converts a numeric expression into an ASCII character. CHR$ is a convenient way to print characters not found on the Workstation's keyboard or to print double quotes. This is the only way to print double quotes since they are the delimiter for text strings.

Examples

```
10 PRINT CHR$(64)

>RUN
@

READY - RAM 1
>
```

```
10 FOR I = 48 TO 57
20 PRINT CHR$(I);
30 NEXT

>RUN
0123456789

READY - RAM 1
>
```

```
10 PRINT CHR$(34);"Nematron"; CHR$(34)
20 REM Print double quotes along with the name

>RUN
"Nematron"

READY - RAM 1
>
```

CHR$ (left side)

Programs a programmable character

CHR$(iexpr) = iexpr1,iexpr2,iexpr3,iexpr4,iexpr5,iexpr6,iexpr7,iexpr8

Where **iexpr** is a number in the range of 16 to 23 and **iexpr1** through **iexpr8** are numbers between 0 and 31 that represent a bit pattern for each row.

Discussion

When used on the left side of the equals sign, CHR$ provides a way to define a programmable character. The argument **iexpr** is the character's ASCII code; its valid values are 16 through 23.

The eight values listed to the right of the equals sign each contains the bit pattern for a row, where the first value is the top row and its low bit is the rightmost bit.

The example program below defines the following character under ASCII code 16:

Data	Row	D5	D4	D3	D2	D1
0Fh	1	.	●	●	●	●
11h	2	●	.	.	.	●
11h	3	●	.	.	.	●
0Fh	4	.	●	●	●	●
05h	5	.	.	●	.	●
09h	6	.	●	.	.	●
11h	7	●	.	.	.	●

Examples

```
>CHR$(16) = 0FH,11H,11H,0FH,05H,09H,11H
```

CLEAR

Clears all variables.

```
CLEAR
```

Discussion

CLEAR initializes variable memory to zero; it erases all variables entirely, so any strings or arrays your program previously declared with the SDIM or DIM statements are also erased.

Examples

```
10 A = 23 : B = 45 : C = 12
20 A$ = "Message"
30 PRINT A$
40 PRINT A,B,C
50 CLEAR
60 PRINT A$
70 PRINT A,B,C

>RUN
Message
23  45  12

0  0  0

READY - RAM 1
>
```

```
>A = 23

>PRINT A
23

>CLEAR

>PRINT A
0
```

CLOCK

CLOCK 1 resets the special TIME variable to 0; CLOCK 0 disables ON TIME interrupt service.

CLOCK 0
or
CLOCK 1

Discussion

The CLOCK commands operate with the ON TIME statement to enable BASIC to call a subroutine at time-based intervals. CLOCK 1 resets the TIME variable to 0, which is typically needed immediately after the ON TIME statement as well as at the end of the subroutine. CLOCK 0 turns off the ON TIME function.

Examples

```
20 ON TIME = 5 GOSUB 100
30 CLOCK 1 : REM Reset timer
40 PRINT "Main Program",CR;
50 FOR Z = 1 TO 30/40 : NEXT
60 GOTO 40 : REM Main program loop lines 40 to 60
100 PRINT : PRINT "Interrupt routine entered here"
110 FOR I = 1 TO 10 : PRINT "Interrupt routine" :
NEXT
120 C = C+1
130 IF C < 21 THEN CLOCK 1 : RETI
150 CLOCK 0 : REM Disable interrupts
160 RETI

>RUN
Main Program
Interrupt routine entered here
Interrupt routine
Interrupt routine
.
.
Main Program
```

CLS

Clears the Workstation’s display.

CLS

Discussion The CLS statement clears the Workstation’s display and moves the cursor to the upper left corner of the screen.

Examples

```
10 CLS

>RUN

READY - RAM 1
>
```

```
>CLS
```

CONT

Resumes execution after the program has aborted.

Command mode only

CONT

Discussion

The CONT command causes the unit to resume execution of a BASIC program where it left off. A program that ends normally or with an error cannot CONTinue. A program can CONTinue if it stopped for one of the following reasons:

- Keyboard interruption ([Ctrl]-C on the console or [F1]-[↵] on the Workstation's keyboard).
- Execution of STOP statement in the program.

Typing CONT resumes execution of your program. Between the stopping and the re-starting of the program, you may display or change the value of variables; however, you may *not* CONTinue if you modify the program or if the program stopped because of an error.

Examples

```
10 FOR I=1 TO 1000
20 PRINT I
30 NEXT

>RUN
  1
  2
  3
  4  - (press [Ctrl]-C on the keyboard here)

STOP - IN LINE 20
READY - RAM 1
>I=10

>CONT
  10
  11
```

COPY

Copies a range of memory from one area to another.

COPY iexpr1,iexpr2,iexpr3

Where **iexpr1** is the starting address of the data to be copied, **iexpr2** is the beginning of the area to copy to, and **iexpr3** is the number of bytes to copy.

Discussion

The COPY command allows your program to copy data from one area of memory to another. The COPY command performs correctly whether the data to be copied are above or below the destination address.

For example, to copy 400 bytes from 4000h–418Fh to 4100h–428Fh, the command would be COPY 4000H,4100H,400. In this case, the Workstation copies the byte at 418Fh to 428Fh first and works backwards until it copies the byte from 4000h to 4100h.

Examples

```
10 COPY 4000H,4100H,400

>RUN

READY - RAM 1
>
```

COS

Returns the cosine.

COS(expr)

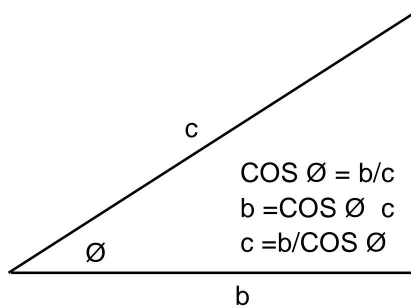
Where **expr** is a number or valid arithmetic calculation in radians.

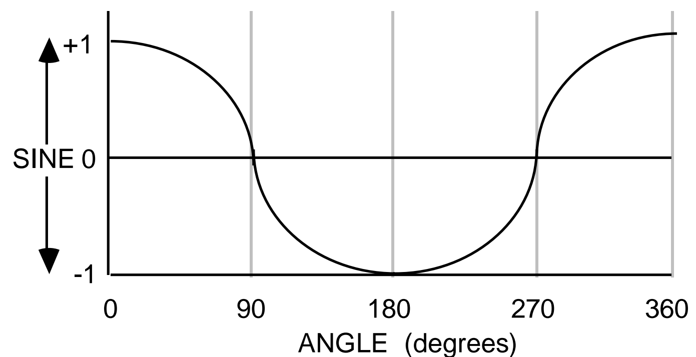
Discussion

Like all numeric functions, COS may appear on the right of an assignment statement, within a PRINT statement, and as part of an arithmetical expression. The argument for the COS function is a value expressed in radians; divide degrees by 57.29577 to convert degrees to radians.

Trigonometric Relationship

The cosine of a triangle is the length of the adjacent side divided by the length of the hypotenuse side (b/c). See the figure below for the relationships between COS Ø, side b, and side c.





Examples

```
10 INPUT "Enter angle in degrees ",A
20 R = A/57.29577 : REM Convert angle to
radians
30 C = COS(R) : Calculate Cosine of angle
40 PRINT "The cosine of the angle is ";C

>RUN
Enter angle in degrees 45
The cosine of the angle is .7071067

READY - RAM 1
>
```

CR

Generates carriage return but suppresses line feed in PRINT statement.

CR

Discussion

The CR statement in a PRINT statement prints a carriage return only. This is useful when your program must "overprint" data that is continuously updated.

Note Typically, CR is the last item in a PRINT statement and is followed by a semicolon (;). If there is no final comma or semicolon, then the PRINT statement defeats the CR function because a PRINT statement normally concludes by printing a carriage return/line feed.

Examples

```
10 CLS : REM Clear Workstation's display
20 FOR I = 65 TO 255
30 FOR N = 1 TO 100 : REM Slow down display
40 NEXT : REM Increment display delay timer
50 PRINT I, CHR$(I), CR;
60 NEXT : REM Increment ASCII character

>RUN
65 A  (Display "overwrites" on this line)

READY – RAM 1
>
```

CSRLIN

Identifies the line number of the display's cursor.

```
CSRLIN
```

Discussion

The CSRLIN statement returns the current line number of the cursor on the Workstation's display. The value returned is between 1 and 2.

POS is a companion statement to CSRLIN which returns the cursor's horizontal (column) position. POS varies between 1 and 20.

Examples

```
10 CLS
20 PR#0
30 PRINT"Hello"
40 PRINT"There"; POS, CSRLIN
50 PR#1

>RUN

READY - RAM 1
>
```

After execution of this example, the Workstation's screen shows the following:

```
Hello
There  6  2
```

DATA

Stores numeric or string data in a program.

Run mode only

DATA const{, const . . .}

Where **const** is a number or string.

Discussion

The READ statement gets its data from the constants following DATA statements.

Multiple data items following DATA must be separated by commas. There is no limit to the number of data items in a DATA statement (except the limit imposed by line length). If a program requires more data items than fit on a single line, you can add more DATA statements.

String data must be enclosed in double quotes.

Because BASIC does not execute DATA statements themselves, you typically include them at the end of your program. Each time BASIC executes a READ statement, it points to the next DATA item. If your program tries to READ when there is no data, the Workstation displays the message "ERROR: NO DATA." The RESTORE command points back to the first DATA item in the program.

Examples

```
40 READ A, B, C, D, E
50 PRINT A, B, C, D, E
60 READ A$, B$, C$
70 PRINT A$, B$, C$
80 DATA 12,24,54,67,78
90 DATA "abc","def","ghi"

>RUN
12  24  54  67  78
abc def ghi

READY - RAM 1
>
```

DATE$

Returns or sets the month, day and year.

DATE$ = “mm/dd/yy”
or
DATE$ = “dd.mm.yy”

Where **DATE$** either sets the date (as shown above), or returns the date.

Discussion

DATE$ can be set as shown in the box above. If parameters are omitted, the unit doesn’t change them. For example, if the current date is 12/30/95 and the unit executes “DATE$ = 11”, then the unit sets the date to 11/30/95
.
The Workstation can set and return the date in the international format (“dd.mm.yy”) if you have previously used the on-line configuration menu to select the international format. See Chapter 5 for instructions.

The Workstation updates DATE$ strictly through software time keeping, which is accurate only to within a few minutes a day.

Examples

```
10 INPUT "Enter today's date: ",DATE$
20 PRINT DATE$

>RUN
Enter today's date: 12/24/94
12/24/94

READY - RAM 1
>
```

DBY

Reads from or writes to internal RAM or special function registers.

DBY(expr)

Where **expr** is a number between 0 and 255; DBY returns a number between 0 and 255.

Discussion

The DBY operator retrieves or assigns a value to the microprocessor's internal RAM memory or special function registers at address **expr**. Both the value and the address must be between 0 and 255 inclusive.

Addresses from 0 to 7Fh refer to the microprocessor's internal RAM memory; BASIC uses all of this memory and leaves nothing available for user data storage.

Addresses from 80h to 0FFh refer to special function registers that control the operation of the microprocessor itself; some of those registers are shown below:

Register	DBY() address	Register	DBY() address
TCON	88h	SBUF	99h
TMOD	89h	IE	A8h
TIMER0	8Ah (low), 8Ch (high)	IP	B8h
TIMER1	8Bh (low), 8Dh (high)	T2CON	C8h
PORT1	90h	RCAP2	CAh (low), CBh (high)
SCON	98h	TIMER2	CCh (low), CDh (high)

Note ***Nematron recommends that you do not refer to DBY locations.*** Changing internal memory locations or microprocessor registers could have disastrous consequences.

Examples

```
10 PRINT DBY(35)

>RUN
13

READY - RAM 1
>
```

DEL

Deletes all or part of a program or all programs.

Command mode only

DEL line#1 { , { line#2}}
or
DEL RAM {prog#}

Where **line#1** and **line#2** are any valid line numbers in the program or **prog#** is any valid program number.

Discussion

You can use the DEL command to remove one or more lines from your program.

DEL	No action taken (use DEL, instead)
DEL n	Delete line "n" only
DEL n,	Delete from line "n" to the end of the program
DEL ,n	Delete from the beginning to line "n"
DEL m,n	Delete from line "m" to line "n"
DEL ,	Delete all lines
DEL RAMx	Delete program *x* from RAM memory
DEL RAM	Delete all programs from RAM memory

Before the system actually deletes all programs from memory, it displays the following prompt: "Delete all programs (press backspace to delete)?" To follow through with deleting, you must press the [Backspace] key on your console.

Examples

```
>LIST
 10 SDIM A$(20) : REM Set string length to 20
 20 A$ = "Eldridge, Iowa"
 30 L = LEN(A$)
 40 PRINT "Length of string is",L,"characters"
 50 PRINT A$

READY - RAM 1
>
```

```
READY - RAM 1
>DEL 20,30

>LIST
 10 SDIM A$(20) : REM Set string length to 20
 40 PRINT "Length of string is",L,"characters"
 50 PRINT A$

READY - RAM 1
>
```

```
>DEL RAM 1

READY - RAM 1
>LIST


READY - RAM 1
>DEL RAM
Delete all programs (press backspace to delete)?

READY - RAM 1
>
```

DIM

Declares the size of an array variable.

DIM nvar(iexpr1)
or
DIM svar(iexpr1,iexpr2)

Discussion

The DIM statement creates a numeric array named **nvar** with a size (i.e., number of elements) of **iexpr1** or a string array named **svar** with a size of **iexpr1** and a string length of **iexpr2.** The maximum number of elements in an array is 254, and the maximum length of each string is 254. The value of each element in an array is initially zero.

If a program refers to an array before dimensioning, then BASIC automatically creates the array with a size of 10 items. If it's a string array, each string has the default length (normally 10 characters).

You can change the default string length of 10 characters with the statement *SDIM = x*, where *x* is the desired default length. For example, if you want a default string length of 20, your program would include the statement SDIM = 20 *before* any statements that declare a new string.

You cannot change the size of an array after you have dimensioned it; any attempt to do so results in a "REDIMENSION ERROR."

Elements in an array are numbered from 0. A reference to an element outside the array dimension results in an ARRAY SIZE error.

The amount of memory used (in bytes) for a numeric array is 14 plus 6 times the array size; for example, DIM A(100) allocates 14 + 6 * 100, or 614 bytes of memory. The amount of memory used for a string array is (string length + 1) times (array size plus 1) plus 8; for example, DIM A$(15,20) allocates (15 + 1) * (20 + 1) + 8, or 344 bytes of memory.

A DIM statement can handle more than one variable, as shown in the first example.

Examples

```
>10 DIM A(20), B(35), C$(30,22)
```

This line should be early in the program and assigns an array size of 20 to variable A, an array size of 35 to variable B, and an array size of 30/40 to variable C$ (where each string is 22 characters long).

The program below produces a redimension error at line 120 because line 20 creates the array simply by referring to an array variable:

```
10 X = 0
20 PRINT A(X)
30 DIM A(5)

>RUN
ERROR:    REDIMENSION - IN LINE 30/40

30 DIM A(5)
------- X
```

DIR

Prints a directory of all programs in memory.

```
DIR
```

Discussion

The DIR command prints a list of all programs in memory, with each program's starting address and length (in hex) and any REM statement that appears on the first line.

In order to use the DIR command to its best advantage, the first line of each program should be a REM statement that lists the program's title and purpose.

Examples

In the following example, the first line of the first program contains a REM followed by "TEST Program 1"; the first line of the second program contains no REM statement:

```
>DIR
RAM 1 (1000H,1982H) -  TEST Program 1
RAM 2 (2982H,0254H) -
ROM   (A000H,0125H) -  TEST Program in ROM

READY - RAM 1
>
```

DO . . . UNTIL

Loops until a condition is true.

```
DO
--
UNTIL rexpr
```

Where **rexpr** is a condition that must be true in order to terminate the loop.

Discussion

The DO and UNTIL statements provide a means of “loop control” within a BASIC program. BASIC executes all statements between the DO and the UNTIL until the relational expression following the UNTIL statement is true.

You can “nest” additional control loops in a DO - UNTIL loop, such as DO - WHILE and FOR - NEXT. The program example on the right below shows a DO - UNTIL loop nested in another DO - UNTIL loop.

Note Your program must *not* exit a DO - UNTIL loop with a GOTO. The only way your program should terminate a DO - UNTIL loop is to wait until the condition that terminates the loop is true.

Nesting Limits

The DO, FOR and GOSUB statements all allow BASIC to repeatedly execute a series of statements. Every time BASIC executes one of these statements, it saves information that it eventually needs to execute the WHILE, UNTIL, NEXT or RETURN that marks the end of the series of statements.

BASIC reserves an area that we call the “control stack” to save the information it needs for any active DO, FOR and GOSUB statements. This control stack can hold 178 bytes; DO and GOSUB statements each require three bytes, while the FOR statement requires 18 bytes. Your program can nest these statements in any combination, up to the limit of the control stack.

For example, your program could contains GOSUBs that in turn call GOSUBs, until your program has reached 59 levels (178/3) of nesting. However, your program can contain only 9 levels of FOR loops. Your program can combine these so that FOR loops contain GOSUBs, which themselves contain DO loops and FOR loops, and so on, up to the limit of 178 bytes.

If your program exceeds the capacity of the control stack, then BASIC issues “C-STACK” errors.

Examples

```
SIMPLE DO - UNTIL    NESTED DO - UNTIL

10 A=0               10 DO : A=A+1 : DO : B=B+1
20 DO                20 PRINT A, B, A*B
30 A=A+1             30 UNTIL B=3
40 PRINT A           40 B=0
50 UNTIL A=4         50 UNTIL A=3
60 PRINT "DONE"
                     >RUN
>RUN
1                    1     1        1
2                    1     2        2
3                    1     3        3
4                    2     1        2
DONE                 2     2        4
                     2     3        6
READY - RAM 1        3     1        3
>                    3     2        6
                     3     3        9

                     READY - RAM 1
```

DO . . . WHILE

Loops while a condition is true.

DO
--
WHILE rexpr

Where **rexpr** is a condition that keeps the loop going. When the expression is no longer true, the loop terminates.

Discussion

The DO and WHILE statements provide a means of "loop control" within a BASIC program. BASIC executes all statements between the DO and the WHILE as long as the relational expression following the WHILE statement is true.

You can "nest" additional control loops in a DO - WHILE loop, such as DO - WHILE and FOR - NEXT. The program example on the right below shows a DO - WHILE loop nested in another DO - WHILE loop. For more information about nesting, see the section titled "Nesting Limits" on page 11-25.

Note Your program must *not* exit a DO - WHILE loop with a GOTO. The only way your program should terminate a DO - WHILE loop is to wait until the condition that terminates the loop is no longer true.

Examples

```
SIMPLE DO - WHILE     NESTED DO - WHILE

10 DO                 10 DO : A=A+1 : DO : B=B+1
20 A=A+1              20 PRINT A, B, A*B
30 PRINT A            30 WHILE B<>3
40 WHILE A<4          40 B=0
60 PRINT "DONE"       50 UNTIL A=3

>RUN                  >RUN
1                     1    1          1
2                     1    2          2
3                     1    3          3
4                     2    1          2
DONE                  2    2          4
                      2    3          6
READY - RAM 1         3    1          3
>                     3    2          6
.                     3    3          9

                      READY - RAM 1
```

DUMP

Displays external or code memory in hex.

DUMP iexpr1, iexpr2

Where **iexpr1** is the starting address and **iexpr2** is the number of bytes to display.

Discussion

The DUMP statement provides a handy way to display the contents in hex of a range of memory. For addresses below 8000h, the DUMP statement prints the contents of external RAM memory. For addresses at 8000h and above, the DUMP statement prints the contents of external Flash memory.

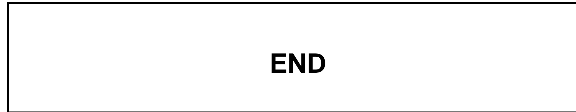
Examples

```
>DUMP 400h,10
1BH 00H 20H 23H 4EH 4FH 51H 5FH 55H 0DH

>
```

END

Terminates program execution and returns BASIC to Command mode.

Run mode only



Discussion

The END statement terminates program execution. The CONT command does not operate if the END statement terminates execution; the Workstation prints CAN'T CONTINUE instead.

The last statement in a BASIC program automatically terminates program execution if there is no END statement .

Examples

```
Last Statement Term.        END Statement Term.

10 FOR I=1 TO 4             10 FOR I=1 TO 4
20 PRINT I                  20 GOSUB 80
30 NEXT                     30 NEXT
>RUN                        40 END
                            80 PRINT I
1                           85 RETURN
2                           RUN
3
4                           1
                            2
READY - RAM 1               3
>                           4

                            READY - RAM 1
                            >
```

ERR and ERL

Returns error code and line number of an error.

```
ERR
ERL
```

Discussion ERR and ERL are reserved variable names that are valid only when BASIC enters an error-handling routine via an "ON ERROR GOTO" statement. ERR is the error number and ERL is the line number where the error occurred. Appendix E contains a list of valid error numbers.

Note Some communications errors can occur asynchronously, which means that ERL may not be valid. For example, if a receive buffer overflows, ERR will be correct but ERL will be the number of the line that the Workstation was executing when the overflow occurred.

For example, when your program opens COM1 with the IB parameter, it receives characters into a buffer. If the device connected to COM1 transmits too much data for the Workstation to process, eventually the buffer fills up and generates an error (ERR = 80, "COM1 RECEIVE BUFFER OVERFLOW"). The Workstation stops program execution wherever it happens to be, prints this error message, and prints the line that it was executing when the error occurred.

Examples

```
10 ON ERROR GOTO 1100
20 INPUT A : INPUT B
30 C = A/B
40 PRINT C
50 END
100 PRINT "Error Code ";ERR
110 PRINT "Error line number ";ERL
120 B = 1
130 RESUME

>RUN
?56
?0

Error Code 10
Error line number 30/40
56
```

EXP

Returns “e” (2.7182818) raised to a power.

EXP(expr)

Where **expr** is any numeric expression.

Discussion The EXP statement returns the mathematical constant “e” raised to **expr**.

Examples

```
10 INPUT "Enter number",X
20 A = EXP(X)
30 PRINT A

>RUN
Enter number 2
7.3890559


READY - RAM 1
>
```

```
>PRINT EXP(3)
20.085536

>
```

FOR . . . NEXT

Loops a specified number of times.

FOR nvar = expr1 TO expr2 {STEP expr3}
NEXT {nvar}

Where **nvar** is a numeric variable that initially takes the value of **expr1** and increases it by **expr3** until **nvar** exceeds **expr2**. If you omit the STEP parameter, the default step value is 1. If the STEP parameter is negative, then **nvar** decreases by the value of **expr3** until it is less than **expr2**.

BASIC adds the STEP value to **nvar** when it executes the NEXT statement, which means that every FOR-NEXT loop is executed at least once regardless of **expr1** and **expr2.** Note that the **nvar** following NEXT is optional. If you include it, however, it must be the same as the **nvar** following FOR.

Discussion

The FOR - NEXT loop is a handy way to execute one or more statements a specific number of times. As indicated in the explanation above, you can specify a STEP value other than 1 so that the loop either adds or subtracts any number from the loop variable (nvar) until the loop is finished.

BASIC always executes a FOR - NEXT loop at least once, because it doesn't check the loop variable against the loop limit (expr2) until the NEXT statement. You can "nest" control loops within each other. For example, the statements between a FOR and NEXT can include additional FOR - NEXT loops as well as other control loops. For more information about nesting, see the section titled "Nesting Limits" in this chapter.

Note Your program must *not* exit a FOR - NEXT loop with a GOTO. The only way your program should terminate a FOR - NEXT loop is to finish it.

Examples

```
10 FOR I = 1 TO 2
20 PRINT I
30 NEXT
40 PRINT I

>RUN
 1
 2
 3
```

This example first sets I to 1 and prints it. Then when BASIC executes the NEXT statement, it adds the default STEP value of 1 to the variable I and compares the result of 2 to the limit of 2. Because I is not greater than 2, BASIC returns to line 20 and prints 2, the current value of I. The next time BASIC reaches the NEXT statement, it changes I to 3, which causes I to exceed the loop limit; BASIC continues execution at line 40.

```
10 FOR I = 4 TO 0 STEP -2
20 PRINT I
30 NEXT I

>RUN
 4
 2
 0
```

FREE

Returns the number of memory bytes available to BASIC.

FREE

Discussion

FREE returns the number of program memory bytes available to BASIC. As you add lines to your program, FREE decreases and heads toward zero. Note that FREE does *not* account for the memory your program needs to store variables; if FREE returns a value of zero, then there is not enough memory for your program to run.

Examples

```
>PRINT FREE
7168

>
```

GOSUB . . . RETURN

Branches to a subroutine.

GOSUB line#

Where **line#** is the number of the line that BASIC executes next.

Discussion

The GOSUB statement causes BASIC to save its current position in the program and continue program execution at **line#**. When BASIC executes a RETURN statement, it recalls its saved position and continues program execution at the statement immediately following the GOSUB.

The portion of the program beginning at **line#** and ending with the RETURN statement is typically called a "subroutine." Subroutines are useful because the main program can perform a function from different points in the program with a single group of statements.

You can "nest" control loops within each other. For example, a subroutine can call another subroutine; in fact, a subroutine can even call itself. For more information about nesting, see the section titled "Nesting Limits" in this chapter.

Examples

```
SIMPLE GOSUB

10 A = 23 : B = 47
20 GOSUB 200
30 END
200 X = A*B
210 PRINT X
220 RETURN
>RUN
 1081

READY - RAM 1
>
```

```
NESTED GOSUB

10 FOR I=1 TO 3
20 GOSUB 200
30 NEXT
40 END
200 PRINT I,
210 GOSUB 400
220 RETURN
00 PRINT I*I
410 RETURN
>RUN

1       1
2       4
3       9

READY - RAM 1
>
```

GOTO

Continues program execution at a specified line number.

GOTO line#

Where **line#** is the number of the line that BASIC executes next.

Discussion

Unlike the RUN command, the GOTO statement, if executed in the Command mode, does not erase the variables. However, if you edit a program line and then execute the GOTO statement, BASIC erases all variables. This is necessary because the variable storage and the BASIC program reside in the same memory. Therefore, editing a program can destroy variables.

Examples

```
10 PRINT "apples"
20 PRINT "oranges"
30 GOTO 50"
40 PRINT "peaches"
50 PRINT "I skipped peaches"
60 END

>RUN
apples
oranges
I skipped peaches

READY - RAM 1
>
```

Line 30 causes execution of the program to continue at line 50. If line 50 does not exist, the Workstation prints the message ERROR: UNDEFINED LINE NUMBER.

In Command mode, the following entry causes the program to execute from line number 50.

```
>GOTO 50
I skipped peaches

READY - RAM 1
>
```

HEX$

Returns the string form of a number in hexadecimal format.

HEX$(iexpr)

Where **iexpr** is a value in the range of 0 to 65,535 and HEX$ returns a string between “0000” and “FFFF.”

Discussion

The HEX$ operator converts the **expr** to a string where the number is in 4-digit hexadecimal format. If you want to convert an expression to a decimal format, you must use STR$ instead.

Examples

```
10 INPUT "Enter decimal number ",X
20 PRINT "The hex number",HEX$(X), "represents",X

>RUN
Enter decimal number 165
The hex number 00A5 represents 165

READY - RAM 1
>
```

```
10 INPUT "Enter decimal number ",X
20 PRINT HEX$(X/2), X

>RUN
Enter decimal number 182
005B  182

READY - RAM 1
>
```

HVAL

Returns the numeric value of a hex string.

HVAL(sexpr)

Where **sexpr** is a string containing numbers or characters that represent a hexadecimal number in the range of 0 to 0FFFFh.

Discussion

HVAL converts a string representing a hexadecimal number to a decimal number. It is important that the string contain only hexadecimal digits (0–9 and A–F). If the string starts with a non-hexadecimal digit, then HVAL returns zero. If the string starts with a hexadecimal digit but contains a non-hexadecimal number, the number returned represents only up to the first non-hexadecimal digit.

Sexpr does not have to begin with a numeric digit and does not have to end with an "H."

Examples

```
10 A$ = "FF"
20 PRINT HVAL(A$)

>RUN
255
```

```
10 A$ = "GA"
20 PRINT HVAL(A$)

>RUN
0
```

```
10 A$ = "FTF"
20 PRINT HVAL(A$)

>RUN
15
```

IBY

Reads from and writes to the indirect internal RAM.

IBY(expr)

Where **expr** is a number between 0 and 255; IBY returns a number between 0 and 255.

Discussion

The IBY operator retrieves or assigns a value to the microprocessor's internal indirect RAM memory at address **expr**. Addresses from 0 to 7Fh are actually direct internal RAM (also accessible with DBY), while addresses above 7Fh are indirect only. In any event, BASIC reserves all of this space for itself, so you must exercise great care in the use of the IBY operator.

Examples

```
10 FOR X = 108 to 110 : PRINT IBY(X),: NEXT

>RUN
 0  51  74

READY – RAM 1
>
```

IF . . . THEN . . . ELSE

Directs program flow based on a condition.

IF rexpr {THEN} statement {ELSE statement}
or
IF rexpr THEN line# {ELSE statement}

Where **rexpr** is a relational expression.

Discussion

The IF statement tests the value of **rexpr**. If **rexpr** is true (i.e., not zero), then BASIC executes the statement following THEN (and skips the statement following ELSE). If **rexpr** is false (i.e., zero), then BASIC executes the statement or statements following the ELSE. If there is no ELSE, then BASIC simply continues execution at the next line in the program.

Note: if a number follows THEN or ELSE, BASIC considers that to be the same as THEN GOTO line number or ELSE GOTO line number. In other words, you can save a little program space by omitting the "GOTO."

Examples

```
10 INPUT X : IF X = 1 GOTO 100 ELSE 70
20 REM BASIC never executes this line.
70 PRINT "X was not 1"
80 END

100 PRINT "X was 1"
>RUN
?1
X was 1

READY - RAM 1
>RUN
?6
X was not 1

READY - RAM 1
>
```

```
10 INPUT "Enter Number ",X
20 IF X = 5 THEN A = 2 ELSE A = 6
30 PRINT X, A

>RUN
Enter Number 5
5  2

READY - RAM 1
>
```

IN#

Switches input to a selected port.

IN# iexpr

Where **iexpr** is 0 or 1 and selects the port from which the Workstation receives subsequent input.

Discussion

The IN# statement selects the input port for all future INPUT and INPUT$ statements, although individual INPUT and INPUT$ statements can contain port selections that temporarily override the IN# port selection.

When communicating, the display and keyboard are also considered to be a port. The following table summarizes the port names and numbers:

Name	Number	Description
COM0	0	Keypad and display
COM1	1	COM1 serial port

For more information about the serial communications ports, consult the OPEN command.

Examples

```
10 IN#0 : REM Get future input from keypad.

>RUN

READY - RAM 1
>
```

INKEY$

Returns a single character, if any, from current input port.

INKEY$ {# port#}

Where **port#** is 0-1 and selects the port the Workstation checks for a key.

Discussion

You can use INKEY$ to test for input on an input port. If you don't specify the port, then INKEY$ tests the current input port. If there is no input available, INKEY$ returns a string of length 0. If there is at least one character available, INKEY$ returns that character as a one-character string.

When communicating, the display and keyboard are also considered to be a port. The following table summarizes the port names and numbers:

Name	Number	Description
COM0	0	Keypad and display
COM1	1	COM1 serial port

See the description of the IN# statement for a complete discussion of the unit's communications ports.

Examples

```
100 SDIM A$(1),B$(254)
110 CLS
120 ON TIME = 1 GOSUB 200 : CLOCK 1
130 A$ = INKEY$ #1
140 IF LEN(A$) = 0 THEN 130
160 IF A$ <> CHR$(13) THEN 130
170 PRINT : PRINT "DONE!" : REM Operator hit
[Enter]
180 END
200 PRINT TIME$ ; CR ; : CLOCK 1 : RETI

>RUN
04:24:50
DONE!

READY - RAM 1
>
```

INPUT

Requests an entry to a variable.

Run mode only

INPUT {# port#,}{"prompt"} {,} var...{,varx}

Where **port#** is an optional input port number (see the description of the IN# statement for a complete discussion of the unit's communications ports). **Prompt** is an optional message that BASIC prints to the current output port before accepting input for each **var.** If there is no comma preceding the first variable, the Workstation sends a question mark; if there is a comma, BASIC doesn't print the question mark.

Discussion

BASIC begins execution of the INPUT statement by looking for an optional input port number. If there is a port number, then BASIC looks for input from that port only for the duration of the current INPUT statement.

If there is an optional prompt, then BASIC transmits that prompt string to the currently selected output port. (The PR# statement selects the current output port.) Then BASIC accepts input to the numeric or string variable(s) listed.

When communicating, the display and keyboard are also considered to be a port. The following table summarizes the port names and numbers:

Name	Number	Description
COM0	0	Keypad and display
COM1	1	COM1 serial port

Operator input

If there is more than one numeric variable in an INPUT statement, the operator can either make one entry at a time (terminated with carriage return), or make all entries at once, where the operator enters a comma between each entry. For string variables, however, the operator must make each entry separately.

If the INPUT statement calls for additional entries, BASIC automatically prints a question mark before each subsequent entry, even if it didn't print a question mark for the first entry.

If your program tries to accept operator input at the bottom line of the display, the display may scroll up one line when the operator presses [Enter]. This happens because the Workstation prints a line feed after the carriage return. To suppress the line feed, OPEN the COM0 without the LF parameter. This disables printing a line feed after printing a carriage return on the display.

TRY AGAIN

If you respond to an INPUT statement with too many or too few items or with the wrong type of value (numeric where a string is expected, or vice versa), BASIC prints the message TRY AGAIN. To avoid this message, use the form *INPUT **svar***, where **svar** is a string variable, and convert it to a value with the VAL operator.

Echo suppression

Ordinarily, BASIC transmits to the current output port each character it receives as input; this is usually referred to as “echoing” the input. You can use the ED parameter in the OPEN command to suppress echoing on specific ports; refer to the description of the OPEN command for more detail.

INPUT delay timer

While BASIC is waiting for INPUT, it cannot do anything else; in fact, even ON TIME interrupts are disabled during INPUT. Although BASIC ordinarily waits forever for input, you can use the TD parameter in the OPEN statement to set up a timer that causes an error when time is up; your error-handling routine can react to that specific error. See the description of OPEN and ON ERROR.

Receiving control characters

You should use the INPUT$ function instead of INPUT when you need to accept input without a terminating carriage return or when you simply want to check the input port and accept any input if it’s there. See the discussion of INPUT$ for more information.

Keep in mind that INPUT does not accept certain characters in a string. To receive any of the following characters, except when 0Dh (the [Enter] key) terminates the INPUT, your program must use the INPUT$( ) function instead.

Console Character	Keypad Character	Code	Function
[Ctrl]-C	F1 and ↵	03h	Halts execution of the program (only if enabled with the CE parameter)
[Ctrl]-D		04h	Deletes all characters received so far
[Ctrl]-X		18h	Same as [Ctrl]-D
Enter	Enter	0Dh	Terminate entry to INPUT.
Delete	Cancel	7Fh	Delete last character typed.

Keypad ASCII Codes

The following table lists the ASCII codes and corresponding characters for each key of the Workstation's keypad. This table also lists the many two-key combination characters that the Workstation supports:

Key(s)	ASCII Code	Character or Function
F1 ↵	3	[Ctrl]-C
↓	10	Line feed
↑	11	Reverse line feed
↵	13	Carriage return
+	43	+
-	45	-
F1 F2	48	0
F2 ↑	49	1
↑ +	50	2
+ x	51	3
F1 x	52	4
F3 F4	53	5
F4 ↓	54	6
↓ -	55	7
- ↵	56	8
F3 ↵	57	9
x	127	Backspace
F1	241	
F2	242	
F3	243	
F4	244	
F1 F3	245	
F2 F4	246	
↑ ↓	247	
x -	248	
x ↵	249	

Examples

This example shows an input prompt, followed by the input of a numeric variable. BASIC displays a question mark because no comma precedes the input variable.

```
10 INPUT "ENTER A NUMBER "A
20 PRINT SQR(A)

>RUN
ENTER A NUMBER ?144
 12
```

In the next example, the INPUT statement expects two numbers to be input; the comma suppresses printing of the first question mark, but BASIC prints the question mark in front of the second input. The first time the program runs, the operator terminates each entry with a carriage return; the second time, the operator enters both answers at once.

```
10 INPUT ,A,B
20 PRINT A,B

>RUN
16
?45
 16  45

READY - RAM 1
>RUN
16,45
 16  45

READY - RAM 1
>
```

In the following example, BASIC first prints the prompt string but suppresses the question mark; then BASIC accepts a string variable.

```
10 INPUT "NAME:",A$
20 PRINT "HELLO ",A$

>RUN
NAME: CAROLEE
HELLO CAROLEE

READY - RAM 1
>
```

In the next example, BASIC expects first a string entry and then a numeric entry.

```
10 INPUT "Enter your name and age - ",A$,A
20 PRINT "HELLO ",A$, ", YOU ARE ",A, " YEARS
OLD"

>RUN
Enter your name and age - KRIS
?34

HELLO KRIS, YOU ARE 34 YEARS OLD

READY - RAM 1
>
```

In the following example, the console is the current output port. Line 20 selects the keypad as the input device (#0); notice that because the current output port is the console, the Workstation echoes the user input to the console, not to the Workstation's screen. You should also notice that this command generates a question mark even though a comma precedes the variable; to eliminate the question mark, you need two commas.

```
10 CLS
20 INPUT #0, A : REM Inputs from keypad.

>RUN
?563 (input from keypad, echoed to console)
```

INPUT$

Requests an entry of a specified length.

INPUT$(expr, {# port#})

Where **expr** is the number of characters expected (if **expr** = 0, then this function returns all the characters waiting in the buffer) and **port#** is an optional input port selection.

Discussion

The INPUT$ function returns a string of length **expr.** If no port number is specified, BASIC uses the current input port. INPUT$ accepts any characters, including control characters. INPUT$ does not echo received characters to the current output port. If the specified length is zero, then INPUT$ returns all characters that are immediately available.

Note Even though INPUT$ normally accepts all characters including [Ctrl]-C**,** you can still break out of a program that has halted on an INPUT$ function by pressing the keypad's [F1] and [↵] keys simultaneously, but only if you have OPEN'ed COM0 with the CE parameter.

You can use INPUT$ to test whether input is available. For example, to test whether a character is available on the keypad, a program could use INPUT$(0,#0) and test the length of the string returned. (Of course, you could also use INKEY$ #0 for the same function.)

Examples

```
10 A$=INPUT$(5):REM A$=Next 5 characters from
current port
20 A$=INPUT$(7,#0):REM A$=Next 7 characters
from keypad

>RUN

READY - RAM 1
>
```

INSTR

Returns the position of a string within another string.

INSTR({iexpr},sexpr1,sexpr2)

Where **iexpr** is an optional offset to the starting position for the search, and **sexpr2** is the string for which INSTR searches within **sexpr1.**

Discussion

INSTR (in string) returns the starting position of string **sexpr2** within string **sexpr1.** If there is an optional offset, then the search starts at the nth character of **sexpr1**, where n = **iexpr**. If there is no offset, then INSTR assumes an offset of 1.

If the string is not found, INSTR returns 0. If the offset exceeds the length of **sexpr1,** INSTR returns 0. If the length of **sexpr2** is 0, INSTR returns the offset (or 1 if no offset is specified).

Examples

```
10 A$ = "Bettendorf"
20 A = INSTR(A$,"tend")
30 PRINT "The position is",A

>RUN
The position is 4

READY - RAM 1
>
```

```
10 A$ = "Bettendorf"
20 B$ = "do"
30 A = INSTR(A$,B$)
40 PRINT "The position is",A

>RUN
The position is 7

READY - RAM 1
>
```

INT

Returns the integer portion of a number.

INT(expr)

Where **expr** is any valid numeric expression.

Discussion INT strips away the fractional portion of an expression, leaving only the whole number.

Examples

```
>PRINT INT((12^2) + 3.548)
147
>
```

```
10 A = 45.767
20 PRINT INT(A)

>RUN
45

READY - RAM 1
>
```

```
10 A = (12^2) + 3.548
20 PRINT A, INT(A)

>RUN
147.548   147

READY - RAM 1
>
```

INV

Returns the inverse.

INV(expr)

Where **expr** is any valid numeric expression.

Discussion

INV returns the “inverse” of the expression. If the expression is 0, then INV returns a 1; if the expression is not 0, then INV returns 0.

This function is very useful in logic operations where you want complementary logic. Its relay ladder equivalent would be “examine off” instead of “examine on.”

Examples

```
>PRINT INV(10), INV(0)
0     1
>
```

LD@

Retrieves data from external memory and places it in a variable.

```
LD@ iexpr, var { ,var2 . . .}
```

Where **iexpr** is the starting address of external memory that contains data, and **var** is the variable to receive the data. Additional variables that receive their data from consecutive memory locations are optional.

Discussion

In combination with the ST@ command, LD@ provides the ability to save data in the unit's memory. ST@ writes variables to memory, while LD@ reads variables from memory.

You can use LD@ to retrieve one or more variables of any type, including floating point numbers, integers, and strings. You must be careful, of course, to retrieve variables in the same sequence in which you saved them, or strange things will happen.

The **iexpr** parameter is the starting address of the data for **var**. If there are more variables following **var**, then they receive their data from consecutive memory locations.

Reserving memory

In order to save data in memory, you must first find the necessary space. This space comes from unused Flash EPROM space above your program. To calculate the first available address, you should use the PLEN command to get the length of your program and add it to A010h; the result is the first available address. The maximum address you can use is 0FFFFh.

Although each location in Flash memory can be written to individually, it must be entirely erased before a location that has already been written can be changed. And because BASIC stores itself and your program in the same Flash, you cannot erase the Flash without downloading firmware and your program again.

A location that has not been written has the value 255 (0FFh); you can change that value to any other value, but the only way to change it back to 255 is to erase the entire Flash.

Memory usage

Floating point and integer numbers use six bytes of memory. For example, if your program contains the statement "LD@ 0F000h,A,B%", then variable A is retrieved from F000h to F005h, while B% is retrieved from F006h to F00Bh.

ST@ stores strings with their actual length plus 1 byte to hold the length. For example, if your program stores the string "ABC" at location F000h, the length byte is at F000h and the characters "ABC" are stored between F001h and F003h. (ST@ uses only enough memory to store the actual string, not its maximum length.)

Variable Type	Memory Usage
Floating point	6 bytes
Integer	6 bytes
String	Actual length plus 1

For more information about how BASIC stores variables, consult the description of the VARPTR statement later in this chapter.

Special VAD variable

BASIC has a special variable called VAD that always points one byte past the last address that the latest LD@ or ST@ used. This is handy to avoid a lot of "bookkeeping" for programs that store lots of strings, whose lengths can vary.

Examples

The example program on the following page saves two variables, S1 and S2, in Flash memory. When the program starts, it finds the two variables in Flash and loads them. If the operator then chooses to change them, the program accepts new values and stores them in memory.

Line 130 establishes address 0F000h as the starting point for permanent variable memory. This address relies on your program's length remaining less than 20,000 bytes; you can find out your program's length by printing PLEN after you have finished writing your program. With roughly 4,000 bytes of memory available between 0F000h and 0FFFFh, and with 12 bytes needed to store the two permanent variables, you can store changes to these variables about 300 times before running out of memory.

Line 140 calls a subroutine starting at line 540 that checks whether the six bytes of memory starting at address VAD are erased. If they are, then line 150 initializes the permanent variables to zero and stores them in permanent memory.

If the six bytes of memory at VAD are not erased, then line 160 loads in the variables that are already stored there. Line 170 calls the subroutine at line 540 again to see if the next six bytes are erased. If they are, then the program continues at line 180 because it has already found the last instance in memory of the two permanent variables.

However, if the following bytes are not erased, then the program reloads the variables from the next higher 12 bytes of memory. This process continues until the program has found the most recent instance of the two permanent variables.

Lines 180 through 260 allow the operator to choose whether to change the permanent variables or continue with the program. Lines 280 through 380 allow the operator to change the variables; before allowing the operator to proceed, however, line 290 tests VAD to see if there is enough space remaining in permanent memory to store another instance of the variables.

Lines 330 through 370 accept new values for the permanent variables, relying on the subroutine starting at line 440. This subroutine displays the current value and allows the operator to press [+] to increase it (line 500), [–] to decrease it (line 510), [↵] to enter the new value (line 490) and [x] to cancel the entry and restore the old value (line 520).

Line 380 concludes by storing the two new values in permanent memory.

```
100   REM Demonstration program of variable
saving program
110   REM Saves variables S1 and S2 in Flash
memory
120   OPEN "COM0:IB,CE,LF,CL,SC,AR,RD1"
130   VAD = 0F000H
140   GOSUB 540 : IF N = 1 THEN 160
150   S1 = 0 : S2 = 0 : ST@ VAD ,S1,S2 : GOTO 130
160   LD@ VAD ,S1,S2
170   GOSUB 540 : IF N = 1 THEN 160
180   CLS
190   PRINT #0,"[F1] - New variables";
200   PRINT #0,"[F2] - Continue";
210   A = ASC(INPUT$(1,#0))
220   IF A = 0F2H THEN 390
230   IF A = 0F1H THEN 270
240   CLS : PRINT #0,"Wrong answer!"
250   PRINT #0,"Try again.";
260   FOR X = 1 TO 500 : NEXT : GOTO 180
270   REM Allow operator to enter new variables
280   CLS
290   IF VAD < 0FFF0H THEN 330/40
300   PRINT #0,"No more memory!";
310   FOR X = 1 TO 500 : NEXT
320   GOTO 390
330   PRINT #0,"S1 =",S1;
340   N = S1 : GOSUB 440 : S1 = N
350   CLS
360   PRINT #0,"S2 =",S2;
370   N = S2 : GOSUB 440 : S2 = N
380   ST@ VAD ,S1,S2
390   REM Run program normally
400   CLS
410   PRINT #0,"S1 =",S1
420   PRINT #0,"S2 =",S2;
430   END
440   REM Accept new value
450   N1 = N
460   LOCATE 2,1
470   PRINT #0,"New value =",N1; : CALL 13
480   A = ASC(INPUT$(1,#0))
490   IF A = 13 THEN N = N1 : RETURN
500   IF A = 43 THEN N1 = N1 + 1
510   IF A = 45 AND N1 > 0 THEN N1 = N1 - 1
520   IF A = 127 THEN RETURN
530   GOTO 460
540   REM Test for erased value at VAD
550   N = 0 : T = VAD : FOR X = 0 TO 5
560   IF XBY(VAD + X ) <> 255 THEN N = 1
570   NEXT : VAD = T : RETURN
```

LEFT$

Returns the left part of a string.

LEFT$(sexpr,iexpr)

Where **sexpr** is any string expression and **iexpr** is the length of the string to return.

Discussion

If **iexpr** is greater than the number of characters in the string, the entire string will be returned. If **iexpr** is zero, a null string (length 0) is returned.

Examples

```
10 A$ = "Bettendorf"
20 B$ = LEFT$(A$,3)
30 PRINT B$

>RUN
Bet

READY - RAM 1
>
```

```
>PRINT LEFT$("Bettendorf",4)
Bett

>
```

LEN

Returns the number of characters in a string.

LEN(sexpr)

Where **sexpr** is any valid string expression.

Discussion

LEN counts the number of characters in a string, including printable, unprintable and blank characters.

Examples

```
10 SDIM A$(20) : REM Dimension string
20 A$ = "Eldridge, Iowa"
30 L = LEN(A$)
40 PRINT "Length of string is",L,"characters"
50 PRINT A$

>RUN
Length of string is 14 characters
Eldridge, Iowa

READY - RAM 1
>
```

```
READY - RAM 1
>C$="hello"

>PRINT C$, LEN(C$)
hello  5

>
```

LET

Assigns the value of an expression to a variable.

{LET} var = expr

Where **var** is the name of any type of variable that is to be assigned the value of the following **expr.**

Discussion As shown in the syntax, LET is an optional word; the equal sign alone is sufficient for assigning an expression to a variable name.

Examples

```
10 LET D = 12
20 PRINT D

>RUN
12
```

```
10 LET A$ = "Oak "
20 LET B$ = "Street"
30 PRINT A$+B$

>RUN
Oak Street
```

LIST

Lists all or part of a program.

Command mode only

LIST {# port#,}{line#1} {,} {line#2}

Where **port#** is an optional output port and **line#** is any valid line number in the program.

Discussion

You can use the LIST command to list all or part of your program on the screen of your PC or of the Workstation (although the latter is not very useful).

LIST	List all program lines
LIST n	List line “n” only
LIST n,	List from line “n” to the end of the program
LIST ,n	List from the beginning to line “n”
LIST m,n	List from line “m” to line “n”

Examples

```
>LIST
10   SDIM A$(20) : REM Dimension string
20   A$ = "Ann Arbor, Michigan"
30   L = LEN(A$)
40   PRINT "Length of string is",L,"characters"
50   PRINT A$

READY - RAM 1
>
```

```
>LIST 10
10   SDIM A$(20) : REM Dimension string

READY - RAM 1
>
```

```
>LIST ,30
10   SDIM A$(20) : REM Dimension string
20   A$ = "Ann Arbor, Michigan"
30   L = LEN(A$)

READY - RAM 1
>
```

```
>LIST 20,40
20   A$ = "Ann Arbor, Michigan"
30   L = LEN(A$)
40   PRINT "Length of string is",L,"characters"

READY - RAM 1
>
```

```
>LIST 20,
20   A$ = "Ann Arbor, Michigan"
30   L = LEN(A$)
40   PRINT "Length of string is",L,"characters"
50   PRINT A$

READY - RAM 1
>
```

LOCATE

Positions the cursor on the display.

LOCATE {row#}, {column#}, {cursor}

Where **row#** is the line (1 - 2), **column#** is the display column (1 - 20), and **cursor** is the style of cursor displayed (0 = none; 1 = flashing box; and 2 = underline).

Discussion The LOCATE command positions the cursor on the display and optionally selects the kind of cursor displayed. Any of the parameters may be omitted.

Examples The following example places a block cursor in row 2, column 4.

```
10 LOCATE 2,4,1 : REM Row 2, column 4, block cursor
>RUN

READY - RAM 1
>
```

The following example places the cursor in column 3, and leaves the row the same.

```
10 LOCATE ,3

>RUN

READY - RAM 1
>
```

LOG

Returns the natural logarithm.

LOG(expr)

Where **expr** is any number or numeric expression greater than zero.

Discussion

The natural logarithm is the logarithm to the base e (2.718281828). LOG and EXP are inverse functions. Therefore the LOG of EXP(x) is x.

To calculate the logarithm in any other base, use the formula $\log_b(x) = \log(x)/\log(b)$.

Examples

```
>PRINT LOG(34.67)

 3.545875

>
```

```
10 A = EXP(14)
20 X = LOG(A)
30 PRINT X

>RUN
14

READY - RAM 1
>
```

```
10 INPUT "Enter number",A
20 X = LOG(A) : REM Calculate natural log of A
30 Z = X/LOG(10) : Convert to base 10 log
40 PRINT A,X,Z

>RUN
Enter number 34.67
34.67  3.545875  1.5399547

READY - RAM 1
>
```

MID$ (right side)

Returns a string from within another string.

MID$(sexpr1, iexpr1, iexpr2)

Where **sexpr1** is the string variable; **iexpr1** is the new string's starting point within the current string; and **iexpr2** is the length of the new string.

Discussion

On the right side of an equals sign, MID$ returns a string from within another. The string returned starts at the nth character of **sexpr,** where n = **iexpr1,** and continues for **iexpr2** characters. If **iexpr1** exceeds the length of **sexpr,** the resulting string is null (length = 0).

Examples

```
10 A$ = "Bettendorf"
20 B$ = MID$(A$,4,4)
30 PRINT B$

>RUN
tend

READY – RAM 1
>
```

MID$ (left side)

Places a string within another string.

MID$(svar, iexpr1 {,iexpr2})

Where **svar** is a string variable, **iexpr1** is an offset into the string and **iexpr2** is an optional length.

Discussion

On the left side of an equals sign, MID$ copies the string expression on the right of the equals sign into the string starting at a specified offset, **iexpr1**, and continuing for an optionally specified number of characters, **iexpr2**. If **iexpr1** or **iexpr2** equals zero, then **svar** doesn't change.

Examples

```
>A$ = "Bettendorf" : MID$(A$,7,2) = "BAD" :
PRINT A$
BettenBArf

>MID$(A$,9) = "rt" : PRINT A$
BettenBArt

>
```

MTOP

Returns or sets the highest memory address available.

MTOP = iexpr

Where **iexpr** is the highest memory address (plus 1) that is available to your BASIC program.

Discussion

MTOP sets or returns the highest memory address (plus 1) available to your BASIC program. The usual value of MTOP is 32767. If you try to set MTOP above the highest RAM location physically available, then BASIC issues a "BAD ARGUMENT ERROR."

The only reason to reduce MTOP is to use the space in order to save variables in a more efficient means than BASIC uses. For example, if you needed to save 5,000 integers, BASIC uses six bytes for each one and would need 30/40,000 bytes. But you could use the XBY() command to store integers in only two bytes, which means your total memory requirement would be 10,000 bytes.

Note When your program sets MTOP (using the format MTOP = x), BASIC clears any variables already declared. In other words, your program must set MTOP before doing anything else.

Examples

```
>PRINT MTOP
32767

>
```

NEW

Deletes the program currently in memory and clears all variables.

Command mode only

NEW

Discussion

NEW erases the program and clears all variables. You normally use it just before you download or start writing a new program.

Examples

```
READY - RAM 1
>NEW

>
```

NOT

Returns a 16-bit 1's complement.

NOT(iexpr)

Where **iexpr** is a valid integer between 0 and 65535 (0FFFFh).

Discussion NOT inverts each bit of an integer.

Examples

```
10 INPUT "Enter number ",A
20 B = NOT(A)
30 C = 65535 - (B)
40 PRINT B,C

>RUN
Enter number 34
65501  34

READY - RAM 1
>
```

ON ERROR GOTO

Enables error handling routine.

Run mode only

ON ERROR GOTO line#

Where **line#** is a line number in the program to which BASIC transfers control when it finds an error. (Setting **line#** to 0 disables error handling.)

Discussion

The ON ERROR GOTO statement tells BASIC to go to a routine if an error occurs. When an error occurs, whether BASIC is in the Run mode or the Command mode, BASIC executes the routine starting at line number **line#**.

When BASIC enters the error routine, it sets up the special variables ERR and ERL to hold the error number and line number of the error; in any other situation, the status of these variables is invalid. In order to resume program execution, the error handler must exit with a RESUME statement. See the description of RESUME for more information.

If the error handling routine terminates program execution (with the STOP, END, or ON ERROR GOTO 0 statement), BASIC immediately prints the error. If an error occurs within the error handler, BASIC terminates program execution and reports the error.

See the Appendix for a list of error codes.

Some communications errors can occur asynchronously, which means that ERL may not be valid. For example, if a receive buffer overflows, ERR will be correct but ERL will be the number of the line that the Workstation was executing when the overflow occurred. For an example, see the note under the description of the ERR and ERL statements earlier in this chapter. See the error message listing in the Appendix for a list of the asynchronous communications errors.

BASIC does not report asynchronous communications errors that occur during the error-handling routine. Your routine must RESUME before your program can detect that type of error.

During debugging, you may want to set up your error-handling routine to print a status message to the console and then immediately terminate program execution. In this case, most of your status message may be lost, because BASIC clears its communications buffers before printing an error message. In other words, BASIC erases any part of the message that still remains in the buffer. To ensure that the buffer empties, you may want to add a FOR – NEXT loop to idle a few seconds before ending the program.

You must be careful that your error-handler RESUMEs without disrupting control loops such as GOSUB – RETURN or FOR – NEXT. For example, if your error handler RESUMES outside a FOR – NEXT loop, the system's internal control stack will be left out of whack.

Examples

```
10 ON ERROR GOTO 100
20 INPUT A
30 B = 100/A
35 PRINT "100 divided by";A;" equals";B
40 GOTO 20
100 IF ERR <> 10 THEN END
110 PRINT "You have attempted to divide by zero!"
115 A=1
120 PRINT "We have substituted 1 for your input"
130 RESUME 30

>RUN
?34
100 divided by 34 equals 2.9411765
?0
You have attempted to divide by zero!
We have substituted a 1 for your input
100 divided by 1 equals 100
?
```

ON . . . GOSUB

Calls one of a list of subroutines.

Run mode only

ON expr GOSUB line#, line#. . . ,line#

Where **expr** is an expression that BASIC automatically rounds to an integer; BASIC passes control to the nth **line#,** where n = **expr** + 1.

Discussion

The value of the expression determines which line number in the list is the starting line of the subroutine that BASIC calls. For example, if the expression is 0, BASIC calls the first subroutine in the list. BASIC ignores any fractional part of the expression. The subroutine must end with a RETURN, at which point BASIC passes control to the next statement following the ON GOSUB statement.

Examples

```
10 INPUT"Enter number ",A
20 ON A GOSUB 100, 200, 300, 400
30 PRINT "DONE"
40 GOTO 10
100 PRINT"The answer to A was 0"
110 RETURN
200 PRINT"The answer to A was 1"
210 RETURN
30 PRINT"The answer to A was 2"
310 RETURN
400 PRINT"The answer to A was 3"
410 RETURN

>RUN
Enter number 2.5
The answer to A was 2
DONE

READY - RAM 1
>
```

ON . . . GOTO

Branches to one of a list of lines.

Run mode only

ON expr GOTO line#,line#. . . ,line#

Where **expr** is an expression that BASIC rounds to an integer; BASIC passes control to the nth **line#,** where n = **expr** + 1.

Discussion

The value of the expression determines which line number in the list to which BASIC transfers control. For example, if the expression is 0, BASIC goes to the first line number in the list. BASIC ignores any fractional part of the expression.

.

Examples

```
10 INPUT "Enter number ",A
20 IF A >= 4 THEN 40
30 ON A GOTO 100, 200, 30/400, 400
40 PRINT "You blew it, Jack"
50 END
100 PRINT "The answer to A was 0"
110 GOTO 10
200 PRINT "The answer to A was 1"
210 GOTO 10
30 PRINT "The answer to A was 2"
310 GOTO 10
400 PRINT "The answer to A was 3"
410 GOTO 10

>RUN
Enter number 2.5
The answer to A was 2
Enter number 4
You blew it, Jack

READY - RAM 1
>
```

ON TIME = . . . GOSUB

Sets up time-based interrupt handler.

Run mode only

ON TIME = expr GOSUB line#

Where **expr** is an integer setpoint for the timer; **line#** is the line number of the subroutine that handles timer interrupts.

Discussion

The ON TIME statement tells BASIC to call a subroutine after a specified period of time. This "interrupt" capability makes it easy for you to set up events to occur on a regular schedule. After BASIC executes an ON TIME statement, it must then execute a CLOCK1 statement in order to reset the timer and enable the ON TIME interrupt.

After executing the CLOCK1 statement, BASIC continually monitors the status of the timer, which is stored in a special variable called TIME. When TIME is equal to or greater than **expr,** BASIC calls the subroutine beginning at **line#**. The best resolution possible from the timer is 0.005 seconds, although you must plan on a "latency" or delay in handling the interrupt until BASIC completes the statement it is currently executing. This latency could be a very long time if BASIC happens to be in the middle of an INPUT statement when the timer times out.

Note If you want to use the ON TIME capability and still have the capability of using the INPUT statement, you should write a subroutine that emulates the INPUT statement. You can use INKEY$ #0 or INPUT$(0,#0) to test for characters received from the keypad and build response strings with some additional decoding logic
.

If you want BASIC to execute the timer subroutine on a regular schedule, your subroutine would have to reset TIME either at the beginning or the end of the routine, depending on your requirements, with another CLOCK1 statement. If you want your timer subroutine to cancel further timer interrupts, your subroutine should contain a CLOCK0 statement. The subroutine can also change the timer setpoint with another ON TIME statement. In any event, your timer subroutine must end with a RETI statement, not a RETURN statement.

Examples

This demonstration calls the timer subroutine at 1.105 second intervals:

```
10 PRINT USING "#####.###"
20 ON TIME = 1.105 GOSUB 100
30 CLOCK1 : REM Reset timer and enable interrupts
40 PRINT TIME ; CR ;
50 FOR Z = 1 TO 30 : NEXT
60 GOTO 40
100 PRINT TIME
110 CLOCK 1 : RETI
```

This demonstration calls the subroutine every second with virtually no accumulation of error:

```
10 PRINT USING "#####.###"
20 T = 1 : ON TIME = T GOSUB 100
30 CLOCK 1 : REM Reset timer and enable interrupts
40 PRINT TIME ; CR ;
50 FOR Z = 1 TO 30 : NEXT
60 GOTO 40
100 PRINT TIME
110 T = T + 1 : IF T >= 65536 THEN CLOCK 1
120 ON TIME = T GOSUB 100 : RETI
```

OPEN COM

Declares a port's communication parameters.

OPEN "port: {,param1} {,param2} . . . {,paramN} " {AS #alias]

Where **port** is the hardware port designation followed by any combination and sequence of optional parameters as specified in the table below, followed by an optional designation of an alias. A full discussion of each parameter follows the table.

Because the data between quotation marks is actually a string, you can use a string variable. For example, OPEN A$ is a valid statement, provided A$ is in the correct format.

The display and keyboard are also considered to be a port; the following table summarizes the port names and numbers:

Name	Number	Description
COM0	0	Keypad and display
COM1	1	COM1 serial port

There are three "default" columns in the following table because there are three situations under which default conditions can occur:

Initial Default Refers to the status of the parameter the first time you power up.

OPEN Default Refers to the status of the parameter if you omit it. For example, omitting the "TD" parameter sets up the port as if you entered "TD100"; the only way to disable the TD parameter is to use "TD0" in your OPEN statement.

Param Default Refers to the default value of the parameter if you omit the optional number that follows the letters.

Caution There are some combinations of parameters that could cause unpredictable behavior. Within the same OPEN statement, do not use CS with TX; in other words, don't mix hardware handshaking and software handshaking. Some parameters are mutually exclusive; you can select only one of RS, RH, or RO; WA or SC; and CR or CL.

Para-meter	Valid Port	Description	Valid Values	Initial Default	OPEN Default	Param Default
Port	all	"COM0" = keypad/display "COM1" = first serial port	0 – 1			
Alias	all	Software port number; all program statements refer to this port by its alias, not by its actual hardware port number.	0 – 5		Hard-ware port #	
Rate	1	Baud rate (communications speed)	110, 300, 600,1200, 2400, 4800, 9600, 19200	9600	9600	
Parity	1	No, even, or odd parity	N,E,O	N	N	
Data	1	Data size (if you set COM1 parameters to 7,N,1 or 7,N,2, you *must* set the other device to 7,N,2)	7,8	8	8	
Stop	1	Stop bits	1,2	1	1	
IB	all	Enable buffered communications		IB		
RS	1	Assert RTS during transmit				
CSx	1	Wait for CTS before transmitting	1 to 255; 50 msec resolution; 0 = disabled			100
RX	1	Generate XON/XOFF handshake on receive; set time to wait before generating error	1 to 255; 50 msec resolution; 0 = disabled			100
TXx	all	Respect XON/XOFF handshake on transmit and set time to wait for XON	1 to 255; 50 msec resolution; 0 = disabled	TX100		100
TDx	all	Set time-out when waiting for input (for the INPUT and INPUT$ statements)	0 to 255; 50 msec resolution; 0 = disabled	TD100	TD100	100
ED	all	Disable "echo" on INPUT				
LF	all	Print line feed after carriage return		LF		
PSx	1	Translate parity errors to character of ASCII code "*x*"	0 to 255			7Eh ("~")
LLx	all	Prevent lines from exceeding *x*; to disable, set *x* = 0	0 to 255	LL0	LL0	0
CE	all	Allow [Ctrl]-C received on this port to cause a "break"		CE		
RH	1	Assert RTS when receive buffer empty				
RN	1	Never assert RTS				
RO	1	Assert RTS at the start of transmitting (and leave asserted when finished)				

The following parameters control only the Workstation's keypad and display, and are therefore valid only for COM0:

SC	0	Enable scrolling on display		SC		
WA	0	Enable display to "wrap around" from last character to first				
CR	0	Enable automatic carriage return at end of line on display				
CL	0	Enable automatic carriage return/line feed at end of line on display		CL		
IC	0	Enable IBM PC ASCII code emulation				
ARx	0	Enable keyboard auto-repeat and delay *x* * 50 msec to first repeat	0 to 255; 50 msec resolution	AR10		10
RDx	0	Enable keyboard auto-repeat and delay *x* * 50 msec between repeats	0 to 255; 50 msec resolution	RD2		2

Note The CS and TD communications errors can occur asynchronously, which means that ERL may not be valid. For example, if a receive buffer overflows, ERR will be correct but ERL will be the number of the line that the Workstation was executing when the overflow occurred. For an example, see the note under the description of the ERR and ERL statements earlier in this chapter.

Alias

The alias is an optional parameter that allows the software to refer to the port by a different number. For example, OPEN "COM1:" AS #2 sets up COM1 as port number 2. All further references to port 2, such as PR#2, IN#2, PRINT #2 and INPUT #2, refer to COM1.

The alias capability is not very useful for the Series 30/40 Workstation, because it has only one communications port. We suggest that you omit using the alias.

AR; Auto-Repeat Enable

This parameter enables the auto-repeat capability on the keypad, which allows the operator to hold a key down instead of pressing it repeatedly. Optionally following this parameter is the delay between pressing the key and the first repetition. The resolution of the delay time is 50 milliseconds.

For example, the parameter AR10 enables the auto-repeat function and configures the Workstation to start repeating after 500 milliseconds (50 milliseconds * 10). The RD parameter sets the speed of subsequent repeats; you can change the time between repeats from the default value of 100 milliseconds.

Warning You should *not* enable the auto-repeat function if there are any keys that cause your machinery to perform a potentially hazardous control operation.

CE; [Ctrl]-C Enable

The CE parameter enables the Workstation to react to an ASCII character code of 3 as an interrupt. On most computers, you can generate this code by holding down the [Ctrl] key while pressing the letter "C"; on the Workstation's keypad, you can generate this code by holding down [F1] and [↵] at the same time.

BASIC handles a [Ctrl]-C interrupt as an error: it either terminates program execution or goes to the routine specified by an ON ERROR statement.

CL; Enable CRLF

The CL parameter affects only the operation of the Workstation's display; if you include the CL parameter when opening COM0, then BASIC automatically moves the cursor to the first position of the next line after it prints a character in the last column.

If the cursor is already on the bottom row of the display, then the action of the screen depends on the WA and SC parameters. If the WA parameter is included, the cursor moves to the top line; if the SC parameter is included, BASIC moves the bottom line to the top, clears the bottom line, and leaves the cursor on the bottom line.

CR; Enable CR

The CR parameter affects only the operation of the Workstation's display; if you include the CR parameter when opening COM0, then BASIC automatically moves the cursor to the first position of the current line after it prints a character in the last column.

CS; Wait for CTS

If you specify the CS parameter, then the Workstation requires the port's CTS input to be asserted before transmitting. This is a type of "hardware handshaking" where you connect a control line from another device that indicates "ready to receive" to the "Clear To Send" input of the port.

This capability is useful when the port is transmitting to a device that cannot accept data at full speed. If the device has a hardware handshaking capability for its receiver, then you can wire its output to the port's CTS input.

When your program tries to transmit, the Workstation tests the CTS input; if CTS is not asserted, the Workstation starts a timer and continues to test CTS. If CTS fails to be asserted within the time limit, then BASIC issues a timeout error.

Optionally following the CS parameter is the setpoint for the timer; the resolution of this timer is 50 milliseconds. If you omit the timer setpoint, BASIC uses the default value of 100, which is 5 seconds. If you specify a setpoint of 0, then BASIC waits forever for CTS and never generates an error.

Data

You can specify the number of bits in each data byte as 7 or 8; the default value is 8. You must select the data size to match the data size of the device you connect to the port. If you have a choice, we recommend you set the device to a data size of 8 bits.

Note If you set COM1 to 7 data bits and no parity, then you must set the stop bits of the device connected to COM1 to 2, regardless of the number of stop bits you set for COM1.

ED; Echo Disable

The ED parameter indicates that the Workstation should not "echo" characters received on an INPUT statement.

Normally, the Workstation re-transmits every character to the current output port that it receives in response to an INPUT statement. This is very nice when receiving input from a person, but can be a problem when the port is receiving input from another intelligent device.

As an example, suppose the port is receiving data from a motion control system. In a typical exchange, the port may send a command to the motion controller to send back its status. The Workstation would then use an INPUT statement to receive this status information from the port.

With echoing enabled, the Workstation would receive the status information and send it back out to the motion controller (or whatever the current output device is). With echoing disabled, the Workstation would simply receive the status information.

IB; Enable Buffers

The IB parameter indicates that the Workstation should send and receive characters using 255-character buffers instead of 1-character buffers. The differences in operation are summarized in the table below:

	Opened with IB	Opened without IB
Receive	The Workstation can hold up to 255 characters in its buffer for eventual exchange with a BASIC program. When the buffer is full, the unit loses additional characters and BASIC generates an overflow error.	The Workstation can receive only one character at a time. If the program fails to take a character (using an INPUT, INPUT$, or INKEY$ statement) before a second arrives, the unit loses the second; BASIC generates no error.
Transmit	The program can place up to 255 characters in a buffer (using the PRINT statement); the Workstation transmits from the buffer while continuing to execute the program. If the program tries to PRINT when the buffer is already full, the Workstation's response depends on the status of the port. If the port is actively transmitting, then program execution stops until the buffer is sufficiently empty for the remaining characters. However, if handshaking has made the port inactive, BASIC returns a BUFFER FULL error.	If the port is actively transmitting, program execution halts while the Workstation transmits each character to be PRINTed. However, if handshaking has made the port inactive, BASIC returns a BUFFER FULL error.

Buffered communications are useful when the port is connected to another device that can transmit to the port at any time, because the port receives the characters into its buffer even if your program is not immediately ready to receive them. The Workstation also transmits via a buffer, which means that the user program does not have to wait while the port transmits.

You can use a special CALL to retrieve the current status of a buffer. After the CALL, your program must POP the status of the buffer into a variable (for example, CALL 40 : POP A returns A equal to the number of characters remaining in the receive buffer of the first port):

CALL	Function
40	#1 receive
41	#1 transmit

In many applications, buffered communications can be a problem. For example, if the port is connected to a device that "echoes," then that device sends back to the port every character it receives. With no buffers, however, the port loses those characters, except perhaps the first character, which the user program must discard.

When you enable buffered communications on COM0, the Workstation can accept up to 255 keypresses before its buffer overflows; without buffers, the Workstation remembers only the first keypress.

IC; IBM-Compatible

Inclusion of this parameter when opening COM0 causes the Workstation to display characters using the same ASCII codes as the IBM PC's "multilingual" code page, which is code page 850.

The Workstation's display supports only a few of the foreign language characters, so the IC parameter is not particularly helpful.

LF; Line Feed Enable

The LF parameter indicates that the Workstation should send a "line feed" character after every "carriage return" character.

After the first power-up, this parameter comes up enabled; in an OPEN statement, however, the default is disabled. In order to enable this capability in an OPEN statement, you must include the LF code.

The LF parameter is useful only when the port is transmitting to a relatively "dumb" device like a terminal. Many intelligent devices accept a carriage return as a terminating character and reject the line feed as an error.

LL; Set Line Length

You can specify the maximum length of a line with the LL parameter followed by the line length. When BASIC transmits (PRINTs) a character to that maximum position, it automatically transmits a carriage return. To disable this feature, you should simply omit this parameter. The LL parameter is primarily useful when transmitting to a printer.

Parity

You can select N for no parity, E for even parity, or O for odd parity. You must choose a parity selection that matches the device connected to the port. If you select even or odd parity and the port receives a character with the wrong parity, it generates an error.

If you have a choice, we recommend you select E or O, but then you'll probably need a routine to handle any parity errors.

Port

Every OPEN command must contain a designation for the hardware port you are opening. The previous table lists the valid hardware ports.

PS; Select Parity Character

The PS parameter specifies the ASCII code of the character that the Workstation substitutes for any characters it receives with a parity error. In other words, the Workstation can translate a parity error into another character instead of reporting the error.

For example, the parameter PS126 causes the Workstation to substitute the character "~" in place of any characters it receives with an error in parity.

Rate

You can specify a baud rate for the serial ports; the previous table lists the baud rates that the Workstation supports.

RD; Set Repeat Rate

The RD parameter enables the keypad's auto-repeat capability and sets the time between the second and subsequent repeats. The resolution of the time delay is 50 milliseconds.

You should see the description of the AR parameter for more information about the auto-repeat capability.

RH; Receive Handshake Enable

The RH parameter enables "hardware handshaking" on the port's receiver, which is useful when your program is unable to receive from another device at full speed.

The RH parameter causes the port to assert its RTS output whenever its receive buffer is less than 3/4 full. If you connect that RTS output to the CTS input of another device, the other device will transmit only when the Workstation is ready to receive.

As a practical matter, when you use the RH parameter you should also select buffered communications with the IB parameter.

RN; RTS Never On

The RN parameter causes the port to leave its RTS output off. This parameter allows you to use CALL 30/40 effectively, because any other choice for RTS handshaking renders CALL 30/40 ineffective.

RO; RTS On at Start

The RO parameter causes the port to assert its RTS output at the start of transmitting and to leave RTS asserted indefinitely. You can use CALL 30/40 to turn off RTS later.

RS; RTS On During Transmit

The RS parameter causes the port to assert RTS only when it has characters to transmit. If the OPEN statement omits the RH, RO, and RS parameters, then RTS remains asserted all the time.

You must include the RS parameter when you are using the port in an RS-422/485 application because RTS also enables the transmitter itself. When RTS is not asserted, the transmitter turns off, which allows another transmitter to communicate on the same wires.

RX; XON/XOFF on Receive

The RX parameter enables "software handshaking" using the XON/XOFF ([Ctrl]-S/[Ctrl]-Q) protocol when receiving.

When the Workstation's receive buffer becomes 75% full, the Workstation transmits an "XOFF" character to the transmitting device, which should stop sending. When the Workstation's receive buffer is emptied to less than 50% full, the Workstation sends an "XON" character to the transmitting device to re-start communications.

Your program should *not* enable RX if the other device does not support XON/XOFF, or the Workstation may operate in an unexpected manner.

As a practical matter, when you use the RX parameter you should also select buffered communications with the IB parameter.

SC; Scroll Enable

The SC parameter enables the Workstation's display to scroll when printing past the last character on the screen or before the first character. In most applications, you should include the CL parameter as well.

Stop

You can specify the number of stop bits that the port transmits after each data byte as 1 or 2; the default value is 1. You must select the data size to match the data size of the device you connect to the port. If you have a choice, we recommend you set the device's stop bits to 1. When you set the number of data bits for COM1 to 7 and set its parity to none, then you must set the number of stop bits for COM1 to 2, because COM1 doesn't support the combination 7,N,1.

TD; Set Input Delay

When you open a port with the TD parameter, BASIC starts a 5-second timer at the beginning of an INPUT or INPUT$ statement and generates an error if no input occurs during the time period.

You can optionally specify a different timer setpoint following TD. The unit of measure is 50 milliseconds; for example, "TD10" tells the Workstation to wait for input no longer than 500 milliseconds (0.5 second) before issuing an error message.

This capability is useful to detect a fault without leaving the program "hung" while waiting for input. If a time delay is enabled, then the Workstation can handle a "time-out" error with an error-handling routine accessed by an ON ERROR statement.

To disable the timer, just enter a setpoint of 0; for example: TD0.

TX; XON/XOFF for Transmit

The TX parameter configures the port for "software handshaking" using the XON/XOFF ([Ctrl]-S/[Ctrl]-Q) protocol when transmitting. After receiving an XOFF, the Workstation starts a 5-second timer; if it times out before the Workstation receives an XON, the Workstation generates an error.

You can optionally specify a different timer setpoint following TX. The unit of measure is 50 milliseconds; for example, "TX10" tells the Workstation to wait for XON no longer than 500 milliseconds (0.5 second) before issuing an error message.

Your program should disable the TX capability if the transmitting device does not support the XON/XOFF protocol. This is especially important if the transmitting device is sending "binary" data where an XON or XOFF character could be part of the data stream.

WA; Wrap Enable

When you OPEN COM0 with the WA parameter, the Workstation "wraps around" to the first position on the screen after it prints to the last position. In most applications, you should include the CL parameter as well.

Examples

```
10 OPEN "COM0:IB,CE,LF,CL,SC"
20 OPEN "COM1:IB,CE,LF,TX0"

>RUN

READY – RAM 1
>
```

PH0. and PH1.

Prints numbers in hexadecimal format.

PH0. {#port#,} {expr} {, expr} . . .
PH0. {#port#,} {expr} {; expr} . . .
PH1. {#port#,} {expr} {, expr} . . .
PH1. {#port#,} {expr} {; expr} . . .

Where **port#** is an optional port number to print to and **expr** is an expression of any type to print.

Discussion

The PH0. and PH1. statements operate just like PRINT, except that they print all numbers in hexadecimal format instead of floating point format. Consult the description of PRINT for details.

The PH1. statement prints hex numbers in 4-digit format followed by the letter H. The PH0. statement does not print leading zeroes unless the number is less than 10h.

Examples

```
>A = 34 : PH0.A : PH1.A
22H
0022H

>A = 100 : PH0.A : PH1.A
64H
0064H

>A = 255 : PH0.A : PH1.A
FFH
00FFH

>A = 256 : PH0.A : PH1.A
100H
0100H

>
```

```
>PH0.(66)
42H

>PH1.(66)
0042H

>PH0.(1000)
3E8H

>PH1.(1000)
03E8H

>
```

PI

Equals π (3.1415926).

PI

Discussion

PI is useful in many mathematical relations, such as the area of a circle, ($A = \pi r^2$). You can use PI as the constant in any formula requiring the value for (π).

You may wonder why PI equals 3.1415926 when it is actually closer to 3.1415927. The reason is that the SIN, COS and TAN functions are more accurate if the equation PI = PI/2 + PI/2 holds true.

Examples

```
10 INPUT "Enter radius of circle ",R
20 A = PI*(R^2)
30 PRINT "The area of the circle is ",A

>RUN
Enter radius of circle 2
The area of the circle is 12.56637

READY - RAM 1
>
```

PLEN

Returns the length of your program.

PLEN

Discussion

PLEN (Program Length) returns the length of your program in number of bytes.

Examples

```
10 INPUT A$
20 A = LEN(A$)
30 PRINT A
40 PRINT PLEN

>RUN
?Circle tap
10
34

READY - RAM 1
>
```

POP

Equates a variable to the number at the top of the argument stack.

POP var

Where **var** is equated to the number popped from the stack.

Discussion

POP sets **var** to the number at the top of the argument stack. See the description of PUSH for more information about the stack.
Some built-in CALLs return a result on the argument stack that your program must remove with the POP command. If there is no data on the argument stack, then the Workstation issues an A_STACK error.

Examples

```
10 CALL 40 : POP S
20 PRINT "Port #1 receive buffer size =", S

>RUN
Port #1 receive buffer size = 0

READY - RAM 1
>
```

POS

Returns the column number of the screen display's cursor.

POS

Discussion The POS statement returns the column number occupied by the cursor on the display. The column number varies between 1 and 20.

POS is a companion statement to CSRLIN, which returns the current line number of the cursor, which varies between 1 and 2.

Examples The following program continuously updates the time on the display independently of the rest of the program. Line 100 saves the current cursor position so that line 120 can restore it later.

```
10  CLS : PR# 0
20  ON TIME = 1 GOSUB 100 : CLOCK 1
30  FOR I = 1 to 99999
40  PRINT USING "#####";I;CR;
50  NEXT
60  GOTO 30/40
100 RS = CSRLIN : CS = POS
110 LOCATE 2,5 : PRINT TIME$;
120 LOCATE RS,CS
130 CLOCK 1 : RETI
```

PR#

Switches output to specified port.

PR# iexpr

Where **iexpr** is a port number between 0 and 4.

Discussion

PR# directs all output from PRINT statements to a specified port, although a PRINT statement can contain a port selection that temporarily overrides the PR# port selection.

The display and keyboard are also considered to be a port; the following table summarizes the port names and numbers:

Name	Number	Description
COM0	0	Keypad and display
COM1	1	COM1 serial port

Examples

```
10 CLS
20 PR#0: REM Select Workstation's display for
printing
30 PRINT "This is the Workstation's display"
40 PR#1: PRINT "This is on the computer screen"

>RUN
This is on the computer screen

READY - RAM 1>
```

```
This is the Workstat
ion's display
```

PRINT

Prints to an output device.

PRINT {#port#,} {expr} {, or ; expr} . . .
or
? {#port#,} {expr} {, or ; expr} . . .

Where **port#** is an optional port number and **expr** is any valid numeric and/or string expression. Multiple expressions must be separated by a comma or semicolon.

Discussion

You can use the PRINT statement to print any combination of characters, strings, and numbers to the display or to the communications ports. The following discussion describes the various options of this powerful command in detail.

Spaces between expressions

The PRINT statement can print one or more expressions, where each is separated by a comma or semicolon. The comma inserts a space between each expression, while the semicolon inserts no spaces. (Note that BASIC handles commas differently than most BASICs, which use the comma as an implied tab.)

Suppressing the carriage return/line feed

At its conclusion, the PRINT statement sends a carriage return to the current output device. By default, the Workstation sends a line feed after a carriage return, although you can use the OPEN statement to defeat this feature (just omit “LF” from the parameter list). You can defeat printing the carriage return by ending the statement with a semicolon or a comma.

Printing numbers

BASIC prints positive floating point numbers with a leading space and negative floating point numbers with a leading minus sign. The only way to eliminate the leading space is to convert the number to a string with the STR$ statement. You can adjust the output format of floating point numbers with the USING statement.

BASIC prints hexadecimal numbers without leading or trailing spaces, but you must use the alternative print statements PH0. or PH1.

Print USING format

A PRINT statement can contain one or more USING statements that apply only to the following floating point numbers in the current statement. (However, if USING appears alone in a PRINT statement, then that output format becomes the default format for all subsequent PRINT statements; consult the description of PRINT USING next in this chapter for more information.)

Printing multiple spaces or tabs

BASIC can print multiple spaces using the SPC operator; for example, SPC(10) tells the Workstation to print ten spaces. Don't forget, however, that if the SPC operator is both preceded and followed by a comma, BASIC prints a total of 12 spaces.

BASIC can also skip to another column using the TAB operator. For example, TAB(10) tells the Workstation to print enough spaces to move the cursor to column 10 (unless the cursor is already past column 10).

Printing strings

The Workstation can print any character or group of characters if they are enclosed in double quotation marks. In order to print a double quotation mark or "control" character with an ASCII value of less than 20h, however, you must use the CHR$ function. For example, to print a double quotation mark, your program would read "PRINT CHR$(34)."

Output port

The output of the PRINT statement goes to the device selected by the most recent PR# statement. However, you can override this selection for the current PRINT statement by specifying a port number first. For example, PRINT #0, "TEST" sends the string "TEST" to the Workstation's display.

The display and keyboard are also considered to be a port; the following table summarizes the port names and numbers:

Name	Number	Description
COM0	0	Keypad and display
COM1	1	COM1 serial port

Examples

```
>PRINT 10 * 3
 30

>PRINT "I AM FINE"
 I AM FINE
```

```
10 INPUT "Enter temperature",A
20 PRINT "The temperature is ";A;"degrees"
30 PRINT "The temperature is ",A,"degrees"
40 PRINT "The temperature is ";A,"degrees"
50 PRINT "The temperature is ",A;"degrees"

>RUN
Enter temperature 78
The temperature is 78degrees
The temperature is  78 degrees
The temperature is 78 degrees
The temperature is  78degrees

READY - RAM 1
>
```

```
10 INPUT "ENTER NUMBER - ",A
20 INPUT "ENTER SECOND NUMBER - ",B
30 C = A + B : D = A * B
40 PRINT C,D

>RUN
ENTER NUMBER - 12
ENTER SECOND NUMBER - 44
56 528

READY – RAM 1
>
```

PRINT USING

Sets up format for printing numbers.

PRINT USING sexpr

Where **sexpr** specifies the format of printed variables; “0” specifies the general floating point format.

Discussion

The format string that follows the USING statement specifies to BASIC the output format for floating point numbers. The default format is “0”, which means that BASIC prints floating point numbers according to its standard rules. The # symbol indicates a character position; for example, the format string “##.##” indicates that BASIC should print two digits to the left of the decimal point and two digits to the right. If you use the letter “Z” in place of the “#” anywhere in the format string, then BASIC prints the leading zeroes.

The format string can include no more than eight “#” or “Z” symbols, because BASIC stores numbers with only 8 significant digits.

If a number is too big to print according to the format statement, then BASIC prints a question mark and then prints the number in the general floating point format.

Floating point format (USING “0”)

The printed appearance of a number in the floating point format depends on the size of the number. If the number is between 0.000001 and 10,000,000, then BASIC prints it just as it appears here. BASIC prints any number outside that range in exponential format, whose general format is “n Ee” where “n” is up to 8 significant digits of the number and “e” is the exponent of the number.

BASIC prints positive numbers with a leading space and negative numbers with a leading minus sign. To eliminate the leading space, your program must first convert the number to a string with the STR$ command.

A PRINT statement that contains only a USING statement (followed by the format string, of course) does not actually print; instead, it simply establishes the format as a “global” format that applies to all subsequent PRINT statements. A PRINT statement can override the global format by including a USING statement that applies only to the remainder of the PRINT statement.

Examples

The comma following the format statement prints a space before printing the number; to suppress this, you should use a semicolon instead of a number. Line 70 shows how to use the STR$ command to eliminate the leading space for positive numbers.

```
10 PRINT PI
20 PRINT USING "###.##z", PI
30 PRINT USING "#.#######", PI
40 A$ = "#.##" : PRINT USING A$; PI
50 PRINT USING "##.##",PI*100
60 PRINT PI * 10 ^ 10
70 PRINT STR$(PI)

>RUN
 3.1415926
 003.141
  3.1415926
 3.14
 ? 314.15926
 3.1415926 E+10
3.1415926

READY - RAM 1
>
```

PUSH

Places an expression on the argument stack.

PUSH aexpr

Where **aexpr** is an integer or floating point number and is pushed onto the argument stack.

Discussion

BASIC maintains a “stack” where it saves the intermediate results of various calculations. The PUSH command simply evaluates the subsequent numeric expression and leaves the result on the stack.

BASIC does not provide any standard statements that require a preceding PUSH. However, BASIC supports some undocumented CALLs that do require a PUSH.

RAM

Selects a program for editing or inserts a copy of a program.

RAM prog#
or
RAM prog1 = RAM prog2

Where **prog#**, **prog1**, and **prog2** are numbers less than 256, and **prog2** exists.

Discussion

When you use the command RAM*x* by itself, it selects program *x* for editing. If program *x* doesn’t exist, then BASIC creates a new empty program at the end of the directory. For example, if you type RAM 5 when there are only three valid programs in memory, BASIC creates a new, empty program at RAM 4 and selects RAM 4 for editing.

To insert a copy of one program before another, you would use the command format RAM*x* = RAM*y*, which inserts a copy of program *y* before program *x*. If program *x* doesn’t exist, then BASIC copies program *y* to the end of the directory. For example, if you type RAM 5 = RAM 1 when there are only three valid programs in memory, then BASIC makes a copy of RAM 1 at RAM 4.

Examples

```
READY - RAM 1
>RAM 2

READY - RAM 2
>DIR
RAM 1 (1000H,0020H) - PROGRAM 1
RAM 2 (1020H,0020H) - PROGRAM 2

READY - RAM 2
>RAM 4

READY - RAM 3
>DIR
RAM 1 (1020H,0020H) - PROGRAM 1
RAM 2 (1040H,0020H) - PROGRAM 2
RAM 3 (1060H,0005H) -

READY - RAM 3
>RAM1 = RAM 2

READY - RAM 3
>DIR
RAM 1 (1000H,0020H) - PROGRAM 2
RAM 2 (1020H,0020H) - PROGRAM 1
RAM 3 (1040H,0020H) - PROGRAM 2
RAM 4 (1060H,0005H) -

READY - RAM 3
>
```

REACT

Specifies the start-up action after reset.

REACT {par} {,par} . . . {,par}

Where **par** is a parameter (R, RAMx, C or P) that alters the Workstation's start-up actions.

Discussion

REACT (REset ACTion) allows you to set up the Workstation's start-up sequence. Typically, you use this command to make the Workstation automatically RUN its program after reset.

Each REACT command replaces any previous REACT commands, except that the system remembers REACT Cx commands. You can cancel all but REACT Cx by entering the REACT command followed by no parameters.

If no parameters are listed, then the Workstation performs its default functions. The default sequence after reset is:

1. Go to the Command mode.
2. Select the default port as the console port (COM1).

Note If you want your program to auto-start, then you must use the REACT command *before* you copy your program to ROM.

R; Run Program

This parameter causes the Workstation to automatically run a program after power-up.

```
READY - RAM 1
>REACT R

READY - RAM 1
>ROM = RAM 1

READY - RAM1
>
```

C {#}; Console

This parameter tells the Workstation which port is the console port. For example, REACT C2 tells the Workstation to use port #2 as the console. You should be careful, however, because this port selection refers to the alias and not the actual hardware port number.

P; Protect

When you use it with the R parameter, the P parameter locks BASIC in the run mode; if program execution terminates for any reason, BASIC simply re-runs the program.

Note If you use the P parameter, you should provide a “back door” to terminate program execution or you will be unable to perform troubleshooting. If you find yourself unable to terminate a program, then you must download new firmware, which erases everything.

READ

Reads values from a DATA statement and assigns them to variables.

READ var {,var} {,var} . . . {,var}

Where **var** is any valid numeric or string variable.

Discussion

The READ statement gets its data from DATA statements. Every time BASIC READs a data, it points to the next item of data.

Unlike most BASICs, you must enclose DATA strings in quotation marks, as shown in the second example below.

Examples

```
10 FOR I = 1 TO 6
20 READ A(I)
30 PRINT A(I), : NEXT
40 DATA 23, 56, 125, 400, 530, 409

>RUN
23 56 125 400 530 409

READY - RAM 1
>
```

If insufficient data exists to fill the list, an ERROR: NO DATA message is issued. To re-read DATA statements from the beginning, use the RESTORE statement.

```
10 FOR I = 1 TO 8
20 READ A$(I)
30 PRINT A$(I),
40 IF I = 6 THEN RESTORE
50 NEXT
60 DATA "ABC","DEF","GHI","JKL","MNO","PQR"

>RUN
ABC DEF GHI JKL MNO PQR ABC DEF

READY - RAM 1
>
```

REM

Indicates that the rest of the line is only a remark.

REM {remark}

Where **remark** is any string of characters.

Discussion

BASIC ignores everything following the REM statement until the end of the line. Numbered REM statements occupy program memory.

If you write your program in a word processor, you should consider including REM statements without line numbers. When you download your program to the Workstation, the REM statements don't take any program memory because the Workstation doesn't save them.

Examples

```
10 INPUT "Enter first number ",A
20 INPUT "Enter second number ",B
30 PRINT A*B : REM MULTIPLY NUMBERS
40 PRINT A+B : REM ADD NUMBERS
50 PRINT A/B : REM DIVIDE NUMBERS
60 PRINT A-B : REM SUBTRACT NUMBERS
70 PRINT A^B : REM RAISE TO EXPONENT

>RUN
Enter first number 45
Enter second number 4
180
49
11.25
41
4100625
```

RENUM

Renumbers all or part of a program.

RENUM {new}{,inc}{,start}{,end}

Where **new** is the first new line number, **inc** is the amount by which each subsequent line number will be increased, **start** is the old line number of the first line to renumber, and **end** is the old line number of the last line to renumber. The default parameters are 100,10,0,65535; if you omit any parameter, BASIC uses its default.

Discussion

The RENUM command is a handy way to renumber your program. You can renumber all or part of your program, depending on whether you specify any parameters.

You cannot use RENUM to re-arrange lines in your program. For example, you cannot renumber lines 100 through 199 so that they appear after lines 200 through 299; if you try, BASIC issues a BAD ARGUMENT error instead.

If the system runs out of memory while renumbering, it prints the error message OUT OF MEMORY. This normally would happen only if you have very little memory remaining and insufficient space remains for BASIC to store a table that holds the old line numbers and their corresponding new line numbers.

Note In very rare cases, it's possible for BASIC to run out of memory while actually renumbering. This could happen if free memory is almost gone and a new line number makes a line longer than it used to be. For example, if GOTO 10 becomes GOTO 1000, then the line becomes two bytes longer. If there is insufficient free memory to store the longer line, BASIC issues an error message and stops renumbering in the middle, which leaves your program scrambled beyond use.

To avoid this problem, we recommend that you save your program on disk before renumbering.

Examples

```
>LIST
 10 PRINT "LINE 10" : ON ERROR GOTO 40
 20 PRINT "LINE 20" : GOTO 30
 30 PRINT "LINE 30" : END
 40 RESUME 40

READY - RAM 1
>RENUM

>LIST
 100 PRINT "LINE 10" : ON ERROR GOTO 130/40
 110 PRINT "LINE 20" : GOTO 120
 120 PRINT "LINE 30" : END
 130 RESUME 120

READY - RAM 1
>
```

RESTORE

Resets the pointer to the DATA items.

RESTORE {line#}

Where **line#** is any valid line number.

Discussion

RESTORE resets the pointer used with READ and DATA statements. After RESTORE, the READ starts with the first DATA statement again.

If you specify an optional line number after RESTORE, then BASIC resets the pointer to the first DATA statement at or following the line number.

Examples

```
10   GOSUB 100
20   RESTORE
30   GOSUB 100
40   RESTORE 1010
50   GOSUB 100
60   END
100  FOR I = 1 TO 4 : READ X : PRINT X, : NEXT :PRINT
110  RETURN
1000 DATA 1,2
1010 DATA 3,4,5,6

>RUN
 1 2 3 4
 1 2 3 4
 3 4 5 6

READY - RAM 1
>
```

RESUME

Continues program execution at the end of error-handling.

Run mode only

RESUME {line#}
or
RESUME {NEXT}

Where **line#** is the line number where BASIC continues program execution at the conclusion of the error-handling routine.

Discussion

If you use the ON ERROR capability, your error-handling routine must terminate with a RESUME command in order to continue program execution.

If there is no **line#**, BASIC continues execution where the error occurred; if RESUME is followed by NEXT, BASIC continues execution at the statement following the statement that caused the error.

Examples

```
5 ON ERROR GOTO 100
10 INPUT "Enter first number ",A
20 INPUT "Enter second number ",B
30 PRINT A*B : REM MULTIPLY NUMBERS
40 PRINT A+B : REM ADD NUMBERS
50 PRINT A/B : REM DIVIDE NUMBERS
60 PRINT A-B : REM SUBTRACT NUMBERS
70 PRINT A^B : REM RAISE TO EXPONENT
80 END
100 REM ERROR HANDLING
110 IF ERR = 1 THEN END
120 IF ERR = 10 THEN PRINT "Can't divide by
zero"
130 RESUME 60

>RUN
Enter first number 45
Enter second number 0
0
45
Can't divide by zero
45
1

READY - RAM 1
>
```

RETI

Returns from ON TIME handling routine.

Run mode only

RETI

RETI has no additional parameters.

Discussion

The RETI statement terminates an ON TIME service routine. The RETI statement does the same thing as RETURN except that it also clears a software interrupt flag so BASIC can handle subsequent interrupts.

Examples

```
10 ON TIME = 5 GOSUB 100
20 CLOCK 1
30 FOR I = 1 TO 50 : NEXT
40 PRINT "Waiting for interrupt",CR,
50 GOTO 30
100 PRINT "Interrupt evoked at",TIME, "seconds"
120 CLOCK 1 : RETI

>RUN
Waiting for interrupt
Interrupt evoked at 5.02 seconds
Interrupt evoked at 5.02 seconds
```

The above example works because of the RETI statement.

```
10 ON TIME = 5 GOSUB 100
20 CLOCK 1
30 FOR I = 1 TO 50 : NEXT
40 PRINT "Waiting for interrupt",CR,
50 GOTO 30
100 PRINT "Interrupt evoked at",TIME,
"seconds"
120 CLOCK 1 : RETURN : REM THIS IS THE WRONG
RETURN

>RUN
Interrupt evoked at 5.02 seconds
Waiting for interrupt
```

The above program does not work properly (only one interrupt executed) because the RETURN statement was used instead of the RETI statement.

RETURN

Returns from a subroutine.

```
RETURN
```

Discussion RETURN terminates a subroutine originally called with GOSUB. If the subroutine is an ON TIME service routine, then you must use RETI instead of RETURN.

Examples

```
10 FOR I = 1 TO 10
20 IF I = 6 THEN GOSUB 100
30 PRINT I
40 NEXT
50 PRINT : "The job is finished"
60 END
100 PRINT "We are at number 6 now"
110 RETURN

>RUN
1
2
3
4
5
We are at number 6 now
6
7
8
9
10
The job is finished

READY - RAM 1
>
```

RIGHT$

Returns the rightmost characters of a string.

RIGHT$(sexpr,iexpr)

Where **sexpr** is a string and **iexpr** is the number of characters in the result.

Discussion

If **iexpr** is greater than the number of characters in the string, the entire string is returned. If **iexpr** is zero, a null string (length 0) is returned.

Examples

```
10 A$ = "Phoenix"
20 B$ = RIGHT$(A$,3)
30 PRINT B$

>RUN
nix

READY - RAM 1
>
```

```
>PRINT RIGHT$("Phoenix",3)
nix

>
```

RND

Returns a random number between 0 and 1.

```
RND
```

Discussion

The RND statement returns a random number between 0.0000000 and 0.9999999. The Workstation always generates random numbers in the same sequence after a power-up.

The random number "seed" is a 16-bit binary number, and the random numbers that the Workstation generates are in the range of 0/65535 to 65535/65535.

Examples

```
10 FOR I = 1 TO 5
20 PRINT INT(RND*100)
30 NEXT

>RUN
57
88
95
3
36

READY - RAM 1
>
```

ROM

Copies a program in RAM to permanent memory in Flash.

ROM = RAM prog#

Where **prog#** is a number less than 256.

Discussion The ROM command allows you to copy a program in RAM to the Workstation's permanent memory in Flash.

Note To set up your program to start running automatically after power-up, you must use the REACT R command *before* you use the ROM command to copy your program to Flash.

Examples

```
READY - RAM 1
>DIR
RAM 1 (1000H,0020H) - PROGRAM 1

READY - RAM 1
>REACT R

READY - RAM 1
>ROM = RAM 1

READY - RAM 1
>DIR
RAM 1 (1000H,0020H) - PROGRAM 1
ROM 1 (A000H,0020H) - PROGRAM 1

READY - RAM 1
>
```

RUN

Starts execution of a program.

RUN {line#} {RAM prog#}

Where **line#** is the line number at which to begin execution and **prog#** specifies which program to run.

Discussion

RUN clears all variables and starts program execution. You can optionally specify a line number after RUN in order to begin execution with a line other than the first, or you can specify a program number to run a program other than the current. The valid combinations follow:

RUN	Runs the current program starting at the first line
RUN *line#*	Runs the current program starting at *line#*
RUN RAM*x*	Runs program *x* starting at its first line
RUN *line#* RAM*x*	Runs program *x* starting at *line#*

Examples

```
10 PRINT 1,
20 PRINT 2,
30 PRINT 3

>RUN
 1 2 3

READY - RAM 1
>RUN 20
 2 3

READY - RAM 1
>
```

SDIM

Sets the maximum length of a string.

SDIM var$(iexpr)
or
SDIM = iexpr

Where **var** is the string name and **iexpr** is the length of the string (between 1 and 254)

Discussion

The default length of a string variable is 10 characters. You can assign another length using SDIM, but you must do so before your program refers to the string. If your program contains an SDIM that sets the length of a string that your program has already referenced, then BASIC prints a REDIMENSION error message.

You can change the default length of string variables by using the second form of SDIM shown above. For example, to change the default string length to 20, your program should include the statement “SDIM = 20.” This statement does not change the lengths of any strings that your program already defined.

SDIM sets up the length only of a scalar (non-array) string; to establish the length of a string array, you must use the DIM statement.

You can use SDIM to set up more than one variable at a time; see line 10 below for an example.

Examples

```
10 SDIM A$(20), B$(35), C$(20), D$(50)
20 A$ = "Christmas time"
30 B$ = " is for children,"
40 C$ = " and adults too."
50 D$ = A$+B$+C$
60 PRINT D$

>RUN
Christmas time is for children, and adults too.

READY - RAM 1
>
```

The following program example produces an error at line 120 because it tries to change the length of string variable A$ after your program has already created the string.

```
10 PRINT A$
     .
     .
120 SDIM A$(30)

>RUN
ERROR: REDIMENSION - IN LINE 120

120 SDIM A$(30)
-------- X
```

SGN

Returns the sign.

SGN(expr)

Where **expr** is a numeric expression.

Discussion

SGN returns a 1 if the argument is positive; a 0 if the argument is zero; and a -1 if the argument is negative.

Examples

```
10 A=0: B=-4.56: C=56.98
20 PRINT SGN(A), SGN(B), SGN(C)

>RUN
0  -1  1

READY - RAM 1
>
```

SIN

Returns the sine.

SIN(expr)

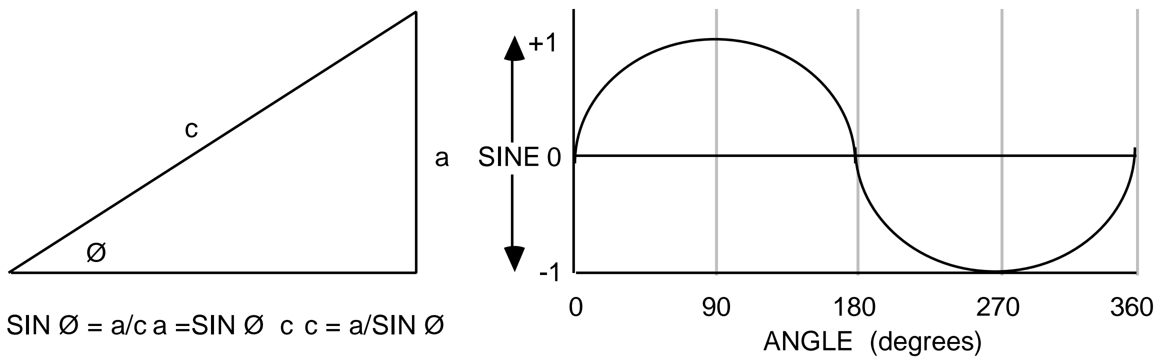
Where **expr** is a number or valid arithmetic calculation in radians.

Discussion

Like all numeric functions, SIN may appear on the right of an assignment statement, within a PRINT statement, and as part of an arithmetical expression. The argument for the SIN function is a value expressed in radians; divide degrees by 57.29577 to convert degrees to radians.

Trigonometric Relationships

The sine of a triangle is the length of the opposite side divided by the length of the hypotenuse side (a/c). See the figure below for the relationships between SIN Ø, side a, and side c.



Examples

```
10 INPUT "Enter angle in degrees ",A
20 INPUT "Enter length of side a ",L
30 R = A/57.29577 : S = SIN(R) : REM Calculate sine
40 C = L/S : REM Calculate length of side C
50 PRINT "Length of side c is ";C

>RUN
Enter angle in degrees 30
Enter length of side a 70
Length of side c is   140

READY - RAM 1
>
```

SPC

Prints spaces in a PRINT statement.

SPC(iexpr)

Where **iexpr** must be between 0 and 255.

Discussion SPC prints the number of spaces specified in **iexpr** in a PRINT statement.

Examples

```
10 A$="over" : B$="there"
20 C$="insure" : D$="vehicles"
30 PRINT A$; SPC(15); B$
40 PRINT C$; SPC(15); D$

>RUN
over                   there
insure                   vehicles

READY – RAM 1
>
```

SQR

Returns the square root.

SQR(expr)

Where **expr** is a number equal to or greater than zero.

Discussion SQR returns the square root of the number.

Examples

```
10 FOR X = 1 TO 21 STEP 4
20 PRINT X, SQR (X)
30 NEXT

>RUN
1 1
5 2.236068
9 3
13 3.6055513
17 4.1231057
21 4.5825757

READY – RAM 1
>
```

ST@

Stores one or more variables in memory.

ST@ iexpr, var1 {,var2} . . . {,varx}

Where **iexpr** is the starting memory location to hold variable **var1** and optional variables **var2** through **varx**.

Discussion

The ST@ statement stores one or more variables starting at a specified location within memory. This is useful when your program must save numbers and/or strings in either memory for later retrieval.

There is a special variable called VAD that holds the last address plus 1 used by your most recent ST@ (or LD@) command. This is especially useful when storing strings, because it helps you eliminate the bookkeeping that is otherwise necessary to keep track of the next available location.

See the description of LD@ earlier in this chapter for a complete example of using ST@.

STOP

Terminates program execution.

Run mode only

STOP

Discussion

When BASIC encounters a STOP statement, it terminates program execution and prints the line number where it stopped. From the Command mode, you can enter the CONT command to resume program execution immediately after the STOP.

Examples

```
10 FOR X = 1 TO 21 STEP 4
20 PRINT X, SQR (X)
30 NEXT
40 STOP
50 FOR R = 1 TO 20
60 PRINT R,
70 NEXT

>RUN
1 1
5 2.236068
9 3
13 3.6055513
17 4.1231057
21 4.5825757
STOP - IN LINE 40

READY - RAM 1
>CONT
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20

READY - RAM 1
>
```

STR$

Returns the string form of a number.

STR$(expr)

Where **expr** is any numeric expression.

Discussion

The STR$ function is the inverse of the VAL function; it turns a number into a string. This is handy when you want to print a number without printing any leading spaces.

STR$ converts expr to a string according to the current PRINT USING format, except that it omits leading spaces.

Examples

```
10 INPUT "Enter number ",A
40 A$=STR$(A) : REM Convert A to a string in A$
60 PRINT USING "#z#.##"
70 B$=STR$(A) : REM Convert according to USING format.
80 PRINT A : PRINT A$ : PRINT B$

>RUN
Enter number 12.578
 12.578
12.578
012.57

READY - RAM 1
>
```

TAB

Moves the cursor to the position specified.

TAB(iexpr)

Where **iexpr** returns a number between 0 and 255.

Discussion Within a PRINT statement, TAB tells BASIC to send enough spaces to move the cursor to the column specified.

Examples

```
10 A$="over" : B$="there"
20 C$="insure" : D$="vehicles"
30 PRINT A$; TAB(15); B$
40 PRINT C$; TAB(15); D$

>RUN
over                there
insure              vehicles

READY - RAM 1
>
```

Compare the printed result of this example to the example shown in the SPC description.

TAN

Returns the tangent.

TAN(expr)

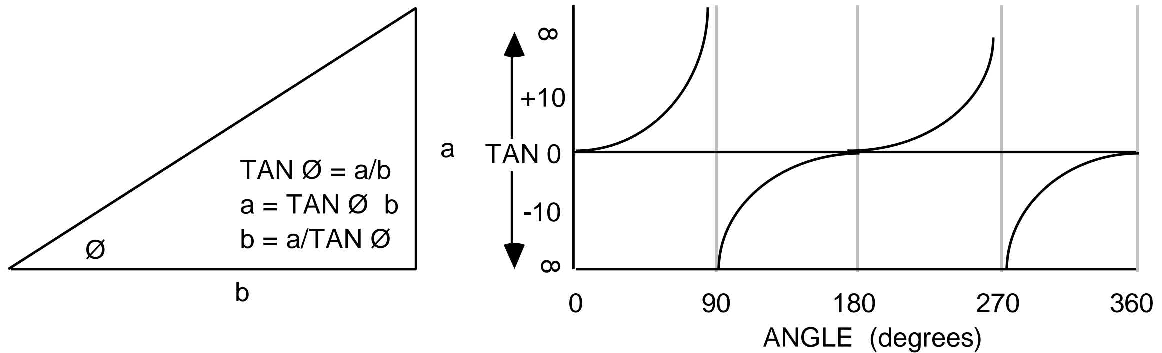
Where **expr** is a number or valid arithmetic calculation in radians.

Discussion

Like all numeric functions, TAN may appear on the right of an assignment statement, within a PRINT statement, and as part of a logical expression. The argument for the TAN function is a value expressed in radians; divide degrees by 57.29577 to convert degrees to radians.

Trigonometric Relationships

The tangent of a triangle is the length of the opposite side divided by the length of the adjacent side (a/b). See the figure below for the relationships between TAN Ø, side a, and side b.



Examples

```
10 INPUT "Enter angle in degrees ",A
20 INPUT "Enter length of side a ",L
30 R=A/57.29577 : T=TAN(R) : B = L/T
40 B=B*100 : B=B+0.5 : B=INT(B) : B=B/100 : REM ROUND
50 PRINT "Length of side b is ";B

>RUN
Enter angle in degrees 73
Enter length of side a 536
Length of side b is   163.87

READY - RAM 1
>
```

TIME

Returns or sets the built-in TIME variable.

```
TIME
```

Discussion

BASIC has a special variable called TIME that increases every 5 milliseconds. When TIME reaches the value of 65,535.995 seconds, it rolls over to 0 instead of going to 65,536.

The CLOCK 1 command resets TIME to 0, and if BASIC has already executed an ON TIME command, then CLOCK 1 also enables the ON TIME interrupt. (CLOCK 0 has no effect on the TIME variable; it simply disables the ON TIME interrupt.)

Your program can also set the value of TIME by simply equating it to a numeric value.

Consult the discussion of “ON TIME” for more information.

Examples

The example below shows how your program can set the TIME variable but that it continually increases:

```
>TIME = 43.5 : PRINT TIME, TIME

43.5  43.51
>
```

TIME$

Returns or sets the time of day.

TIME$ = “hh : mm : ss”

Where **hh : mm : ss** is hours, minutes and seconds respectively in 24 hour format.

Discussion

On power-up, the Workstation sets TIME$ to “00:00:00.” Your program can set TIME$ as shown in the box above. The Workstation considers missing parameters to be zero; for example, if TIME$ is “11:25:30/40”, the statement TIME$ = “12” sets the time to 12:00:00.

The Workstation is accurate only to within a few minutes every day.

Examples

```
>TIME$ = "15:34:22"

READY - RAM 1
>
```

```
>TIME$ = "15:34:22"

READY - RAM 1
>LIST
10   L$ = LEFT$(TIME$,2)
20   M$ = MID$(TIME$,4,2)
30   R$ = RIGHT$(TIME$,2)
40   PRINT "The time is ";L$;" hours, "; M$; " minutes "

READY - RAM 1
>RUN
The time is 15 hours, 34 minutes

READY - RAM 1
>
```

TROFF

Turns off tracing of program execution.

TROFF

Discussion TROFF turns off the tracing of program execution; consult the description of TRON for more information.

Examples

```
10 FOR I 1 TO 5
20 IF I = 3 THEN GOSUB 100
30 PRINT I
40 NEXT
50 END
100 PRINT "This is GOSUB"
110 RETURN

>TRON
>RUN
[10] [20] [30] 1
[40] [20] [30] 2
[40] [20] [100] This is GOSUB
[110] [30]  3
[40] [20] [30] 4
[40] [20] [30] 5
[40] [50]

READY - RAM 1
>TROFF
>RUN
1
2
This is GOSUB
3
4
5

READY - RAM 1
>
```

TRON

Enables tracing of program execution.

TRON

Discussion TRON tells BASIC to print to the console the line number (enclosed within brackets) of each statement it executes. The TROFF command turns off the trace mode.

Examples

```
10 FOR I 1 TO 5
20 IF I = 3 THEN GOSUB 100
30 PRINT I
40 NEXT
50 END
100 PRINT "This is GOSUB"
110 RETURN

>TRON
>RUN
[10] [20] [30] 1
[40] [20] [30] 2
[40] [20] [100] This is GOSUB
[110] [30]  3
[40] [20] [30] 4
[40] [20] [30] 5
[40] [50]

READY - RAM 1
>
```

VAD

Returns last address plus 1 used by the most recent ST@ or LD@

VAD

Discussion VAD returns the last address plus 1 used by the most recent ST@ or LD@ command.

See the description of LD@ for more details and examples.

VAL

Converts a string to a number.

VAL(sexpr)

Where **sexpr** is a string expression whose contents are assumed to be numeric.

Discussion

VAL converts a string to a number (note that this is the reverse of the STR$ function). If the string does not start with a digit, it has a value of 0. If the string starts with a number but contains letters, VAL returns only up to the first non-numeric digit.

VAL can return the value of strings in hexadecimal format, but you should remember that a hex number ends with the letter H. If you want to convert a hexadecimal string that does not contain the H, you can use HVAL instead.

Examples

```
10 INPUT "Enter area ZIP Code ",Z$
20 GOSUB 500
30 IF VAL(Z$) = 61201 THEN PRINT "Rock Island, IL"
40 IF VAL(Z$) = 52748 THEN PRINT " Eldridge, IA"
50 IF VAL(Z$) = 52804 THEN PRINT "Davenport, IA"
60 IF VAL(Z$) = 61265 THEN PRINT "Moline, IL"
70 IF VAL(Z$) = 53115 THEN PRINT " Delavan, WI"
80 END
500 PRINT "The city is ",
510 RETURN

>RUN
Enter area ZIP Code 52748
The city is Eldridge, IA

READY - RAM 1
>
```

VARPTR

Returns the memory address of a variable.

VARPTR(var)

Where **var** is any variable name of any type.

Discussion

VARPTR returns the starting address in external memory of a variable. Your program can access external memory with the XBY() operator.
The storage format for various variable types is shown below:

Floating Point

Byte	Function
0	Most significant byte
1	Next most significant byte
2	Next least significant byte
3	Least significant byte
4	Sign (0 if positive, 1 if negative)
5	Exponent (2 to 129 are negative exponents -127 to -1; 130/40 is an exponent of 0; 131 to 255 are positive exponents 1 to 126).

Integer

Byte	Function
0	Least significant byte
1	Most significant byte
2	Sign (0 if positive, 1 if negative)
3	Equals 0 if integer is 0; else equals most significant byte OR'ed with least significant byte
4	Unused
5	Unused

String (scalar)

Byte	Function
0	Maximum allowed length of string plus 1
1	High byte of address where actual string is stored
2	Low byte of address where actual string is stored
3	Unused
4	Unused
5	Unused

The storage of the string itself begins with a length byte that is the actual length of the string (not the maximum length allowed). The bytes following are the ASCII characters for the string.

To find out the address of an array variable, **var** must refer to an element in the array. For example, VARPTR(A(1)) returns the address of the second element in the floating point array named A.

Note that VARPTR operates differently for strings. It returns the same value regardless of the array element:

String (array)

Byte	Function
0	Maximum allowed length of string plus 1
1	High byte of address where first string is stored
2	Low byte of address where first string is stored
3	Number of strings in array
4	Unused
5	Unused

Examples

```
>A = 10
>PRINT VARPTR(A)
 63479

>
```

VERSION

Returns the version number of the BASIC firmware.

VERSION

Discussion VERSION returns the version number of the BASIC firmware in the Workstation
.

Examples

```
>PRINT VERSION
 5.50

>
```

XBY

Retrieves or assigns a value to external memory.

XBY(iexpr)

Where **iexpr** returns a number between 0 and 65,535; if **iexpr** is less than 32,768 (8000h), then XBY refers to external data memory (RAM); otherwise, XBY refers to external code memory (Flash EPROM).

Discussion

XBY retrieves from or assigns a byte value to the external data memory at address **iexpr**. If your program refers to addresses at or above 32,768 (8000h), then XBY refers to code memory, which is Flash EPROM.

You must be careful not to use XBY to change code memory without careful study. Some of BASIC itself exists in the firmware space above 8000h, and BASIC stores your program at 0A000h in the same area. Please see the discussion of LD@ earlier in this chapter for more details.

Examples

```
>PH0. XBY(1000H)
12H

>
```

Chapter 11

Operators

Series 30/40 BASIC contains a complete set of arithmetic and relational operators. The generalized form of all arithmetic operators is as follows:

expr op expr
where **op** is one of the arithmetic operators

Precedence

BASIC scans an expression from left to right, performing operations of higher precedence first and equal precedence from left to right. The order of precedence for solving mathematical expressions is as follows:

1. Operators that use parentheses ( )
2. Exponentiation (^)
3. Negation (–)
4. Multiplication (*) and Division (/)
5. Addition (+) and Subtraction (–)
6. Relational Operators (=, <>, >, >=, <, <=)
7. Logical AND, OR, and INV
8. Logical XOR

A good rule of thumb to follow is "when in doubt, use parentheses."

Examples

```
10 A = 4 + 3 * 2
20 PRINT A

>RUN
 10
```

In the preceding example, BASIC first multiplies 3 by 2 and then adds 4.

```
10 A = 2 * (17 + 4^3)
20 PRINT A

>RUN
 162
```

In the example above, BASIC first performs exponentiation (4^3), adds that result to 17, and multiples that result by 2.

+ (addition)

Returns the sum of numbers or joins strings.

expr1 + expr2

Where **expr1** and **expr2** are any expressions. If both are strings, then BASIC joins string2 to the end of string1 and returns that combination as a single string.

Examples

```
>PRINT 3 + 5
  8

>A$ = "HELLO" : B$ = " THERE" : PRINT A$ + B$
HELLO THERE

>
```

```
10 INPUT "Enter first number",A
20 INPUT "Enter second number",B
30 C = A + B
40 PRINT : PRINT "The answer is ";C

>RUN
Enter first number 12
Enter second number 14

The answer is 26

READY - RAM 1
>
```

– (subtraction or negation)

Returns the difference.

expr1 – expr2
or
– expr3

Where **expr1**, **expr2**, and **expr3** are numeric expressions.

Examples

```
>PRINT 10 - 25
 -15

>PRINT - MTOP
 -8192
```

```
10 INPUT "Enter first number",A
20 INPUT "Enter second number",B
30 C = A - B
40 PRINT : PRINT "The answer is ";C

>RUN
Enter first number 22
Enter second number 14

The answer is 8

READY – RAM 1
>
```

* (multiplication)

Returns the arithmetic product of two expressions.

expr1 * expr2

Where **expr1** and **expr2** are any numeric expressions.

Examples

```
>PRINT 3 * 5
 15
```

```
10 INPUT "Enter first number",A
20 INPUT "Enter second number",B
30 C = A * B
40 PRINT : PRINT "The answer is ";C

>RUN
Enter first number 12
Enter second number 14

The answer is 168

READY - RAM 1
>
```

/ (division)

Returns the arithmetic quotient of two expressions.

expr1 / expr2

Where **expr1** and **expr2** are any numeric expressions. If **expr2** is zero, then BASIC issues a DIVIDE BY ZERO error (error code 10).

Examples

```
>PRINT 100 / 25
 4
```

```
10 INPUT "Enter first number",A
20 INPUT "Enter second number",B
30 C = A / B
40 PRINT : PRINT "The answer is ";C

>RUN
Enter first number 22
Enter second number 14

The answer is 1.571428

READY - RAM 1
>
```

^ (exponentiation)

Returns the arithmetic result of a number raised to an exponent.

expr ^ iexpr

Where **expr** is any numeric expression; **iexpr** is an integer between 0 and 255.

Examples

```
>PRINT 6 ^ 4
 1296
```

```
10 INPUT "Enter first number",A
20 INPUT "Enter exponent",B
30 C = A ^ B
40 PRINT : PRINT "The answer is ";C

>RUN
Enter first number 22
Enter exponent 5

The answer is 5153632

READY - RAM 1
>
```

= (equal)

Compares two expressions and returns “true” if they are equal.

expr1 = expr2

Where **expr1** and **expr2** are either both numeric expressions or both string expressions.

Discussion

Relational expressions involve the operators = (equal), <> (not equal), > (greater than), >= (greater than or equal), < (less than), and <= (less than or equal). These operators compare two numeric or string expressions and return a result of “true” or “false.” Numerically, a true result is 65,535 and a false result is 0.

The relational operators can work with either numeric expressions or string expressions. If a relational expression contains an argument of each type, BASIC issues a TYPE MISMATCH error.

BASIC considers strings to be equal or if they are exactly identical. BASIC compares strings by taking one character at a time from each string and comparing their ASCII codes. If the ASCII codes are different, then BASIC considers the lower to be less than the higher. If BASIC reaches the end of one string without differences, then it considers the shorter string to be less than the longer one. All characters and spaces count; BASIC does not ignore leading or trailing spaces.

Examples

```
>PRINT 6 = 4
 0

>A = 3 : IF A = 3 THEN PRINT "SAME"
SAME
```

```
10 INPUT "Enter first number",A
20 INPUT "Enter second number",B
30 IF A = B THEN PRINT "Numbers are equal"
40 IF A <> B THEN PRINT "Numbers are not
equal"

>RUN
Enter first number 22
Enter second number 22
Numbers are equal

READY - RAM 1
>
```

```
10 PRINT : INPUT "Enter first string: ",A$
20 INPUT "Enter second string: ",B$
30 IF A$ = B$ PRINT "Strings are identical"
40 IF A$ < B$ PRINT "First string is less than second"
50 IF A$ > B$ PRINT "First string is greater than
second"
60 GOTO 10

>RUN
Enter first string: ABC
Enter second string: ABC
Strings are identical

Enter first string: abc
Enter second string: ABC
First string is greater than second

Enter first string: ABC
Enter second string: ABCD
First string is less than second

READY – RAM 1
>
```

<> (not equal)

Compares two expressions and returns “true” if they are not equal.

expr1 <> expr2

Where **expr1** and **expr2** are either both numeric expressions or both string expressions.

Discussion For a complete description of relational operators, see the Discussion on page 12-7.

Examples

```
>PH0. 6 <> 6
0H

>PRINT 4^3 <> 64
 0
```

```
10 INPUT "Enter first number",A
20 INPUT "Enter second number",B
30 IF A = B THEN PRINT "Numbers are equal"
40 IF A <> B THEN PRINT "Numbers are not
equal"

>RUN
Enter first number 22
Enter second number 102

Numbers are not equal

READY - RAM 1
>
```

< (less than)

Returns “true” if the first expression is less than the second.

expr1 < expr2

Where **expr1** and **expr2** are either both numeric expressions or both string expressions.

Discussion For a complete description of relational operators, see the Discussion on page 12-7.

Examples

```
>PRINT 4 < 6
 65535

>PRINT 6 < 4
 0

>PRINT 4^3 < 64
 0
```

You can use the NOT statement in combination with relational operators to invert a result, as shown in the examples below:

```
10 INPUT "Enter first number",A
20 INPUT "Enter second number",B
25 PRINT
30 IF A < B THEN PRINT "Result is true"
40 IF NOT(A < B) THEN PRINT "Result is not true"

>RUN
Enter first number 22
Enter second number 22

Result is not true

READY - RAM 1
>
```

> (greater than)

Returns “true” if the first expression is greater than the second.

expr1 > expr2

Where **expr1** and **expr2** are either both numeric expressions or both string expressions.

Discussion

For a complete description of relational operators, see the Discussion on page 12-7.

Examples

```
>PRINT 4 > 6
 0

>PRINT 6 > 4
 65535

>PRINT 4^3 > 4^3
 0
```

You can use the NOT statement in combination with relational operators to invert a result, as shown in the examples below:

```
10 INPUT "Enter first number",A
20 INPUT "Enter second number",B
25 PRINT
30 IF A > B THEN PRINT "Result is true"
40 IF NOT(A > B) THEN PRINT "Result is not
true"

>RUN
Enter first number 45
Enter second number 22

Result is true

READY - RAM 1
>
```

<= (less than or equal to)

Returns “true” if the first expression is less than or equal to the second.

expr1 <= expr2

Where **expr1** and **expr2** are either both numeric expressions or both string expressions.

Discussion

For a complete description of relational operators, see the Discussion on page 12-7.

Examples

```
>PRINT "12" <= "012"
 0

>PRINT 6 <= 4
 0

>PRINT 4^3 <= 4^3
 65535
```

You can use the NOT statement in combination with relational operators to invert a result, as shown in the examples below:

```
10 INPUT "Enter first number",A
20 INPUT "Enter second number",B
25 PRINT
30 IF A <= B THEN PRINT "Result is true"
40 IF NOT(A <= B) THEN PRINT "Result is not
true"

>RUN
Enter first number 45
Enter second number 22

Result is not true

READY - RAM 1
>
```

>= (greater than or equal to)

Returns “true” if the first expression is greater than or equal to the second.

expr1 >= expr2

Where **expr1** and **expr2** are either both numeric expressions or both string expressions.

Discussion

For a complete description of relational operators, see the Discussion in Chapter 11.

Examples

```
>PRINT 4 >= 6
 0

>PRINT 6 >= 4
 65535

>PRINT 4^3 >= 4^3
 65535
```

You can use the NOT statement in combination with relational operators to invert a result, as shown in the examples below:

```
10 INPUT "Enter first number",A
20 INPUT "Enter second number",B
25 PRINT
30 IF A >= B THEN PRINT "Result is true"
40 IF NOT(A >= B) THEN PRINT "Result is not
true"

>RUN
Enter first number 45
Enter second number 22

Result is true

READY - RAM 1
>
```

Chapter 12

Logic

BASIC provides the logic functions associated with industrial control applications. In all, seven logic functions are available; these are:

Logic Function	BASIC Function
AND	AND
OR	OR
XOR (Exclusive OR)	XOR
INV(Invert)	INV
NAND	INV(AND)
NOR	INV(OR)
XNOR (Exclusive NOR)	INV(XOR)

Truth Tables

The results of logic functions to their inputs are summarized in the truth table below. For the following, the function is assumed to contain two inputs, however the truth table is identical for any number of inputs to the function.

Operation	Result of 0 : 0	Result of 0 : 1	Result of 1 : 0	Result of 1 : 1
AND	0	0	0	1
OR	0	1	1	1
XOR (Exclusive OR)	0	1	1	0
INV(Invert) (1 input only)	1			0
NAND	1	1	1	0
NOR	1	0	0	0
XNOR (Exclusive NOR)	1	0	0	1

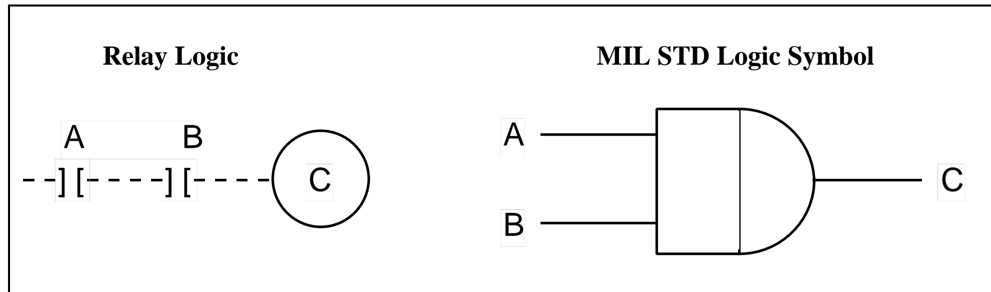
AND

Returns the logical AND.

iexpr1 AND iexpr2

Where **iexpr1** and **iexpr2** are any positive integer expressions. Although control logic results are 0 or 1, the AND operator does a "bitwise" AND on each bit of the integers. For example:

Bit Format		Decimal Format
	0010 1011	43
AND	1010 0101	165
	0010 0001	33



Truth Table

C = A AND B

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Examples

```
>PRINT 3 AND 2
  2
```

```
>PRINT 1 AND 0 AND 1 AND 0
  0
```

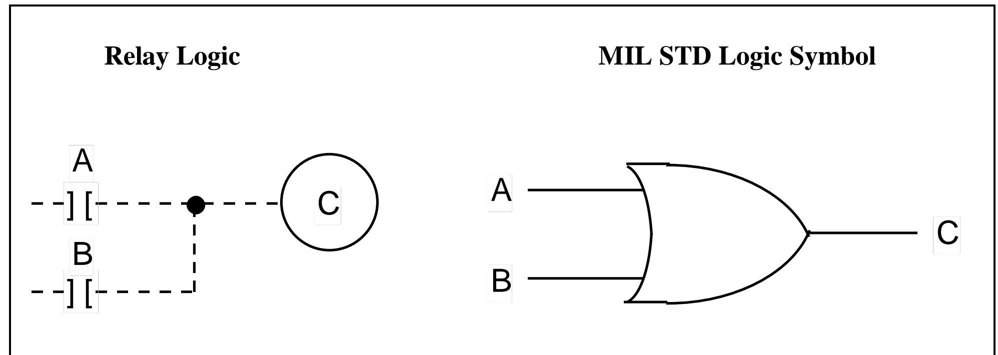
OR

Returns the logical OR of two expressions.

iexpr1 OR iexpr2

Where **iexpr1** and **iexpr2** are any positive integer expressions. Although control logic results are 0 or 1, the OR operator does a "bitwise" OR on each bit of the integers. For example:

	Bit Format	Decimal Format
	0010 1011	43
OR	1010 0101	165
	1010 1111	175



Truth Table

C = A OR B

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

Examples

```
>PRINT 4 OR 1
  5
```

```
>PRINT 1 OR 0 OR 1 OR 0
  1
```

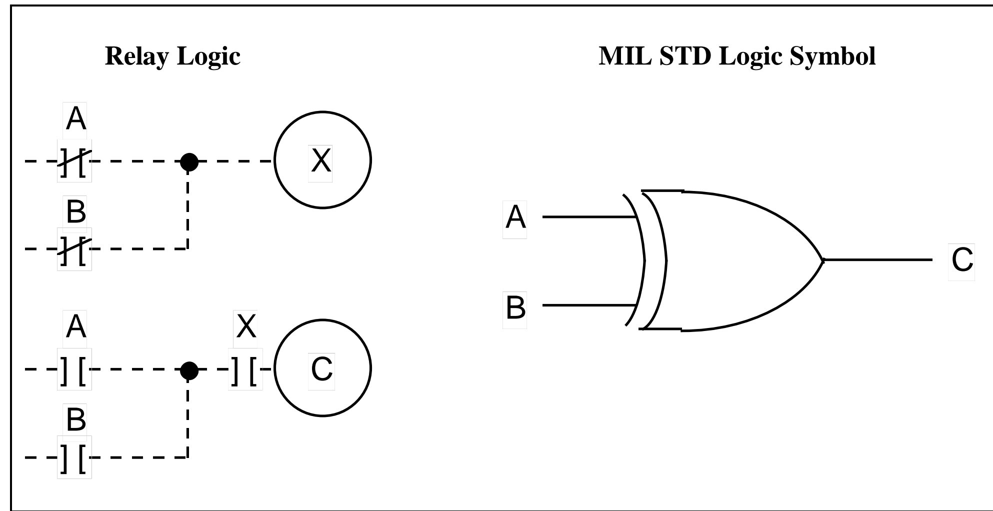
XOR

Returns the logical exclusive-OR.

iexpr1 XOR iexpr2

Where **iexpr1** and **iexpr2** are any positive integer expressions. Although control logic results are 0 or 1, the XOR operator does a "bitwise" XOR on each bit of the integers. For example:

Bit Format		Decimal Format
	0010 1011	43
XOR	1010 0101	165
	1000 1110	142



Truth Table

C = A XOR B

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Examples

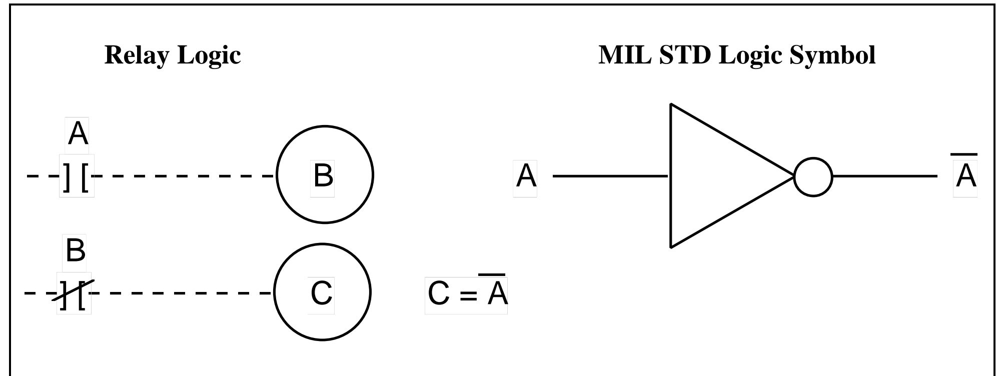
```
>PRINT 9 XOR 12
  5
```

INV

Returns 0 if the expression <> 0; returns 1 if the expression = 0.

```
INV(iexpr)
```

Where **iexpr** is any positive integer; if **iexpr** is not 0, then the INV operator returns 0; if **[expr]** is 0, then INV returns 1.



Truth Table

B = INV(A)

A	B
0	1
1	0

Examples

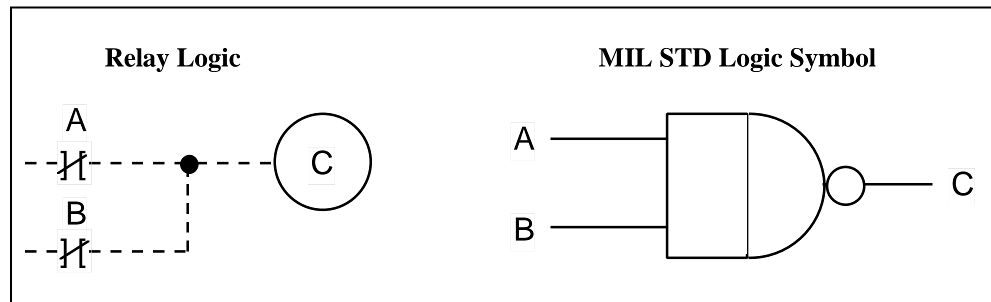
```
>PRINT INV(1)
 0
```

INV AND

Returns the logical NAND.

INV(iexpr1 AND iexpr2)

Where **iexpr1** and **iexpr2** are any positive integer expressions. INV returns 0 if the result of the AND is non-zero, and returns 1 if the result is 0.



Truth Table

C = INV(A AND B)

A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

Examples

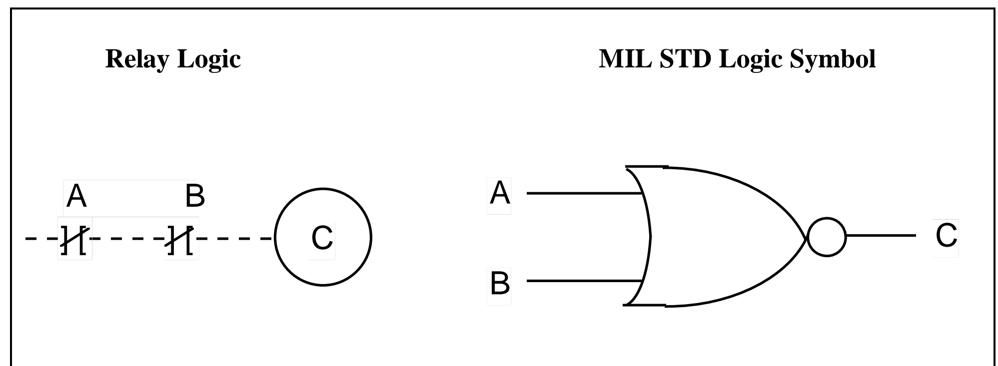
```
10 INPUT A : INPUT B
20 C = INV(A AND B) : REM NAND logic result
30 PRINT C

>RUN
?0
?1
1

READY - RAM 1
>
```

INV OR

Returns the logical NOR.

INV(iexpr1 OR iexpr2)

Where **iexpr1** and **iexpr2** are any positive integer expressions. The previous numeric examples don't really apply, since INV returns 0 if the result of the OR is non-zero, and INV returns 1 if the result of is 0.



Truth Table

C = INV(A OR B)

A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Examples

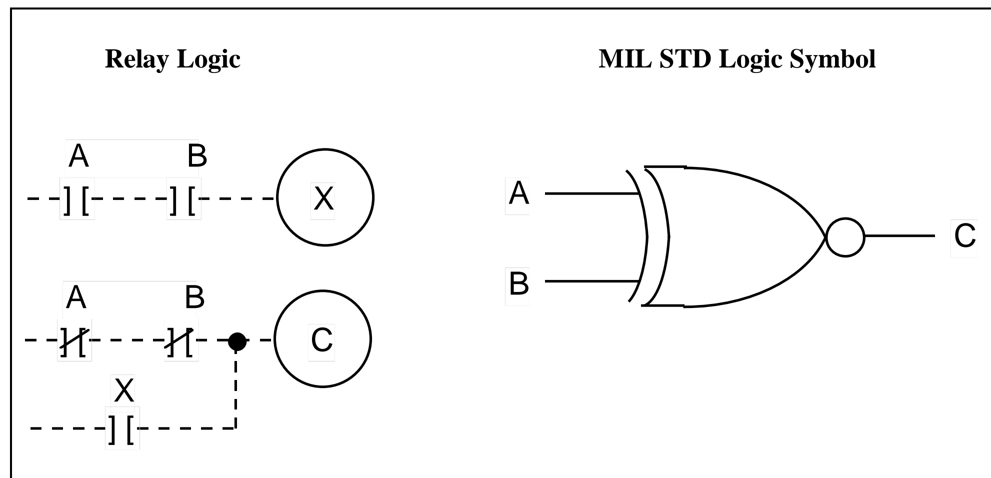
```
10 INPUT A : INPUT B
20 C = INV(A OR B) : REM NOR logic result
30 PRINT C

>RUN
?0
?1
0

READY - RAM 1
>
```

INV XOR

Returns the logical XNOR.

INV(iexpr1 XOR iexpr2)

Where **iexpr1** and **iexpr2** are any positive integer expressions. The previous numeric examples don't really apply, since INV returns 0 if the result of the XOR is non-zero, and INV returns 1 if the result of is 0.



Truth Table

C = INV(A XOR B)

A	B	C
0	0	1
0	1	0
1	0	0
1	1	1

Examples

```
10 INPUT A : INPUT B
20 C = INV(A XOR B) : REM XNOR logic result
30 PRINT C

>RUN
?17
?17
 1

READY - RAM 1
>
```

Chapter 13

CALLs

To make specific features of the Workstation easier to use, we've included some built-in subroutines that you can access through a CALL statement. If the number following the CALL is between 1 and 127, then the CALL refers to a built-in subroutine, not to a machine language subroutine you may have placed in memory yourself.

When a CALL requires that you provide data, your program must first PUSH the data on the stack; other CALLs return a result that you must POP after the CALL. The rest of this chapter describes each CALL and indicates when it requires input data or returns output data.

If the CALL does not require input or return a result, you need only the CALL alone, as shown in the format below, where **iexpr** is the number of the CALL.

CALL iexpr

If the CALL requires input, you must PUSH the data on the stack first, as shown in the following format, where **expr** must return a numeric result:

PUSH expr : CALL iexpr

If the CALL returns a result, you must POP the result into a numeric variable after the CALL, as shown in the following format:

CALL iexpr : POP var

CALL 12

Clears to the end of the display.

CALL 12

Discussion CALL 12 erases the display from the current cursor position to the bottom right corner of the screen.

```
>80 CALL 12 : REM Clear to end of display
```

CALL 13

Clears to the end of the line on the display.

CALL 13

Discussion CALL 13 erases the display from the current cursor position to the end of the current line.

```
>80 CALL 13 : REM Clear to end of line
```

CALL 30

Turn on COM1's RTS line.

CALL 30/40

Discussion CALL 30 turns on the RTS line on COM1. In order for this to work properly, you must open COM1 with the RN parameter. CALL 33 turns off RTS.

```
>10 CALL 30 : REM Turn on COM1 RTS
```

CALL 33

Turn off COM1's RTS line.

CALL 33

Discussion CALL 33 turns off the RTS line on COM1. In order for this to work properly, you must open COM1 with the RN parameter. CALL 30 turns on RTS.

```
>10 CALL 33 : REM Turn off COM1 RTS
```

CALL 38

Enters the on-line configuration menu.

CALL 38

Discussion CALL 38 enters the on-line configuration menu, which allows you to configure many of the Workstation's operating parameters.

When you exit from the on-line configuration menu, the Workstation "warmstarts" BASIC, which means that it clears all variables. Although a running program can contain CALL 38, exiting from the on-line configuration menu cannot return you to the program.

However, if you set up your Workstation with REACT R, then the unit automatically runs the program after returning from the on-line configuration menu.

CALL 39

Enters the monitor.

CALL 39

Discussion CALL 39 enters the Workstation's internal "monitor" program. When you enter the monitor, you see the following screen:

```
Debug Monitor(V2.00)
Ready>
```

When the monitor is running, you can download new firmware. You can also type some special commands, including INIT, which performs a first-time initialization of BASIC and erases the current program from memory. The other monitor commands are not useful.

CALL 40 and 41

Returns the number of characters in COM1 buffer.

CALL 40 : POP var

Discussion

These CALLs return the number of characters in a port's receive or transmit communications buffer. For example, if the result of CALL 40 is 2, that means that there are two characters waiting in port #1's receive buffer; your program can fetch these characters with an INPUT or INPUT$ statement.

As another example, if the result of CALL 41 is 15, that means that there are 15 characters in port #1's transmit buffer remaining to be sent; your program can watch this value to make sure characters are going out; if the value doesn't change, then transmitting has halted for some reason.

CALL	Function
40	Port #1 receive
41	Port #1 transmit

```
10 CALL 40 : POP A : Return number of characters in port#1
20 PRINT A
```

CALL 82

Prints a list of all variables used.

```
CALL 82
```

Discussion CALL 82 prints a list of all the variables created by your program. You must execute this CALL after you run your program. The list is in the sequence in which the variables are first used by your program. For each item in the list that is an array, string, or string array, the list includes the number of elements in the array and/or the maximum length of the string.

```
>CALL 82

A$(127)
B$(27)
K$(27,20)
K(27)
I
I%
I%(10)
M
MH
X
P
ER
```

Appendix A

Speed-up Hints

This section offers some programming techniques you can use to squeeze the maximum speed out of your BASIC program.

Subroutines

Whenever BASIC executes an instruction that goes to a line number (GOTO, GOSUB, ON ERROR and ON TIME), it starts looking for that line number from the beginning of the program.

If you have subroutines that your program calls frequently, then you should place them as close as possible to the beginning of your program. For example, many experienced programmers start their programs with the instruction "GOTO 1000" and use line numbers 10 through 999 for subroutines.

BASIC uses about 40 microseconds to scan through a line while performing a line search. For example, if BASIC finds a subroutine at the 10th line instead of the 110th line, it takes 4 milliseconds less time to find it.

Variables

Each time your program creates a variable, BASIC assigns it to the next position in the list. And every time your program refers to a variable, BASIC looks through the list starting with the first variable created.

To gain maximum speed in the use of variables, your program should create the most-used variables first. For example, if your program frequently refers to X and Y, then your program could simply contain the following line to create those variables: Y = 0 : X = 0. Because you listed Y first, BASIC finds it first.

BASIC uses 12 microseconds to scan through each variable when it searches for a variable.

Constants

When BASIC encounters a constant, such as 7 or 3.2728, it must convert that number into its internal format. Because this takes more time than simply looking up a variable, you should convert frequently-used constants to variables instead.

For example, you might want to initialize an array of 100 numbers to the value 55. Although each of the following routines accomplishes this, the second routine is 25 milliseconds faster than the first:

```
10 REM This uses a constant.
20 DIM D(100)
40 FOR I = 1 TO 100 : D(I) = 55 : NEXT


10 REM This uses a variable instead of a constant.
15 REM (And this is 25 milliseconds faster!)
20 DIM D(100)
30 X = 55
40 FOR I = 1 TO 100 : D(I) = X : NEXT
```

Integer vs. Floating Point

BASIC performs 4-function arithmetic about twice as fast when using integer variables (indicated with the % symbol) instead of floating point.

For floating-point operations such as trigonometric functions, your program runs faster if you use floating-point variables. When you use integer variables in floating-point operations, BASIC converts the integer to floating point before processing.

FOR . . . NEXT Loops

The best way to speed up a FOR-NEXT loop is to omit the optional variable following NEXT. For example, although the following two lines are equivalent, the second executes about 10% faster than the first:

```
10 FOR I = 1 TO 1000 : NEXT I
20 FOR I = 1 TO 1000 : NEXT : REM This is about 10% faster.
```

Appendix B

BASIC Differences

Differences between Series 30/40 BASIC and GW-BASIC

This section describes the main differences between the BASIC in the Series 30/40 Workstation and the GW-BASIC in IBM-compatible PCs.

Hardware-specific functions

Because the Series 30/40 Workstation is fundamentally different from the IBM PC, it lacks GW-BASIC's graphics, sound, disk-handling, microprocessor-specific functions, DOS-specific functions, and some keyboard capabilities.

Because the Workstation's microprocessor has four distinct memory types instead of the IBM PC's single type of memory, Series 30/40 BASIC uses XBY, DBY, CBY, and IBY commands instead of GW-BASIC's PEEK and POKE commands.

Variable labels, arrays, functions

Because the Workstation supports a much smaller memory, its labels have only two significant characters while GW-BASIC supports up to 40. Also, Series 30/40 BASIC supports only single-dimension arrays, while GW-BASIC supports up to 255 dimensions. Finally, Series 30/40 BASIC does not support the function (FN) capability of GW-BASIC.

Strings

Series 30/40 BASIC requires you to specify the maximum length of a string (up to 254 characters) before you use it; otherwise, its default maximum length is 10. In GW-BASIC, all strings may vary in size up to 255 characters, but when the string space is full, GW-BASIC pauses to perform “garbage collection.” Thanks to the Workstation's pre-defined string lengths, it never has to take time to collect garbage.

Numbers

The numeric variable types available in Series 30/40 BASIC are fewer in number and different from GW-BASIC's number types. The Workstation's floating point numbers have 8 digits of precision and a range of 10^{-99} to 10^{99}, while GW-BASIC's floating point numbers have 7 digits (single-precision) or 17 digits (double-precision) of precision and a range of 10^{-38} to 10^{38}. Series 30/40 BASIC supports 17-bit integers that range from -65,535 to +65,535, while GW-BASIC's integers are only 16 bits that range from -32,767 to +32,767.

PRINT USING

The Workstation's USING statement offers only a subset of the capabilities of GW-BASIC's USING statement. However, because virtually all of the unsupported features are handy only for accounting-oriented systems, you probably won't feel the loss.

Other differences

Series 30/40 BASIC does not support octal representation of numbers, but its support for hexadecimal numbers significantly exceeds that of GW-BASIC. Series 30/40 BASIC's implementation of the OPEN command is significantly different, and its storage format for variables means that the VARPTR function yields significantly different results.

Appendix C

Memory Map

This section shows how BASIC uses the Workstation's memory. At the end of this section is an illustration of BASIC's use of the microprocessor's memory. In order to understand fully how this memory map works, you should be familiar with the internal and external memory capabilities of the 8032 microprocessor and have data sheets for the microprocessor and Optrex DMC20261NY-LY-NM LCD display.

Note This memory map is provided for your reference only and is *not* guaranteed to remain consistent with future firmware revisions.

Internal RAM

DBY Address	Function
00h to 07h	Microprocessor registers
08h	BASIC text pointer (low byte).
09h	Argument stack pointer (low byte; high byte is a constant).
0Ah	BASIC text pointer (high byte).
0Bh to 0Fh	Scratch pad registers.
10h to 17h	Microprocessor registers used by XMIT service.
18h to 23h	Microprocessor registers used by interrupt service routines.
20h.1	When low, disables all system timer functions except resetting the watchdog (only during initialization).
20h.2	Indicates that the Flash EPROM contains valid firmware.
20h.3	Toggles every 2.5 milliseconds if the vector at 17Ah is initialized.
20h.4	Unused; reserved for future use.
20h.5	Indicates the system timer is on a one-second tick.
20h.6 to 20h.8	Unused; reserved for future use.
21h.1	Indicates that COM1 has not finished transmitting.
21h.2	Unused; reserved for future use.
21h.3, 21h.4	Current state of the COM1 receiver; 00 = ready; 10 = halted due to handshaking.
21h.5	Indicates that COM1 needs to transmit an XON.
21h.6	Indicates that COM1 needs to transmit an XOFF.
21h.7	Scratch pad bit.
22h.1 to 22h.6	Unused; reserved for future use.
22h.7	Indicates to various internal routines that the system is inserting a flashing character into the display buffer.
22h.8	Indicates to the internal line-finding routine that the caller wants to point to the end of the current program.
23h.1 to 23h.6	Unused; reserved for future use.
23h.7	Indicates when the display is displaying flashing characters as spaces.
23h.8	Indicates that the display routine should only write the character to the display buffer without displaying it.

DBY Address	Function
24h.1	Indicates that a programming console is connected and active.
24h.2	Holds status of mode (command/run) bit for error-handler (in case error-handler doesn't conclude with a RESUME).
24h.3	Disables echoing of input for INPUT statement.
24h.4	Enables printing LF after CR.
24h.5	Set if an ON ERROR handler is currently executing.
24h.6 to 24h.7	Unused; reserved for future use.
24h.8	Indicates that the STOP statement was executed.
25h.1	Indicates that ON TIME interrupts are enabled.
25h.2	Indicates that an interrupt is in progress.
25h.3	Reserved for future use.
25h.4	Indicates that an ON ERROR statement was executed.
25h.5	Indicates that an ON TIME interrupt is in progress.
25h.6	Indicates ON TIME statement was executed.
25h.7	Unused; reserved for future use.
25h.8	Indicates that the CONT command will work.
26h.1	Indicates that no program changes have occurred (i.e., if reset, indicates the user changed the program, which causes BASIC to clear all variables).
26h.2	Indicates that BASIC is in the middle of executing an SDIM statement.
26h.3	Indicates that the internal error-processing routine is busy (in order to inhibit asynchronous communications errors during error processing).
26h.4	Scratch pad bit used by floating-point math routines.
26h.5	Indicates to the expression evaluator that the argument stack has at least one value.
26h.6	Set when RETI executed; cleared during its processing.
26h.7	Indicates to the expression evaluator that an integer argument is on the stack (only if the next bit is zero).
26h.8	Indicates to the expression evaluator that a string argument is on the stack.
27h.1	Forces PRINT statement to use the display regardless of the communication port configuration; used internally for displaying messages on the screen (e.g., error messages).
27h.2	Enables program tracing (set by TRON, reset by TROFF).
27h.3	Indicates that a STR$ statement is in progress.
27h.4	Indicates that the Workstation is in the Command (or direct) mode; when reset, indicates that a program is running.
27h.5	Unused; reserved for future use.
27h.6	Indicates to an internal routine that it is printing a message stored in firmware.
27h.7	Indicates that PRINT should not print leading zeroes for a hex print (i.e., PH0. executed).
27h.8	Indicates that PH0. or PH1. was executed and that PRINT should print in hex mode.
28h to 29h	Reserved.
2Ah to 3Dh	Floating point temporaries.
3Eh	Unused; reserved for future use.
3Fh	Scratch byte

DBY Address	Function
40h	Milliseconds counter (5 msec resolution); free-running.
41h	Milliseconds counter (5 msec resolution); corresponds to fractional value of BASIC's TIME variable.
42h	Scan counter (5 msec/scan) for reading keyboard every 50 milliseconds; essentially a down counter that starts at 10 and decrements every 5 milliseconds.
43h	Scan counter (5 msec/scan) for duration of keyboard beep.
44h, 45h	Unused; reserved for future use.
46h.1	Keyboard buffer overflow error.
46h.2	Keyboard fault (invalid combination of two or more keys detected).
46h.3 to 46h.5	Unused; reserved for future use.
46h.6	[F1]-[↵] combination detected.
46h.7	Indicates that the input timer for keyboard has timed out.
46h.8	At least one character is available in the keyboard buffer.
47h	Run-time copy of selected configuration parameters for COM1:
47h.1	XON/XOFF handshaking enabled on COM1 transmit.
47h.2	XON/XOFF handshaking enabled on COM1 receive.
47h.3	Parity and parity substitution enabled on COM1.
47h.4	Even parity selected on COM1.
47h.5	Buffered communications enabled on COM1.
47h.6	RTS for receive handshake enabled on COM1.
47h.7	[Ctrl]-C interrupts enabled on COM1.
47h.8	7-bit data size selected on COM1.
48h.1	COM1 receive buffer overflowed.
48h.2	Always 0.
48h.3	COM1 received parity error.
48h.4	Always 0.
48h.5	Always 0.
48h.6	COM1 received a [Ctrl]-C.
48h.7	COM1 input timer timed out.
48h.8	COM1 receive buffer not empty.
49h.1	COM1 CTS input tested and not asserted.
49h.2	XOFF received on COM1.
49h.3 to 49h.6	Reserved for future use.
49h.7	COM1 transmit buffer full.
49h.8	COM1 transmit handshaking timeout was already reported.
4Ah to 50h	Unused; reserved for future use.
51h	Fractional part of ON TIME setpoint (equals fractional part * 5).
52h, 53h	Pointer to program is saved here in case CONT executed (high byte, low byte).
54h	Internal pointer for string expression evaluation.
55h, 56h	Temporaries for transcendental functions.
57h, 58h	Whole portion of ON TIME setpoint (high byte, low byte).
	READ text pointer (low byte)
	5Ah Control stack pointer (low byte; high byte is a constant)
5Bh	READ text pointer (high byte)

DBY Address	Function
5Ch	Local/global printer port; if the high bit is set, the low nibble is a local printer port selection and the high nibble is the global printer port selection; the port number varies between 0 and 4.
5Dh	PRINT USING format information.
5Eh	Local/global input port; if the high bit is set, the low nibble is a local input port selection and the high nibble is the global input port selection; the port number varies between 0 and 4.
5Fh to 60h	Start of current program (high byte, low byte).
61h to FFh	Microprocessor stack.

Special Function Registers

DBY Address	Function
80h	Port 0.
81h	Stack pointer.
82h, 83h	Data Pointer (low byte, high byte).
87h	Power control; bits 1-7 are reserved for future use; bit 8 doubles baud rate when Timer 1 is the baud rate generator.
88h	TCON; Timer/Counter Control Register.
88h.1	Interrupt 0 trigger: 1 selects falling edge, 0 selects low level (default = 0).
88h.2	Interrupt 0 edge flag; set when interrupt detected, cleared when serviced.
88h.3	Interrupt 1 trigger: 1 selects falling edge, 0 selects low level (default = 0).
88h.4	Interrupt 1 edge flag; set when interrupt detected, cleared when serviced.
88h.5	Timer 0 enable: 1 selects run, 0 selects stop (default = 0).
88h.6	Timer 0 overflow flag; set on overflow, cleared when serviced.
88h.7	Timer 1 enable: 1 selects run, 0 selects stop (default = 1).
88h.8	Timer 1 overflow flag; set on overflow, cleared when serviced.
89h	TMOD; Timer/Counter Mode Control Register.
89h.1, 89h.2	If 00, TL0 is a 5-bit prescaler (13-bit 8048 mode); if 01, TL0 and TH0 are cascaded; if 10, TL0 is an 8-bit auto-reload timer and TH0 is its value; if 11, TL0 is an 8-bit timer controlled by Timer 0 bits while TH0 is an 8-bit timer controlled by Timer 1 control bits. The default setting is 01, which makes Timer 0 a 16-bit timer; Timer 0 serves the TONE command.
89h.3	Selects counter mode if set, timer mode if reset (default = reset).
89h.4	If reset, Timer 0 is controlled by bit 88h.5; if set, Timer 0 is controlled by a combination of bit 88h.5 and the active condition of the interrupt 0 input (pin INT0) (default = reset).
89h.5, 89h.6	If 00, TL1 is a 5-bit prescaler (13-bit 8048 mode); if 01, TL1 and TH1 are cascaded; if 10, TL1 is an 8-bit auto-reload timer and TH1 is its value; if 11, Timer 1 is stopped. The default setting is 00, which makes Timer 1 a 13-bit timer; Timer 1 is the baud rate generator for COM1.
89h.7	Selects counter mode if set, timer mode if reset (default = reset).
89h.8	If reset, Timer 1 is controlled by bit 88h.7; if set, Timer 1 is controlled by a combination of bit 88h.7 and the active condition of the interrupt 1 input (pin INT1) (default = reset).
8Ah, 8Ch	Timer 0 setpoint (low byte, high byte).
8Bh, 8Dh	Timer 1 setpoint (low byte, high byte); this varies depending on the baud rate of COM1.

DBY Address	Function
90h	PORT 1.
90h.1	Selects low half of 128K Flash EPROM (active low); when this bit is high, the upper half of a 128K Flash EPROM is selected. 128K Flash EPROM is available as an option.
90h.2	Selects lower 32K of currently selected half of Flash EPROM; this is useful only when programming the Flash.
90h.3	Disable Flash EPROM programming (active low); when this bit is low, the 32K block of data memory from 0 to 7FFFh is re-mapped to the same addresses in code memory, while one of the four 32K blocks of code memory is re-mapped to data memory starting at 8000h; the code block is selected with bits 90h.1 and 90h.2.
90h.4	COM1 RTS output (active low).
90h.5	Keyboard column select.
90h.6	Keyboard column select.
90h.7	Keyboard column select.
90h.8	Keyboard column select.
98h	SCON; Serial Port Control Register.
98h.1	COM1's receive interrupt flag; set by hardware when a character is received and cleared by software when the interrupt is serviced.
98h.2	COM1's transmit interrupt flag; set by hardware when a character is sent and cleared by software when the interrupt is serviced.
98h.3	Holds the 9th bit received in mode 2 or 3 (bits 98h.7 and 98h.8 set to 10 or 11) (default = unused); only used if configured for 8,N,2; 8,O; 8,E; 7,O,2; or 7,E,2.
98h.4	Holds the 9th bit to be transmitted in mode 2 or 3 (bits 98h.7 and 98h.8 set to 10 or 11) (default = unused); only used if configured for 8,N,2; 8,O; 8,E; 7,O,2; or 7,E,2.
98h.5	Receive enable (default = set).
98h.6	Enables multiprocessor communications (default = reset).
98h.7, 98h.8	If 00, serial port operates as a shift register at the one-twelfth of the oscillator's frequency (11.0592/12 MHz); if 01, operates as an 8-bit serial port with a variable frequency using Timer 1 or Timer 2; if 10, operates as a 9-bit serial port at 1/32 or 1/64 of the oscillator's frequency; if 11, operates as a 9-bit serial port using Timer 1 or Timer 2. The default is 01, which means that the COM1 port operates as a simple 8-bit serial port; Timer 1 generates the baud rate for the COM1 port.
99h	Serial data buffer; received data is read from this byte and transmitted data is written to this byte.
A0h	PORT 2.
A8h	IE; Interrupt Enable Register.
A8h.1	Enables external interrupt 0 (default = disabled); because COM1_CTS is tied to the input for external interrupt 0, interrupts are enabled only when CS handshaking is enabled.
A8h.2	Enables Timer 0 interrupt (default = disabled; enabled when TONE or BEEP statement is in progress).
A8h.3	Enables external interrupt 1 (default = enabled; this interrupt comes from the VLSI 16C452 communications chip that handles COM2, COM3, and LPT1).
A8h.4	Enables Timer 1 interrupt (default = disabled because Timer 1 is the baud rate generator for the COM1 serial port).

DBY Address	Function
A8h.5	Enables the COM1 serial port interrupt (default = enabled).
A8h.6	Enables Timer 2 interrupt (default = enabled).
A8h.7	Reserved.
A8h.8	Master Interrupt Enable (default = enabled).
B0h	PORT 3.
B0h.1	COM1's receive line.
B0h.2	COM1's transmit line.
B0h.3	Interrupt 0 input; COM1's CTS input.
B0h.4	Keyboard row read.
B0h.5	Keyboard row read.
B0h.6	Keyboard row read.
B0h.7	Microprocessor's write line to external memory.
B0h.8	Microprocessor's read line to external memory.
B8h	IP; Interrupt Priority Register.
B8h.1	High interrupt priority select for external interrupt 0 (default = low), which is COM1's CTS input.
B8h.2	High interrupt priority select for Timer 0 interrupt (default = high), which is used when beeping the horn.
B8h.3	High interrupt priority select for external interrupt 1 (default = low), which is used for COM2, COM3, and LPT1.
B8h.4	High interrupt priority select for Timer 1 interrupt (default = low), which is used as the COM1 baud rate generator.
B8h.5	High interrupt priority select for COM1 serial port interrupt (default = low).
B8h.6	High interrupt priority select for Timer 2 interrupt (default = low), which is used as the system timer.
B8h.7	Reserved for future use.
B8h.8	Reserved for future use.
C8h	T2CON; Timer/Counter 2 Control Register.
C8h.1	When set, enables captures on negative transitions at microprocessor pin T2EX; when reset, auto-reloads occur either with Timer 2 overflows or negative transitions at T2EX when bit C8h.4 is set. When either bit C8h.5 or C8h.6 is on, the setting of bit C8h.1 doesn't matter and the timer does an auto-reload on overflow. The default is reset in order to cause an auto-reload; this helps the background interrupts occur at 5-millisecond periods with no accumulation of error (except that caused by the oscillator's frequency).
C8h.2	Enables Timer 2 to operate as an external event counter; if reset, Timer 2 operates as a timer (default = reset).
C8h.3	Enables Timer 2 to run (default = enabled).
C8h.4	Enables events at T2EX pin (if unused for serial port) (default = disabled).
C8h.5	Enables using Timer 2 for the COM1 port's baud rate generator when transmitting (default = disabled).
C8h.6	Enables using Timer 2 for the COM1 port's baud rate generator when receiving (default = disabled).
C8h.7	If bit C8h.4 is high, then bit C8h.7 is set by hardware when a capture or reload occurs on a negative transition of T2EX; otherwise, if this bit is set the microprocessor vectors to the Timer 2 interrupt service routine (default = reset).

DBY Address	Function
C8h.8	Timer 2 overflow flag; set by hardware when Timer 2 overflows, provided bits C8h.5 and C8h.6 are reset.
CAh, CBh	Timer 2 setpoint (default = EE00h to generate an interrupt every 5 milliseconds) (low byte, high byte).
CCh, CDh	Timer 2 current value (low byte, high byte).
D0h	Microprocessor's Processor Status Word (PSW):
D0h.1	Microprocessor's parity flag.
D0h.2	Reserved for future use.
D0h.3	Microprocessor's overflow flag.
D0h.4, D0h.5	Register bank select bits; 00 = low bank, 01 = 2nd bank, 10 = 3rd bank, 11 = 4th bank.
D0h.6	Scratch pad flag.
D0h.7	Microprocessor's auxiliary carry flag.
D0h.8	Microprocessor's carry flag.
E0h	Microprocessor's accumulator.
F0h	Microprocessor's "B" register.

External RAM

XBY Address	Function
00h to 04h	Firmware signature; BASIC's signature is 15h 32h 71h 24h A5h; if the signature doesn't match, BASIC clears the REACT status, sets the console to port 1, and resets all port configuration parameters to their defaults.
05h	Status of LED control register (because the hardware register is write-only).
06h, 07h	RAM size; 0 = faulty; 7FFFh = 32K; F7FFh = 64K.
08h	Flash EPROM status; 0 = faulty; 80h = 64K; 81h = 128K.
09h	Unused; reserved for future use.
0Ah	Unused; reserved for future use.
0Bh	Display status; 0 = faulty; 1 = okay.
0Ch	Always 0.
0Dh	Always 0.
0Eh	Always 0.
0Fh	Always 0.
10h	Always 0.
11h	RAM signature, which should be 82h 59h 66h 24h; if the signature doesn't match, BIOS performs destructive tests on RAM, resets all port configuration parameters to their defaults; and initializes all hardware.
15h,16h	Unused; reserved for future use.
17h to 19h	Hours, minutes, and seconds for the time of day; these bytes are BCD, not binary!
1Ah to 1Ch	Day, month, and year for the date; these bytes are BCD, not binary!
1Dh	ASCII code for the date separation character; for example, BASIC sets up this character as "/", which is an ASCII code of 2Fh.
1Eh	Date is input and displayed in the international format if this byte is not zero.
1Fh	Alias for COM5, the "memory" port (not accessible from BASIC).
20h, 21h	Address pointer for the memory port; points to the next available byte where the transmit routine will send the byte.
22h	Alias for the console port.

XBY Address	Function
23h	Control byte for the keyboard:
23h.1	Unused; reserved for future use.
23h.2	Auto-repeat enable.
23h.3 to 23h.4	Unused; reserved for future use.
23h.5	Buffer enable.
23h.6	[F1]-[↵] enable.
23h.7 to 23h.8	Unused; reserved for future use.
24h	Input timer setpoint.
25h	Duration of keypress beep (50 msec resolution).
26h, 27h	Reload value for beep timer; BASIC writes the number stored here into the bytes at DBY(8Ah) and DBY(8Ch), which is the Timer 0 register.
28h	Time delay until the first key repeat (50 msec resolution).
29h	Time delay between repeats (50 msec resolution).
2Ah	High byte of the page in external memory used for re-mapping the keyboard's ASCII codes. Currently, no configuration bit exists to enable this capability.
2Bh	Control byte for the display:
2Bh.1	Unused; reserved for future use.
2Bh.2	IBM PC-compatible map enable.
2Bh.3	Unused; reserved for future use.
2Bh.4	Enable for automatic carriage return at end of line.
2Bh.5	Enable for automatic carriage return/line feed at end of line.
2Bh.6	Scroll enable.
2Bh.7	Wraparound enable.
2Bh.8	Display re-map enable.
2Ch to 2Eh	Unused; reserved for future use.
2Fh	Cursor type; bits 1 and 2 select the type: 00 = undefined, 01 = solid block; 10 = underline; 11 = undefined; bits 3 through 7 have no function; bit 8 is the cursor enable.
30/40h	Unused; reserved for future use.
31h	Unused; reserved for future use.
32h	High byte of the page in external memory used for re-mapping the display's ASCII codes before sending them to the display.
33h	COM1 alias.
34h	ASCII code for the character used to replaced those with parity errors received on COM1.
35h	COM1 configuration parameters; byte 1:
35h.1, 35h.2	COM1 data size: 00 = undefined; 01 = 7 bits; 10 = undefined; 11 = 8 bits.
35h.3	COM1 stop bits; 0 = 1 stop bit; 1 = 2 stop bits.
35h.4, 35h.5	COM1 parity; 00 = none; 01 = odd; 11 = even.
35h.6 to 35h.8	COM1 baud rate; 000 = 110; 001 = 30/400; 010 = 600; 011 = 1200; 100 = 2400; 101 = 4800; 110 = 9600; 111 = 19200.
36h	COM1 configuration parameters; byte 2:
36h.1	Parity translate enable.
36h.2	Buffer enable.
36h.3	[Ctrl]-C enable.
36h.4, 36h.5	Reserved for future use as DTR handshaking line control; always 0.

XBY Address	Function
36h.6	CTS handshake enable.
36h.7	Reserved for future use as CD handshake line enable; always 0.
36h.8	Reserved for future use.
37h	COM1 configuration parameters; byte 3:
37h.1	XON/XOFF handshaking enabled on transmit.
37h.2	XON/XOFF handshaking enabled on receive.
37h.3	RTS always on.
37h.4	RTS on at the start of transmitting.
37h.5	RTS on only during transmitting.
37h.6	RTS receive handshake enable.
37h.7	Reserved for future use.
37h.8	Reserved for future use.
38h	Setpoint for the timer for CTS or XON/XOFF handshaking on COM1 transmit.
39h	Setpoint for the timer for XON/XOFF handshaking on COM1 receive.
3Ah to 4Ah	Unused; reserved for future use.
4Bh	Station number for monitor protocol.
4Ch to 51h	Time and date of last power-up.
52h, 53h	Total quantity of power-ups since the last firmware download.
54h to 63h	Unused; reserved for future use.
64h to 69h	Time and date of last download.
6Ah, 6Bh	Total quantity of firmware downloads since the beginning of time.
6Ch to 71h	Time and date of last reset of the remaining diagnostic records.
72h, 73h	Total quantity of diagnostic resets.
74h to 79h	Time and date of last receive overflow error on COM1.
7Ah, 7Bh	Total quantity of receive overflow errors on COM1.
7Ch to 81h	Time and date of last receive overrun error on COM1.
82h, 83h	Total quantity of receive overrun errors on COM1.
84h to 89h	Time and date of last parity error on COM1.
8Ah, 8Bh	Total quantity of parity errors on COM1.
8Ch to 91h	Time and date of last framing error on COM1.
92h, 93h	Total quantity of framing errors on COM1.
94h to D3h	Unused; reserved for future use.
D4h to D9h	Time and date of last keyboard buffer overflow error.
DAh, DBh	Total quantity of keyboard buffer overflow errors.
DCh	Reset source:
DCh.1	Power up.
DCh.2	Reserved; always 0.
DCh.3	Reserved; always 0.
DCh.4	Software.
DCh.5	First time.
DCh.6 to DCh.8	Undefined.
DDh to E4h	Reserved.
E5h	If set to 47h, enables internal flash access routines to reference routines downloaded to RAM instead.
E6h, E7h	Address of RAM-based flash write routine.
E8h, E9h	Address of RAM-based flash erase routine.

XBY Address **Function**

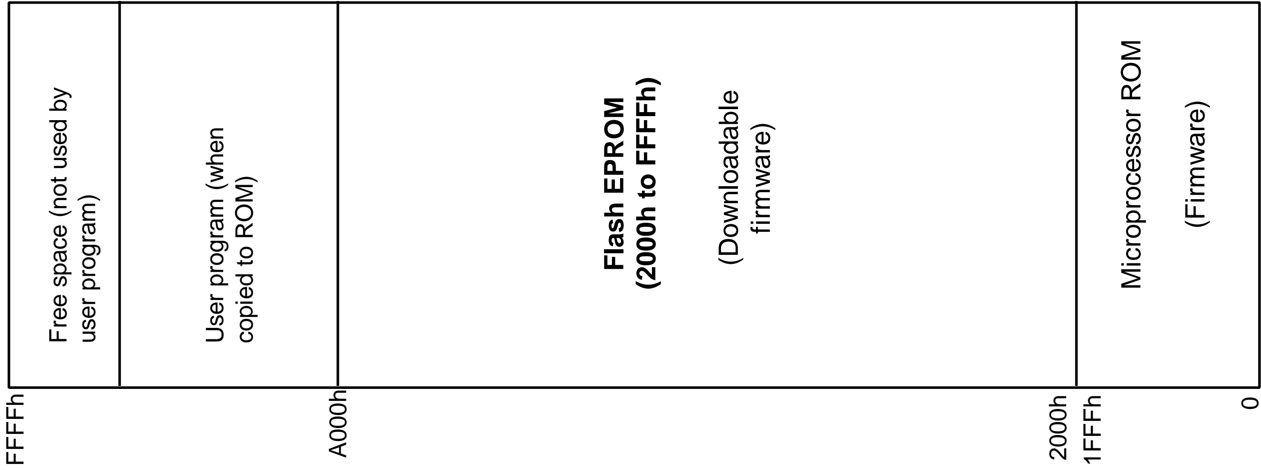
EAh, EBh	Address of RAM-based flash test routine.
ECh	Internal timer to service display (5 msec resolution).
EDh	Actual value of keyboard auto-repeat timer.
EEh	"Raw" code of the key held down on the previous scan of the keyboard; FFh indicates no key pressed; used for key debouncing.
EFh	"Raw" code of the key held down on this scan; FFh indicates no key pressed.
F0h	"Raw" code of the last keypress recognized; used for auto-repeat sensing.
F1h	Unused; reserved for future use.
F2h	Keyboard "head" pointer; increments when a character is removed from the keyboard buffer.
F3h	Keyboard "tail" pointer; increments when a key is inserted to the keyboard buffer.
F4h	Current row position of cursor (0 to 3).
F5h	Current column position of cursor (0 to 19).
F6h to F7h	Unused; reserved for future use.
F8h	COM1 receive "head" pointer; increments when the system removes a character from the receive buffer.
F9h	COM1 receive "tail" pointer; increments when the service routine receives a character from the port and adds it to the receive buffer.
FAh	COM1 transmit "head" pointer; increments when the service routine removes a character from the buffer and sends it out the port.
FBh	COM1 transmit "tail" pointer; increments when the system adds a character to the transmit buffer.
FCh to FFh	Unused; reserved for future use.
100h to 127h	Display buffer; holds an image of the characters currently displayed.
128h to 155h	Unused; reserved for future use.
156h	Actual value of monitor's timer for user entries (1 second resolution).
157h	Actual value of monitor's timer for character times (50 msec resolution).
158h	Actual value of keyboard input timer.
159h	Actual value of COM1 transmit handshake timer.
15Ah	Actual value of COM1 receive handshake timer.
15Bh to 15Fh	Unused; reserved for future use.
160h to 171h	Reserved for internal use.
172h to 177h	Map of actual hardware port numbers for each alias.
178h to 179h	Unused; reserved for future use.
17Ah to 17Dh	Vector for firmware routine to be driven by the system timer service routine every five milliseconds. The first byte is 28h; the second bytes are the high and low byte of the vector; the last byte is an exclusive OR checksum of the vector's two bytes.
17Eh to 17Fh	16-bit seconds counter (high byte, low byte).
180h to 1CFh	Flash enables for each character displayed; 0 = no flash; 80h = flash.
1D0h	Workstation's model number; lower two digits stored in BCD format (30/40h = normal Workstation; 31h = 128K Flash EPROM).
1D1h to 1FFh	Reserved for future use; contact Nematron if you want to reserve any of this area for your own application.

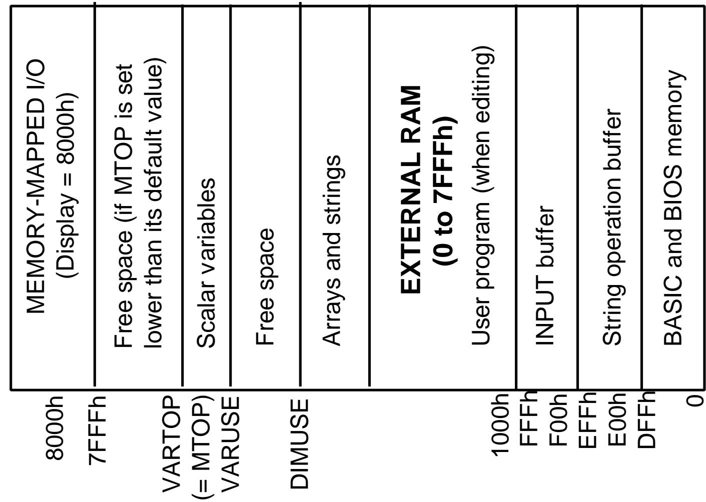
XBY Address	Function
200h to 2FFh	COM1 receive buffer.
30/400h to 3FFh	COM1 transmit buffer.
400h to 8FFh	Unused; reserved for future use.
900h to 9FFh	Keyboard buffer.
A00h to BFFh	Reserved for internal use.
C00h to C14h	Floating-point work area.
	Set by REACT if “R” to auto-start program.
	C15h.3 Set by REACT if “P” to protect program by locking it in the run mode.
C16h	Console port number (default = 1).
C17h, C18h	MTOP; highest address + 1 used by the program; if no buffers for communications, MTOP equals VARTOP (high byte, low byte).
C19h	Program memory type for REACT command; high bit set if ROM; clear if RAM.
C1Ah	REACT command’s program number.
C1Bh	ROM type; 0 = none; 1 = EEPROM; 2 = EPROM; 3 = Flash
C1Ch, C1Dh	Maximum RAM location available to program and variables.
C1Eh, C1Fh	Unused; reserved for future use.
C20h	Length of current program line.
C21h, C22h	Line number of current program line (after entering or editing).
C23h	Maximum line length (for current output port).
C24h, C25h	Pointer to address of current line length (for current output port).
C26h, C27h	Line number where error occurred.
C28h, C29h	Starting address within code memory where error message is stored.
C2Ah	Unused; reserved for future use.
C2Bh	Saved input port (when going to error handler).
C2Ch	Saved output port (when going to error handler).
C2Dh to C44h	Communication parameter storage table. Each port uses four bytes: the first byte is a flag byte, where the low bit is “echo disable” and the second bit is “line feed after carriage return enable”; the second byte is the maximum line length; the third byte is the current line length; and the fourth byte is unused. The parameters for each port follow consecutively for COM0, COM1, COM2, COM3, LPT1, and a spare.
C45h, C46h	Pointer to the display’s current line length (in case the display is opened with an alias).
C47h, C48h	Saved text pointer when the last error occurred.
C49h	Current program number.
C4Ah to CFFh	Control stack (used by FOR, DO, and GOSUB).
D00h	Error code.
D01h, D02h	Line number of ON ERROR handler (high byte, low byte).
D03h, D04h	VARTOP; top of scalar variable storage (i.e., highest address of the area where scalar variables are stored; variable storage begins at the highest address and works down) (high byte, low byte).
D05h, D06h	VARUSE; bottom of scalar variable storage (high byte, low byte).
D07h, D08h	DIMUSE; top of array variable storage (i.e., highest address of the area where array variables are stored; storage begins at the end of the program and works up) (high byte, low byte).
D09h, D0Ah	Random number seed.
D0Bh, D0Ch	Current value of VAD (high byte, low byte).

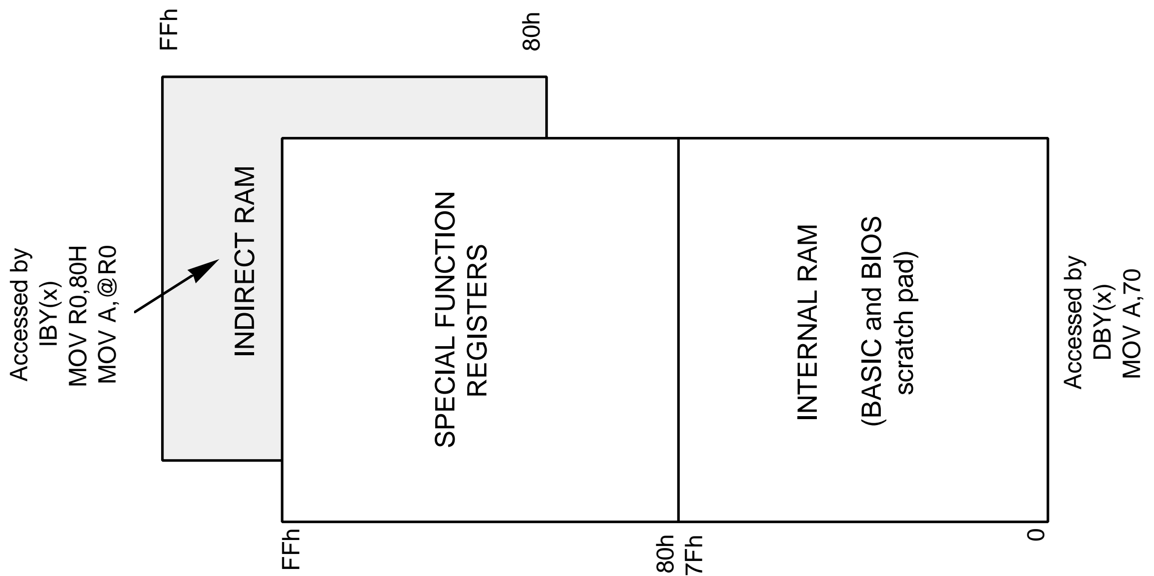
XBY Address	Function
D0Dh	Default string length plus 1 (if not otherwise assigned with SDIM).
D0Eh, D0Fh	Text pointer of ON ERROR line (in case it refers to a line number that doesn't exist).
D10h to D15h	Floating point temporary result.
D16h to D1Bh	Floating point temporary result.
D1Ch, D1Dh	Reserved for future use.
D1Eh, D1Fh	Line number of ON TIME handler (high byte, low byte).
D20h	Global PRINT USING format.
D21h to DFFh	Argument stack.
E00h to EFFh	Buffer for string manipulations.
F00h to FFFh	Input buffer for INPUT statement.
1000h to MTOP	Program, array variables, free space, and scalar variables (in that order).
1000h	Start of first program. The header consists of four bytes; the first byte, which indicates the presence of a valid program, is 0FAh. The second byte is 28h if the checksum should be valid; if it is not 28h, which happens after the user has changed a line, then the next RUN, GOTO, or GOSUB re-calculates and stores the checksum. The third and fourth bytes are the checksum. The rest of the program consists of its lines. Each line starts with a one-byte line length; the next two bytes are the line number (high byte, low byte); the remaining bytes up to the last is the program line itself. Each line ends in a carriage return (decimal code 13). The last line of the program has a length byte of 1 followed by a carriage return. If a program follows, the byte following the last line's carriage return is 0FAh.

Memory-Mapped I/O

XBY Address	Function
800h	Display's command register (for writing only).
8001h	Display's data register (for writing only).
8002h	Display's command register (for reading only).
8003h	Display's data register (for reading only).


Free space (not used by user program)
User program (when copied to ROM)
Flash EPROM (2000h to FFFFh)
(Downloadable firmware)
Microprocessor ROM
(Firmware)
Accessed by CBY(x)
MOVC A, @A+DPTR
MOVC A, @A+PC
FFFFh
A000h
2000h
1FFFh
0


MEMORY-MAPPED I/O
(Display = 8000h)
Free space (if MTOP is set lower than its default value)
Scalar variables
Free space
Arrays and strings
EXTERNAL RAM
(0 to 7FFFh)
User program (when editing)
INPUT buffer
String operation buffer
BASIC and BIOS memory
Accessed by XBY(x)
MOVX A, @DPTR
MOVX A, @R0
8000h
7FFFh
VARTOP
(= MTOP)
VARUSE
DIMUSE
1000h
FFFh
F00h
EFFh
E00h
DFFh
0


Accessed by
IBY(x)
MOV R0,80H
MOV A,@R0
INDIRECT RAM
FFh
80h
SPECIAL FUNCTION REGISTERS
INTERNAL RAM
(BASIC and BIOS scratch pad)
Accessed by
DBY(x)
MOV A,70
FFh
80h
7Fh
0

Appendix D

Reference

Command Summary

Command	Description	Examples
ABS(expr)	Returns the absolute value.	X = ABS(–5) + ABS(6*(–5))
expr1 AND expr2	Returns the logical AND.	X = A AND 3
ASC(sexpr) ASC(sexpr,iexpr)	Returns ASCII value of first character or character at position *iexpr* of string *sexpr*.	X = ASC("A") : PRINT ASC(A$,2)
ATN(expr)	Returns the arctangent.	X = ATN(A)
BEEP	Not available on the Series 30/40 Workstations.	
BIT(expr, bit#) BIT(var, bit#)	Reads from or writes to a bit in an integer.	A = 9 : X = BIT(A,2) : REM X = 0 X = BIT(A,4) = 1 : REM X = 4
CALL iexpr	Branches to built-in routine or machine language subroutine.	CALL 13
CBY(iexpr)	Retrieves a byte from program memory.	X = CBY(1000)
CHR$(iexpr)	Returns 1-character string of ASCII *iexpr*.	A$ = CHR$(64) : PRINT B$,A$
CLEAR	Clears all variables.	CLEAR
CLOCK 0 CLOCK 1	CLOCK0 disables ON TIME interrupts; CLOCK1 resets the TIME variable to zero.	CLOCK 0 CLOCK 1
CLS	Clears the display; homes the cursor.	CLS
CONT	Resumes execution after program aborted.	CONT
COPY iexpr1,iexpr2,iexpr3	Copies *iexpr3* bytes of external memory from *iexpr1* to *iexpr2*.	COPY 4000,X,21
COS(expr)	Returns the cosine.	X = COS(A/PI)
CR	Transmits carriage return without line feed.	PRINT expr, CR;
CSRLIN	Returns current line occupied by cursor.	X = CSRLIN
DATA const {,const}	Holds data in program for READ.	DATA 34, 23 : DATA "AA","ER"
DATE$	Returns or sets the date ("mm/dd/yy").	DATE$="12/23/89"
DBY(iexpr)	Reads or writes to internal RAM or special function register.	X = DBY(35)

Command	Description	Examples
DEL iexpr1,iexpr2	Deletes lines *iexpr1* through *iexpr2*.	DEL 30: DEL 10,40
DEL RAM {prog}	Deletes all programs or program *prog*.	DEL RAM 2 : DEL RAM
DIM nvar (iexpr) DIM svar(iexpr,iexpr2)	Declares the size of an array variable; for strings, *iexpr2* is the length of each string.	DIM A(30) : DIM X(25),Y(30)
DIR	Prints a directory of all programs in memory.	DIR
DO : UNTIL expr	Loops until *expr* is true (not zero).	DO : UNTIL X=5
DO : WHILE expr	Loops while *expr* is true (not zero).	DO : WHILE A<8
DUMP iexpr1, iexpr2	Prints external memory in hex format from address *iexpr1* to address *iexpr1* + *iexpr2*.	DUMP 400h,100
END	Terminates program execution.	END
ERR	After error, contains error code.	PRINT ERR
ERL	After error, contains error line number.	X = ERL
EXP(expr)	Returns "e" raised to the power of *expr*.	X = EXP(12) : REM X = 162754.77
FOR var = start TO stop {STEP incr} : NEXT	Loops a specified number of times.	FOR I = 1 TO 3 : NEXT FOR X = A TO B STEP –1 : NEXT
FREE	Returns number of bytes available.	FREE
GOSUB line#	Branches to subroutine at *line#*.	GOSUB 500 : GOSUB 1000
GOTO line#	Continues program execution at *line#*.	GOTO 40 : GOTO 100
HEX$(expr)	Converts to a string in hex format.	A$ = HEX$(165) : REM A$ = "00A5"
HVAL(sexpr)	Returns the numeric value of *sexpr* (assumed to be a hex number).	X = HVAL("FF") : REM X = 255
IBY(expr)	Reads/writes indirect internal RAM.	PRINT IBY(108)
IF expr {THEN} statement1 X=25 THEN Y=12 ELSE Y=50 {ELSE statement2}	otherwise, performs *statement2*.	If *expr* ? 0, performs *statement1*, IF IF Y > 22 THEN GOSUB 25
IN# iexpr	Switches input to port *iexpr*.	IN#0
INKEY$ {#port}	Returns a character, if available, from the current input port or from *port*.	A$ = INKEY$: B$ = INKEY$ #0
INPUT prompt,var	Requests entry to *var*.	INPUT "Enter number",A
INPUT #port, var	Performs an INPUT from *port*.	INPUT #0, B
INPUT$(iexpr{,#port})	Requests entry of length *iexpr* from *port*.	A$ = INPUT$(7) : B$ = INPUT$(0,#1)
INSTR({iexpr}, s1,s2)	Returns position of string *s2* within string *s1* (starting at position *iexpr* of *s1*).	X = INSTR(A$,B$)
INT(expr)	Returns largest whole number ≤ *expr*.	X = INT(12.8) : REM X = 12
INV(expr)	Returns 0 if *expr* ? 0; returns 1 if *expr* = 0.	X = INV(A)
IOE	Not available on the Series 30/40 Workstations.	

Command	Description	Examples
LD@ iexpr,var{,var}	Retrieves data from address *iexpr* and stores it in *var*.	LD@ 8163,A$,X
LED	Not available on the Series 30/40 Workstations.	
LEFT$(sexpr,iexpr)	Returns a string from the left of *sexpr* with a length of *iexpr*.	A$ = LEFT$(A$,4)
LEN(sexpr)	Returns number of characters in *sexpr*.	X = LEN(B$)
LET var = expr	Optional form of *var* = *expr*.	LET A = 5 : LET A = SQR(B)
LIST {#port,} begin#,end#	Lists all or part of a program.	LIST : LIST #1,40,200 : LIST ,40
LOCATE r,c,t	Positions cursor at row *r*, column *c*, with cursor type *t* (0=none; 1=box; 2=underline).	LOCATE 2,2,2 : LOCATE ,4
LOG(expr)	Returns the natural logarithm.	X = LOG(34.55) : REM X = 3.542407
MID$(sexpr,iexpr1,iexpr2) J$="WS"	Returns a new string, consisting of a partial copy of *sexpr* starting at position *iexpr1* with a length of *iexpr2*.	J$=MID$("IWS10",2,2):REM
MID$(svar,iexpr1{,iexpr2})	Places a string within *svar* starting at position *iexpr1* and continuing for a length of *iexpr2*.	A$ = "IPT" : MID$(A$,2) ="WS" REM A$ = "IWS"
MTOP	Returns or sets the top of memory.	MTOP = 7138 : PRINT MTOP
NEW	Deletes program and clears all variables.	NEW
NOT(expr)	Returns 16-bit 1's complement.	X = NOT(23) : REM X = 65512
ON ERROR GOTO line#	Enables error handling routine.	ON ERROR GOTO 700
ON ex GOSUB line#, line#	Selects subroutine number *ex* and calls it.	ON A GOSUB 100, 200, 300 (gosub 100 if A = 0, 200 if A = 1, etc.)
ON expr GOTO line#, line#	Selects line number *expr* and goes to it.	ON A GOTO 100, 200, 300 (goto 100 if A = 0, 200 if A = 1, etc.)
ON TIME=ex GOSUB line#	Sets up time-based interrupt handler to call the subroutine at *line#* when TIME ≥ *ex*.	ON TIME = 5 GOSUB 100

Command	Description	Examples
OPEN "port: rate, parity, data bits, stop bits, parameters" AS #ex	Declares communication port's parameters: *port* = COM0 (display) or COM1 *rate* = 110,30/400,600,1200,2400,4800,9600 or 19200 *parity* = N (none); O (odd); E (even) *data bits* = 7 or 8 *stop bits* = 1 or 2 *parameters* = x Enable auto-repeat and set delay to *x* Enable [Ctrl]-C interrupt Enable automatic carriage return/line feed on screen Enable automatic carriage return on screen x Wait for CTS before transmitting Disable echo during input Receive and transmit via buffers Enable IBM PC-compatible ASCII codes Send line feed after carriage return x Set line length to *x* x Set parity substitution character to an ASCII code of *x* x Set auto-repeat rate to *x* Assert RTS when receive buffer empty Don't assert RTS at all Assert RTS at start of transmitting Assert RTS when transmitting Use XON/XOFF when receiving Enable screen scrolling x Wait for input for a specified time x Use XON/XOFF when transmitting x Enable screen wraparound	OPEN "COM1:110, N, 8, 1,TX" OPEN "COM1:IB,ED,TD30/40" OPEN "COM0:IB,CE,LF,CL,SC"
expr OR expr	Returns the logical OR.	C = A OR B OR C OR D
PH0. {#port,} {expr}	Same as PRINT but prints *expr* in 2-digit hex.	PH0. XBY(0E012h)
PH1. {#port,} {expr}	Same as PRINT but prints *expr* in 4-digit hex.	PH1. 1234
PI	Equals 3.1415926.	X = PI
PLEN	Returns the length (in bytes) of program.	PRINT PLEN
POP var	Sets *var* to data at top of argument stack.	CALL 40 : POP X
POS	Returns current column number occupied by cursor.	X = POS
PR# iexpr	Switches output to port *iexpr*.	PR#0 : PR#1
PRINT {#iexpr,} {expr}	Prints *expr* to port *iexpr*.	PRINT : PRINT A*B : PRINT "Hi" PRINT #1, E, U, PLEN

Command	Description	Examples
PRINT USING sexpr	Sets up format of printed numbers.	PRINT USING "##" : PRINT USING I$
PUSH expr	Places *expr* on top of argument stack.	PUSH 1 : CALL 21
RAM prog	Selects program *prog* as the current program.	RAM 1 : RAM 2
RAM prog1 = RAM prog2	Inserts a copy of program *prog2* before *prog1*.	RAM 2 = RAM 5
REACT {par} {,par}	Specifies start-up action after reset. none = deselects all options. R = run program 1 after reset. P = protect program by locking it in run mode. Cx = set console to port *x*	REACT B : REACT R, M, N REACT RAM 1
READ var {,var}	Reads DATA value and assigns to *var*.	READ A, B : READ A(I)
REM remark	Indicates remark is a comment.	REM This is a comment.
RENUM {new}{,inc}{,start} {,end}	Renumbers the lines from *start* to *end*; the new first line number is *new* and subsequent line numbers are incremented by *inc*.	RENUM : RENUM 10,100 RENUM 1000,10,120,290
RESTORE {line#}	Resets READ pointer to first DATA item or to the first DATA item at or following *line#*.	RESTORE
RESUME	Exits error-handler and resumes execution at the line where the error occurred.	RESUME
RESUME line#	Exits error-handler and resumes execution at *line#*.	RESUME 120
RESUME NEXT	Exits error-handler and resumes execution at the line following where the error occurred.	RESUME NEXT
RETI	Returns from ON TIME handling routine.	RETI
RETURN	Returns to program from a subroutine.	RETURN
RIGHT$(sexpr,iexpr)	Returns a string from the right of *sexpr* with a length of *iexpr*.	B$ = RIGHT$(A$,4)
RND	Returns a random number between 0 and 1.	X = RND
ROM = RAM prog2	Copies RAM program to permanent memory in Flash EPROM.	ROM = RAM 1
RUN {line#}{RAM prog}	Executes current program or program *prog* at first line or *line#*.	RUN : RUN 110 : RUN RAM 1 RUN 110 RAM 1
SCAN	Not available on the Series 30/40 Workstations.	
SDIM svar(len)	Sets the maximum length of a string variable.	SDIM A$(45) : SDIM C$(30/40),B$(20)
SGN(expr)	Returns the sign of *expr*; (0 if *expr* = 0; 1 if *expr* > 0; –1 if *expr* < 0).	X = SGN(A)
SIN(expr)	Returns the sine.	X = SIN(45)
SPC(iexpr)	Prints *iexpr* spaces in a PRINT statement.	PRINT SPC(4)
SQR(expr)	Returns the square root.	X = SQR(A)

Command	Description	Examples
ST@ iexpr,var {,var}	Stores *var* starting at memory address *iexpr*.	ST@ 8163,A,X$
STOP	Terminates program execution.	STOP
STR$(expr)	Converts *expr* to a string.	A$ = STR$(X)
TAB(iexpr)	Moves the cursor to position *iexpr*.	PRINT TAB(12)
TAN(expr)	Returns the tangent.	X = TAN(A)
TIME	Sets or returns the internal variable TIME (with a maximum value of 65535.995).	PRINT TIME; CR;
TIME$	Sets or returns the current time (“hh:mm:ss”).	TIME$ = “15:34:00”
TONE	Not available on the Series 30/40 Workstations.	
TROFF	Turns off tracing of program execution.	TROFF
TRON	Turns on tracing of program execution.	TRON
VAD	Returns the last address plus 1 used by the most recent ST@ or LD@.	ST@ VAD,X VAD = 8192 : LD@ VAD,A$
VAL(sexpr)	Converts *sexpr* to a number.	X = VAL(A$)
VARPTR(var)	Returns the address in memory of *var*.	X = VARPTR(A$)
VERSION	Returns the current BASIC firmware version.	X = VERSION
XBY(iexpr)	Sets or retrieves an external memory byte.	XBY(4012h) = 23
expr XOR expr	Returns the logical XOR.	C = A XOR B XOR C XOR D

Operators

Operator	Description	Examples
expr + expr	Returns the sum.	C = A + B : PRINT A + B
expr – expr	Returns the difference.	C = A – B : PRINT A – B – C : C = 3 – 2
expr * expr	Returns arithmetic product.	C = A * B : PRINT A * B * C : C = 3 * 2
expr / expr	Returns arithmetic quotient.	C = A / B : PRINT A / B / C : C = 3 /2
expr1 ^ expr2	Returns expr1 raised to exponent expr2.	C = 45^3 : A = B ^ C
– expr	Negates an expression.	C = –A : C = –56 + A
expr = expr	Compares two expressions for equality.	X = 4 = 4 : REM X = 65535
expr <> expr	Compares two expressions for non–equality.	X = 4 <> 4 REM X = 0
expr1 < expr2	Compares expr1 for *less than* expr2.	X = 3 < 4 : REM X = 65535
expr1 > expr2	Compares expr1 for *greater than* expr2.	X = 3 > 4 : REM X = 0
expr1 <= expr2	Compares expr1 for *less than or equal* to expr2.	X = 3 <= 4 : REM X = 65535
expr1 >= expr2	Compares expr1 for *greater than or equal* to expr2.	X = 4 >= 4 : REM X = 65535

CALLs

CALL	Description	Examples
CALL 12	Clears from cursor position to end of display.	CALL 12
CALL 13	Clears from cursor position to end of line on display.	CALL 13
CALL 30	Turn off the RTS line of COM1.	CALL 30
CALL 33	Turn on the RTS line of COM1.	CALL 33
CALL 38	Runs on-line configuration program.	CALL 38
CALL 39	Runs monitor.	CALL 39
CALL 40	Returns number of characters in port #1 receive buffer.	CALL 40 : POP var
CALL 41	Returns number of characters in port #1 transmit buffer.	CALL 41 : POP var
CALL 82	Prints a list of the variables used.	CALL 82

Workstation Key Codes

The following table lists the ASCII codes and corresponding characters for each key of the Workstation's keypad. This table also lists the many two-key combination characters that the Workstation supports:

Key(s)	ASCII Code	Character or Function
F1 ↵	3	[Ctrl]-C
↓	10	Line feed
↑	11	Reverse line feed
↵	13	Carriage return
+	43	+
–	45	–
F1 F2	48	0
F2 ↑	49	1
↑ +	50	2
+ x	51	3
F1 x	52	4
F3 F4	53	5
F4 ↓	54	6
↓ –	55	7
– ↵	56	8
F3 ↵	57	9
x	127	Backspace
F1		241
F2		242
F3		243
F4		244
F1 F3	245	
F2 F4	246	
↑ ↓	247	
+ –	248	
x ↵	249	

CTRL Characters Sent to Workstation

The following table shows the function that the Workstation performs when it receives a "control" character from the Console. You can create a control character by holding down the [Ctrl] key while pressing another character.

Key	Function	ASCII Code
[Ctrl]-H	Move cursor one space to the left	08h
[Ctrl]-I	Move cursor one space to the right	09h
[Ctrl]-J	Line Feed (move cursor down one line)	0Ah
[Ctrl]-K	Reverse Line Feed (move cursor up one line)	0Bh
[Ctrl]-L	Form Feed (clear screen and home cursor)	0Ch
[Ctrl]-M	Carriage Return and, if enabled, Line Feed	0Dh
[Backspace]	Backspace	7Fh

Ports

When communicating, the display and keyboard are also considered to be a port. The following table summarizes the port names and numbers:

Name	Number	Description
COM0	0	Keypad and display
COM1	1	COM1 serial port

Appendix E

Error Codes

The following table shows the Workstation's error codes. If you write an "ON ERROR" routine, your routine must check the error code returned (ERR) as well as the line number where the error occurred (ERL) in order to handle the error.
At the end of the table are "non-recoverable" errors; if one of these errors occurs, the Workstation does not go to your error handler and instead simply stops program execution and prints the error message.

Note While most errors occur while a specific statement is executing, which means that you can easily detect its cause, there are a few errors that can occur at any time which are known as "asynchronous" errors.

Examples of asynchronous errors include buffer overflow errors, which BASIC reports at the instant the buffer overflows and without regard to the statement it is currently executing.

When BASIC reports an asynchronous error, the error number (ERR) is valid but the error line (ERL) is usually irrelevant, because the error can occur at any time.

Asynchronous errors are noted in the table below with an asterisk.

Code	Error Message	Description
1	BREAK	User pressed [Ctrl]-C or [F1]-[↵]; a user-written error handling routine can "trap" [Ctrl]-C to go to an appropriate point in the program.
10	DIVISION BY ZERO	There's still no mathematical definition for x/0.
20	OVERFLOW	Floating point operation result exceeded ±.99999999E+127 or integer operation result exceeded ± 65,535.
30	UNDERFLOW	Floating point operation result was smaller than ± 1E-127.
40	BAD ARGUMENT	Illegal address for DBY; invalid parameter or parameter format in OPEN or REACT statement; invalid argument for TIME$ or DATE$; invalid parameter for CHR$, LEFT$, RIGHT$, MID$, INPUT$ or INSTR$ statement (i.e., integer exceeds 255); bit number outside the range 1 to 16 in BIT statement; bad argument in LOCATE statement; parameter greater than 255 in many other statements.
50	TYPE MISMATCH	Function that expects a string gets a number instead; a function that expects a number gets a string instead; or a function that expects an integer gets a floating point number instead.

60	STRING TOO LONG	String expression exceeds the maximum length assigned to the variable, or a string operation resulted in an overflow of the string buffer. If the former, a program can "truncate" the string; if the latter, you can break up a single expression into several multiple expressions where each yields an intermediate result. For example, if LEN(E$) > 128, then the expression E$ = LEFT$(E$,LEN(E$)) yields the error STRING TOO LONG because the string buffer must hold E$ twice; once for the LEFT$ operator and once for the nested LEN operator. This expression must be re-written using an intermediate result, as in the following example: X = LEN(E$) : E$ = LEFT$(E$,X).
70 *	KEYBOARD BUFFER OVERFLOW	Operator pressed enough keys to fill up the keyboard buffer. This message occurs only when COM0 is opened with the IB parameter, and usually indicates that the program fails to read the keyboard frequently enough.
72	KEYBOARD INPUT TIMEOUT	Operator failed to press a key in response to an INPUT or INPUT$ statement within the time period specified by the TD parameter in the OPEN statement for COM0. This can be a useful error to generate so as not to leave a program suspended while waiting for operator input.
75	I/O RACKS DISABLED; CAN'T SCAN	Program contains a SCAN command; this command is not supported on the Series 30/40 Workstations.
80 *	COM1 RX OVERFLOW	Workstation received a character when the 255-character buffer was already full; to avoid this error, try implementing XON/XOFF or RTS handshaking (use the RX or RH parameter in the OPEN command).
81 *	COM1 TX TIMEOUT	CTS input did not come active or XON was not received within the time period specified by the CS or TX parameter; this occurs when the other device is not connected properly, not operating properly, or the program doesn't wait long enough for the other device to come ready.
83 *	COM1 PARITY	Parity error occurred during the reception of a character; this can happen because of electrical noise or improper communication parameter settings.
85	COM1 INPUT TIMEOUT	Workstation did not receive a character in response to an INPUT or INPUT$ statement within the time period specified by the TD parameter in the OPEN statement for COM1. This can be a useful error to generate so as not to leave a program suspended while waiting for input.
87	COM1 TRANSMIT OVERFLOW	User program attempted to print to COM1 when the buffer is full. This can occur subsequent to COM1 TRANSMIT HANDSHAKE TIMEOUT if the CTS input is not asserted.
	ARRAY SIZE	In DIM statement, array size exceeds 254, string length exceeds 254, or memory required exceeds memory available; otherwise, element number exceeds number of elements in array.
	A-STACK	Too many levels of parentheses in an expression or PUSHes don't match POPs or CALLs that push and pop; if the former, break up the expression into smaller expressions that yield intermediate results.
	BAD PROGRAM CHECKSUM	Program is scrambled and will not run; indicates possible problem with electrical noise scrambling the memory.

BAD SYNTAX	Invalid command or statement, or variable name includes a reserved word.
CAN'T CONTINUE	CONT won't work if a program terminated normally or with an error; CONT works only if a program terminates with a [Ctrl]-C.
CAN'T RESUME	RESUME won't work unless it's at the end of an error-handling routine.
C-STACK	Mismatch between variable following NEXT and the variable following the corresponding FOR; UNTIL, WHILE, NEXT, RETURN or RETI without corresponding control statement; too many levels of nesting; or end of program found at end of DO or FOR statement. Also, use the CLEAR command from within a subroutine clears all stacks, so the RETURN statement at the end of the subroutine will cause a C-STACK error.
EXTRA IGNORED	INPUT received either string input that exceeded the maximum string length or received more numbers than INPUT expected.
I-STACK	Expression is too complex for BASIC to handle. Almost invariably indicates that an expression contains too many levels of parentheses. The solution is to re-write the expression to eliminate nesting, re-write it so that higher-precedence operators come first, or break it up into two or three expressions that return intermediate values.
ILLEGAL DEFERRED	Attempt to execute a command in the Run mode that executes only in the Command mode, such as CONT.
ILLEGAL DIRECT	Attempt to execute a command in the Command mode that executes only in the Run mode, such as STOP or RESUME.
NO DATA	Attempt to READ found no data, either because the program contains no DATA statements or because previous READs have already exhausted all the DATA statements. This can occur if you re-start a program with a GOTO instead of RUN, so if you want to do that, put a RESTORE before the first READ.
NO SUCH PROGRAM	Reference to a program that doesn't exist; for example, RAM1 = RAM 2 when RAM 2 doesn't exist or DEL RAM 3 when RAM 3 doesn't exist.
REDIMENSION	Program contains a DIM or SDIM for an array variable that already exists either because of a previous DIM or because of a previous use that created the array by default. This can occur when re-starting a program with a GOTO instead of a RUN, but there is no remedy except to avoid executing the extra DIM or SDIM statements.
ROM WRITE	Attempt to write to a location in Flash EPROM that is already written.
OUT OF MEMORY	Program has grown too large during editing or renumbering or the program has created too many variables during execution.
STOP	Program execution halted because of a STOP statement.
UNDEFINED LINE NUMBER	GOTO, GOSUB, ON ... GOTO, ON ... GOSUB, ON TIME, or ON ERROR refers to a line number that doesn't exist in the program.

* Indicates an asynchronous error that can occur at any time during program execution, without regard to the statement BASIC is current executing. When an asynchronous error occurs, the error number (ERR) is valid but the error line number (ERL) is irrelevant.