

Professional™
300series

PRO/BASIC Language Manual

AA-N601C-TH

digital equipment corporation

Preface

INTENDED AUDIENCE

This manual is designed as a reference tool for users with some knowledge of BASIC programming.

You should also be familiar with use of the Professional 350 computer as described in the *Professional 300 Series User's Guide for Hard Disk System*.

HOW YOU SHOULD USE THIS MANUAL

Chapter 1 is on the installation of PRO/BASIC and using the PRO/BASIC Line editor. An example program is included. All readers will want to see this chapter, if only for the use of the Line editor.

Chapters 2 and 3 contain information of a fundamental nature about the components of PRO/BASIC and the use of PRO/BASIC statements. Readers knowledgeable in BASIC can scan these chapters. Those less familiar with BASIC might read chapter 2 and 3 more closely.

Chapters 4 through 7 contain reference information on the use of commands, language statements, library functions and Graphics statements of PRO/BASIC.

DOCUMENTATION CONVENTIONS

The following documentation conventions are used throughout this manual:

□ **Red type**

Red type marks information you type in to PRO/BASIC. For example:

```
10 PRINT "HELLO"
RUN
Hello
Ready
```

□ **RETURN Key**

All sample program lines and user type-in lines are terminated by the RETURN key. For example, the command to list the current program is shown as:

```
LIST
```

What you actually type is the command LIST, after which you press the RETURN key:

```
LIST RETURN
```

□ **UPPER/lower Case**

In the Syntax sections of the keywords, all items in uppercase letters must be typed exactly as they appear. Items in lowercase letters must be replaced as described in the accompanying text. For example:

```
MERGE filespec
```

The above line indicates that you should type MERGE exactly as shown and replace filespec with the appropriate file specification.

□ **[brackets]**

In the Syntax sections of the keywords, brackets enclose an optional portion of the format. When entries are stacked, one entry can be selected. For example:

```
OPEN filespec [FOR INPUT] AS FILE #channelnum [.VIRTUAL]
[FOR OUTPUT]
```

□ **{braces}**

In the Syntax sections of the keywords, braces enclose a mandatory portion of the format. When braces enclose stacked entries, you must choose one of the entries. For example:

```
{ THEN statement(s)
  THEN linenum
  GOTO linenum }
```

- ellipses

In the Syntax sections of the keywords, ellipses indicate the item may be repeated as needed. The example below shows that many channel numbers may be included in the CLOSE statement:

```
CLOSE [#channelnum[,#channelnum] ... ]
```

When used in program examples, vertical ellipses indicate that there are other statements of the program but that they are not shown. For example:

```
10 RANDOMIZE
20 A = RND
30 PRINT A
.
.
.
```

TERMS AND ABBREVIATIONS

The general meaning of terms frequently used in the Syntax sections of commands, statements, and library functions are listed below. Further clarification is provided in the text.

- **#channelnum**—a number or a numeric expression specifying a channel number.
- **expression**—a single item of data (either numeric or string), or a series of data items combined with operators—for instance an arithmetic plus sign—to produce a single data item.
- **filespec**—a file specification or a string expression that specifies a file.
- **identifier**—refers to any of the following: variable names, array variable names, library functions, and user-defined functions.
- **linenum**—a line number in a PRO/BASIC program.
- **programe**—a PRO/BASIC program name.
- **value**—a data item, numeric or string, variable or constant.

RELATED DOCUMENTATION

- *Professional 300 Series User's Guide for Hard Disk System*
- *Professional 300 Series Owner's Manual*

Contents

CHAPTER 1 INTRODUCTION AND USE OF THE LINE EDITOR

1.1	Choosing PRO/BASIC from a Menu	1
1.2	Modes of Operation	2
1.3	The PRO/BASIC Line Editor	2
1.3.1	Using the Line Editor	3
1.3.2	Making Changes to a Program.....	5
1.3.3	Numeric Keypad.....	6
1.3.4	Syntax Checking.....	6
1.4	A Sample Programming Session.....	7
1.4.1	Example 1: Immediate Mode	7
1.4.2	Example 2: Program Mode.....	8
1.4.3	A Few Other Commands.....	9

CHAPTER 2 BUILDING BLOCKS OF PRO/BASIC

2.1	Program Lines	13
2.1.1	Line Numbers	13
2.1.2	Program Documentation.....	14
2.1.3	Statements.....	14
2.2	The PRO/BASIC Character Set	15
2.3	Keywords.....	15
2.4	Constants.....	15
2.4.1	String Constants.....	15
2.4.2	Numeric Constants.....	16

2.5	Variables	17
2.5.1	Identifiers	19
2.5.2	Variables and Data Type	19
2.5.3	Conversion of Numbers of Mixed Data Types	20
2.5.4	Subscripted Variables	21
2.6	Operators and Expressions	22
2.6.1	Numeric Expressions Use Arithmetic Operators	22
2.6.2	Numeric Relational Expressions Use Relational Operators ...	24
2.6.3	Logical Expressions Use Logical Operators	25
2.6.4	Order of Precedence in Numeric Expressions	26
2.6.5	String Relational Expressions Use Relational Operators	29
2.6.6	String Concatenation	29
2.7	Functions	30

CHAPTER 3 USING PRO/BASIC

3.1	Arrays	35
3.1.1	Creating Arrays	35
3.1.2	Assigning Values	36
3.1.3	Implicit Arrays	38
3.2	Files	38
3.2.1	File Organizations	39
3.2.2	Naming Files	39
3.2.2.1	Name and Type	39
3.2.2.2	Versions	40
3.2.3	Specifying Directory and Device	40
3.3	Keyboard Input/Screen Output	41
3.3.1	Displaying Data on the Screen (PRINT statement)	42
3.3.2	Formatting with the PRINT Statement	43
3.3.2.1	Print Format	43
3.3.2.2	Print Margins	44
3.3.2.3	PRINT USING	45
3.3.3	Receiving Data from the Keyboard	45
3.3.3.1	INPUT Statement	45
3.3.3.2	LINPUT Statement	46
3.3.3.3	Storing Data in the Program (READ and DATA)	47
3.4	File Input and Output	48
3.4.1	Opening and Closing Files (OPEN and CLOSE)	48
3.4.1.1	OPEN Statement	48
3.4.1.2	CLOSE Statement	49
3.4.2	Writing to a Sequential File (PRINT #)	49

3.4.3	Reading from a Sequential File (INPUT #,LINPUT#)	49
3.4.4	Appending Records to a File	50
3.4.5	Creating a Virtual Array File (DIM # Statement).....	51
3.4.6	Writing to and Reading from a Virtual Array File.....	51
3.5	Program Errors and Error Handling	52
3.6	Program Control	53
3.6.1	Unconditional Transfer (GOTO).....	54
3.6.2	Multiple Branching (ON GOTO)	54
3.6.3	Conditional Transfer (IF)	55
3.6.4	Loops (FOR and NEXT).....	57
3.6.4.1	How a Loop Works	57
3.6.4.2	Using FOR and NEXT.....	58
3.6.5	Subroutines (GOSUB,ON GOSUB,RETURN)	58
3.7	Halting Program Execution	60
3.8	Chaining.....	61
3.9	Using Control Functions	64
3.9.1	Control Character.....	65
3.9.2	Escape Sequence.....	65

CHAPTER 4 COMMANDS

4.1	CATALOG.....	70
4.2	CONTINUE	72
4.3	DELETE.....	73
4.4	EDIT	74
4.5	EXIT	75
4.6	LIST.....	76
4.7	MERGE.....	77
4.8	NEW.....	80
4.9	OLD	81
4.10	RENAME	82
4.11	RENUMBER.....	83
4.12	RUN.....	86
4.13	SAVE.....	87
4.14	SET.....	88
4.15	SHOW	90
4.16	STEP	92

CHAPTER 5**STATEMENTS**

5.1	CALL COLLATE	98
5.2	CHAIN.....	100
5.3	CLOSE.....	103
5.4	DATA	104
5.5	DECLARE	106
5.6	DEF	108
5.7	DIM.....	111
5.8	DIM #	114
5.9	END.....	118
5.10	FOR/NEXT	119
5.11	GOSUB.....	123
5.12	GOTO	125
5.13	IF	126
5.14	CALL INKEY	129
5.15	INPUT.....	130
5.16	KILL	132
5.17	LET	133
5.18	LINPUT	135
5.19	NAME AS.....	137
5.20	NEXT.....	138
5.21	ON ERROR	139
5.22	ON GOSUB.....	142
5.23	ON GOTO	145
5.24	OPEN.....	147
5.25	PRINT.....	149
5.26	PRINT USING	152
5.27	PROGRAM	158
5.28	RANDOMIZE	160
5.29	READ.....	162
5.30	REM.....	164
5.31	RESTORE	165
5.32	RESUME	166
5.33	RETURN	167
5.34	SET CURRENCY.....	168
5.35	SET RADIX	169
5.36	SET SEPARATOR	170
5.37	STOP	171

CHAPTER 6 LIBRARY FUNCTIONS

6.1	ABS.....	176
6.2	ASCII.....	177
6.3	ATN.....	179
6.4	CCPOS.....	180
6.5	CHR\$.....	182
6.6	COS.....	184
6.7	DATE\$.....	185
6.8	EDIT\$.....	186
6.9	ERL.....	188
6.10	ERR.....	189
6.11	ERT\$.....	190
6.12	EXP.....	191
6.13	FIX.....	192
6.14	INT.....	194
6.15	LEN.....	196
6.16	LOG.....	197
6.17	LOG10.....	198
6.18	MID\$.....	199
6.19	NUM\$.....	201
6.20	PI.....	203
6.21	POS.....	205
6.22	RND.....	207
6.23	SIN.....	209
6.24	SQR.....	210
6.25	TAB.....	211
6.26	TIME\$.....	213
6.27	VAL.....	214

CHAPTER 7 PRO/BASIC GRAPHICS

7.1	Graphics Hardware.....	219
7.2	Coordinates.....	220
7.3	Viewport Coordinates.....	222
7.4	Window Coordinates.....	224
7.5	Two Graphics Statements.....	226
7.6	ASK POSITION.....	227
7.7	CLEAR.....	228
7.8	GRAPHIC PRINT.....	229
7.9	PLOT.....	231
7.10	PLOT ARC.....	233
7.11	PLOT CURVE.....	235

7.12	PRINTSCREEN	237
7.13	SCROLL	238
7.14	SET CHARACTER	240
7.15	SET CHARACTER SIZE	243
7.16	SET CHARACTER SPACING	245
7.17	SET CLIP	247
7.18	SET COLOR	248
7.19	SET COLORMAP	250
7.20	SET FILL	253
7.21	SET FILL OFF	255
7.22	SET FILL STYLE	256
7.23	SET FILLX	258
7.24	SET FILLY	260
7.25	SET FONT	262
7.26	SET ITALICS	263
7.27	SET LINE STYLE	265
7.28	SET POSITION	267
7.29	SET TEXT ANGLE	268
7.30	SET VIEWPORT	270
7.31	SET WINDOW	271
7.32	SET WRITING MODE	273

APPENDIX A THE DEC MULTINATIONAL CHARACTER SET

APPENDIX B KEYWORDS

APPENDIX C LOGICAL OPERATORS

C.1	Numeric Values in Logical Expressions	289
C.1.1	True and False Are Actually Numeric Values	289
C.1.2	Integers in Logical Expressions	290
C.1.3	The Logical Complement of -1 (true) is 0 (false)	291
C.2	Masking	292

APPENDIX D ADVANCED PROGRAMMING TECHNIQUES

APPENDIX E PRO/BASIC PROGRAM ERROR MESSAGES

1

Introduction and Use of the Line Editor

Chapter 1

Introduction and Use of the Line Editor

This chapter introduces PRO/BASIC, selecting and leaving PRO/BASIC, using the PRO/BASIC Line Editor, and using HELP in PRO/BASIC. A programming example is included.

1.1 CHOOSING PRO/BASIC FROM A MENU

PRO/BASIC appears on either the Main Menu or the Additional Applications Menu. When making selections from menus use the ↓ arrow key to move the pointer.

- If PRO/BASIC appears on the Main Menu, move the pointer to PRO/BASIC and press DO.
- If PRO/BASIC does not appear on the Main Menu, move the pointer to Additional Applications and press DO. Another menu will appear. Move the pointer to a Group menu to display a list of applications. Move the pointer to PRO/BASIC on this menu and press DO.

Once you have chosen PRO/BASIC and pressed DO, an identification appears:

PRO/BASIC Version 1.0

(C) Copyright 1982 DIGITAL EQUIPMENT CORPORATION

To leave PRO/BASIC, press EXIT. When you press EXIT to leave PRO/BASIC, a message appears if you have made changes to the program:

Error 109: Unsaved changes, type EXIT again to exit

If you want to include the latest changes to the program, use the **SAVE** command, after which you can press **EXIT** again. Otherwise, follow the instruction and just press **EXIT**.

1.2 MODES OF OPERATION

PRO/BASIC can be used in two modes: immediate mode and program mode.

Immediate mode means that **PRO/BASIC** will execute your instructions as soon as they are entered. For example:

```
PRINT 44/11
4
```

Program mode is how you enter and execute programs. To indicate that the line you type is part of a program, start the line with a number. The line is then stored with other program lines and executed with the **RUN** command. For example:

```
10 PRINT 44/11
RUN
4
Ready
```

1.3 THE PRO/BASIC LINE EDITOR

You use the **PRO/BASIC** Line Editor to enter a line or change any data on the screen; it handles one line at a time.

When you type, a blinking rectangular marker moves from left to right on the screen. This marker is called the cursor. The cursor indicates the location on the screen where the next character will be typed, inserted, or deleted.

Keys for printing characters, the arrow keys, and the delete key will repeat if held down for more than a short time.



Some keys are not used by the Line Editor. **PF1**, **INSERT HERE**, and **FN4** are examples. These keys will beep when pressed.

1.3.1 Using the Line Editor

Below is a list of the keys you use while in the PRO/BASIC Line Editor. The keys are grouped according to the operation they perform.


Moving the Cursor

You can move the cursor back and forth in the line, entering characters at any location. If you enter a character between other characters, the characters to the right of the cursor will move to the right to make room for the new character.

Key	Function
	Moves the cursor one character to the right.
	Moves the cursor one character to the left.

Deleting Characters

Whenever a character is deleted, any characters to the right of the character move to the left, filling in the deleted character's position.

Key	Function
	Deletes the character to the left of the cursor.
F12	Deletes the character at the cursor.
F13	Deletes all characters to the end of the line.
INTERRUPT	Deletes the entire line.
DO	(Press INTERRUPT then DO)

Entering a Line

All the keys listed below enter the current line, that is, the line the cursor is in. The last three keys are used in editing a program.

Key	Function
ENTER	Enters the current line.
RETURN	Enters the current line.

F11

Enters the current line. For use when editing a program. F11 also creates a copy of the current line to edit. This key is useful if you have several similar lines to enter. You can type the first line, and then just edit copies. You must change the original line number, otherwise the original line is overwritten when the copy is entered.



Enters the current line. For use when editing a program. Displays the following line for editing. This is useful when making edits to many lines in an existing program.



Enters the current line. For use when editing a program. Displays the previous line for editing. This is useful when making edits to many lines in an existing program.

Using Other Keys**Key****Function****HELP**

During program execution, HELP is available in PRO/BASIC after an error message is displayed, indicating that an error has occurred in processing. Press HELP to display explanatory information on the latest error.

EXIT

Exits PRO/BASIC and returns to the Main Menu.

TAB

Inserts spaces until you reach a tab stop. Tab stops occur every eight character positions.

CONTROL

The control key can be held down while another key is pressed, producing a value other than the face value of the key pressed. Such values are called control characters. Refer to Control Functions in Chapter 3 for more information on control characters.

1.3.2 Making Changes to a Program

Any instruction given to PRO/BASIC and preceded by a number is considered a program line. Following is an explanation of a few time-saving techniques for use when editing a program.

Moving Around and Displaying the Program

If the cursor is on a program line, press ↑ to enter the current line and display the previous line, press ↓ to enter the current line and display the following line.

If you change the line number of an existing program line and press ↑ or ↓ the line above or below the line's original location will display.

If the cursor is on a blank line, press ↓ to display the first line in the program or ↑ to display the last line in the program.

To display the entire program, or a portion of the program, use the LIST command. Simply type LIST, or LIST followed by a line number or a range of line numbers. Refer to the LIST command in Chapter 4 for more information.

Adding a Program Line

To add a line to a program, enter a line number (0-32767) followed by one or more characters and press ENTER. The line is stored as part of the current program.

Insert a line between existing lines in the program by entering a line with a line number which is between two existing line numbers.

Replacing or Changing a Program Line

To replace a program line enter a line number that matches an existing line number, followed by one or more characters. The new program line replaces the old one.

To change a program line you can use the above method or use the EDIT command. Simply type EDIT followed by a line number. The specified program line will display. You can then make any changes and press ENTER to enter the line.

Deleting a Program Line

To delete a line, enter only the line number of an existing line and press ENTER.

Another way to delete lines in programs is to use the DELETE command. Simply type DELETE followed by a line number or a range of line numbers. Refer to the DELETE command in Chapter 4 for more information.

1.3.3 Numeric Keypad

The keys of the Numeric keypad have the same values as the keys on the standard keyboard. ENTER has the same value as RETURN. The PF1–PF4 keys along the top row of the numeric keypad have no function in immediate mode. During program execution these keys, as well as the keys along the top row of the keyboard, and the keys of the editing keypad generate a variety of escape sequences when pressed. Use CALL INKEY to get these characters in your program. Refer to the Terminal Subsystem Manual for the escape sequence generated by each key.

1.3.4 Syntax Checking

When you enter a line to PRO/BASIC, in immediate mode or in program mode, it is checked for adherence to 'syntax' requirements, for example; that quotation marks match, that a plus sign be used between two numbers to be added, that an equals sign is used to assign a value, or that parentheses be used in matching pairs.

If PRO/BASIC finds no syntax errors in the line, it is accepted.

If there is incorrect syntax in the line, PRO/BASIC moves a pointer to the error, displays an error message to identify the incorrect syntax, and displays the statement again with the cursor positioned at the problem.

We'll make an error to demonstrate how syntax checking works. The example below uses the PRINT statement to tell the computer to display the phrase "My dog has no nose" on the screen. We will not provide ending quotation marks in the immediate mode statement.

```
PRINT "My dog has no nose^
```

```
Missing quote at end of string
```

```
PRINT "My dog has no nose _
```

We can correct the error by placing matching quotation marks at the end of the phrase. Once the error has been corrected the computer can print the phrase.

```
PRINT "My dog has no nose" RETURN
My dog has no nose
```

When entering numbered lines, rather than immediately fixing incorrect syntax, you can press RETURN, which will store the line for you. The syntax error will not be reported again. You can return to it later and correct it.

1.4 A SAMPLE PROGRAMMING SESSION

In this section we will look briefly at PRO/BASIC's two modes, immediate mode and program mode—and see an example of each use.

The first example shows how immediate mode calculations use the Professional computer as a calculator. The second example shows the use of PRO/BASIC in program mode, and what a program looks like. You should be in PRO/BASIC now to try the following examples on your Professional computer.

1.4.1 Example 1: Immediate Mode

To work out how many miles to the gallon your car goes you do a quick computation:

$$10 \overline{)250}$$

The calculation is simple enough; divide miles traveled by gallons used. To do this on your Professional computer, enter the following and press RETURN:

```
250 / 10
```

The number that displays after you press RETURN is the result of the calculation you entered.

In immediate mode the computer does what you tell it immediately. However, your instructions are limited to what can fit on one line, and they are not saved after execution.

We can now continue with a program which will do that calculation for you, any time you want.

1.4.2 Example 2: Program Mode

The program in this section calculates the distance/fuel ratio for a car.

A program uses program mode. In program mode your instructions to the computer remain after they are executed, and you must type the RUN command in order to execute them. So, type in the program below:

```

10 PRINT 'Allow me to calculate your distance/fuel ratio'
20 PRINT 'Please type in the number of miles traveled'
30 INPUT MILES
40 PRINT 'Please type in the number of gallons used'
50 INPUT GALLONS
60 LET MILEAGE = MILES/GALLONS
70 PRINT 'That comes to';MILEAGE;'miles per gallon'
80 PRINT 'Have a Nice Day'
90 END

```

You can correct any typing errors by using the left arrow and the right arrow and DELETE, or you can retype the line. Lines can be typed in any order. PRO/BASIC stores the lines according to the order of their line numbers, regardless of the sequence they are entered in.

Type RUN and press RETURN to start the execution of the program you just typed in. You should see the following on the screen:

```

Allow me to calculate your distance/fuel ratio
Please type in the number of miles traveled
? 250

```

The program pauses here so that you can enter a number. Enter the number 250.

```

Please type in the number of gallons used
? 10

```

Here the program pauses again so that you can enter a number. Enter the number 10. You should then see the following:

```

That comes to 25 miles per gallon
Have a Nice Day

```

Type RUN and press RETURN to run the program again. Try using different numbers.

1.4.3 A Few Other Commands

PRO/BASIC's SAVE command stores a copy of the program on a storage device (disk or diskette).

To use the SAVE command, type SAVE followed by a name you give the program. Keep the name short, and use only letters, perhaps PROGRAM. For example:

```
SAVE some name you give it
```

Now, to prove a point, let's use another command. This command will erase the program that you executed with the RUN command. Type the following:

```
DELETE 10 - 90
```

Now type RUN. This should cause the following error message to display:

```
Error 111: No program to run
```

Now use the OLD command to restore a copy of the program from the storage device. Type:

```
OLD whatever you named it before
```

To verify that you have a copy of the program back, type RUN again, or type LIST to display the program on the screen.

What happened through all this is simply the transfer of a copy of the program from the place where it was created, to the storage device, and then back to the place where it was created.

The place where your program was created is called memory. Memory is the portion of the computer that stores a program for execution. A program can only be executed when it is in memory; it cannot be executed when it is on a disk or diskette.

The SAVE command made a copy of the program and placed it on the storage device. With the DELETE command you erased the copy in memory. With the OLD command you moved a copy of the program back to memory.

2

Building Blocks of
PRO/BASIC

Chapter 2

Building Blocks of PRO/BASIC

This chapter treats the elements of PRO/BASIC that you use to write programs.

2.1 PROGRAM LINES

A line in a PRO/BASIC program consists of the following:

line number one or more statements

A line is ended and entered in PRO/BASIC when the RETURN key is pressed. A line with just one statement is shown below:

```
10 PRINT 'HELLO'
```

2.1.1 Line Numbers

Line numbers precede PRO/BASIC program statements and associate the statements with the line numbers. A line number before a statement (or series of statements) saves that statement for execution until RUN is typed and the RETURN key is pressed. A program consists of a series of numbered lines containing statements.

Programs are executed in order from low line numbers to high line numbers. No two lines can have the same line number. Line numbers also serve as addresses when you want to alter a program's normal order of execution (altering normal execution is treated in Chapter 3).

Line numbers are valid in the range of 1 to 32767.


The convention of using sequences of ten for line numbers allows you to modify a program by inserting up to nine lines between existing lines.

The normal order of program execution is as follows:

```

10
20
30
35
40
50
60
70

```



2.1.2 Program Documentation

Explanatory comments can be written into the program to explain what's happening in a program. Comments can be included anywhere in a program as a numbered line with a REM (remark) statement. For example:

```
400 REM This section prints final totals
```

Anything that follows a REM statement is ignored by the computer.

2.1.3 Statements

A statement is a grammatical unit for PRO/BASIC, like a sentence in natural language. Statements in a program line direct processing, depending upon the combination of elements it contains.

One or more statements may appear in a program line; the maximum length of the program line is 80 characters. If a number of statements are included on the same line, they must be separated by a backslash character \ or by two slash symbols (/). The two following examples have the same effect:

```

100 PRINT A\PRINT B\PRINT C      100 PRINT A
                                   110 PRINT B
                                   120 PRINT C

```

The remainder of this chapter lists the individual elements of PRO/BASIC that are used, alone or joined to other PRO/BASIC components, to write programs.

2.2 THE PRO/BASIC CHARACTER SET

To write PRO/BASIC programs, you can use the following characters. Note that you can use the DEC Supplemental Graphic Set characters only in strings. (Refer to Section 2.4.1, String Constants for more information on strings.)

alphabetic characters	A–Z upper case and lower case
numeric characters	0–9
special characters	+ – * / ' " = - % \$ # ! () \ , ; . < >

DEC Supplemental Graphic Set (use only in quoted strings)

? i ‡ £ ? Y ? 8 x 0 ¢ « ? ? ? ? ? ± ‡ J ? µ ¶ · ? 1 ¢ » ¢ ¢ ? ¢
 Å Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß
 à á â ã ä å æ ç è é ê ë ì í î ï ñ ò ó ô õ ö ø ù ú û ü ý ? ?

PRO/BASIC translates all lower case DEC Multinational alphabetic characters to upper case. All other characters, and characters in strings are accepted as entered. (Appendix A contains a table of the DEC Multinational Character Set.)

Note that the DEC Supplemental Graphic Set uses 8-bit character codes. The remaining characters of the DEC Multinational Character Set use 7-bit character codes. Refer to the Terminal Subsystem Manual for more information on 7- and 8-bit character codes.

2.3 KEYWORDS

Keywords belong to PRO/BASIC, and dictate the action of a program statement. Because they have specific meanings to PRO/BASIC, they cannot be used as variable names. Keywords must be separated from other characters with a blank space. Refer to Appendix B for a list of keywords.

2.4 CONSTANTS

A constant is a data item with a fixed value used in the execution of a PRO/BASIC program. Constants can be specified in the program, read from a file, or entered during execution. Constants can be either string or numeric.

2.4.1 String Constants

A string constant is a series of alphabetic, numeric, and special characters enclosed in single or double quotation marks. String data (also called character data) is treated as a single unit of data. All special characters and DEC Supplemental Graphics characters are allowed in strings.

Following are some examples of string constants:

```
"Name change for DEC Japan"
"the Statue of Liberty"
"$1,999.99"
```

When handling data, there must be agreement in data type; for example a string ("cat") cannot be added to a numeric value (5).

2.4.2 Numeric Constants

Numeric constants consist of the numeric characters 0–9. Numeric data is positive, negative, or zero. Never use commas in numbers when calculating in PRO/BASIC.

Numeric data can be represented as integer or real constants.

- *Integer Constants*—An integer is a whole number between –32768 and 32767 with no decimal point and fractional part. In PRO/BASIC the percentage sign (%) suffix is used to identify an integer. Integers require the least storage space and are the fastest of the numeric constants to process. For example:

```
1234%      2500%
1%         -1451%
```

- *Real Constants*—A real constant has one or more decimal digits, either positive or negative with an optional decimal point and fractional part.

```
25         1
5659       94.36
```

Exponential notation is used to represent very large or very small real constants. Exponential notation expresses a number as a constant multiplied by the appropriate power of 10. A number represented exponentially has the format: an integer, or real constant (the mantissa) followed by the letter E (indicating ten to the power of) followed by a whole number (the exponent). Both the mantissa and the exponent are optionally signed. For example:

```
8.0E+3      5E+3
.4593E-4    34E-2
```

Real constants are of two different types; single precision and double precision—which differ in the amount of precision of the data they represent. The precision of a number is determined by the storage space it uses.

- *single precision*—real constants of up to 6 digits with optional decimal point and fractional part. Single precision real constants use twice as much space as integers and are slower to process. Some examples follow:

57.9
 -2.11E-05
 4501.0

- *double precision*—real constants of at least 7 digits and up to 16 digits with optional decimal point and fractional part. Double precision real constants use four times as much space as integers. Some examples follow:

34567812
 3232.72984
 -2.10534E-4

Note that although single and double precision each store a different number of digits internally, they display the same number of digits. It is possible to display the additional digits of double precision constants by using the PRINT USING statement.

If a real number of six or fewer digits is entered to PRO/BASIC it is stored with single precision, otherwise it is stored with double precision. If the mantissa of a real number in exponential format is of six or fewer digits, the number is stored with single precision, otherwise it is stored with double precision.

When more precision is used more space in memory is taken up. To use many double precision real numbers will slow the execution of a program. Use double precision only when a high degree of accuracy is needed. (Refer to the DECLARE statement in Chapter 5 for more information on creating single and double precision real variables.)

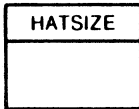
2.5 VARIABLES

A variable is a location that stores a piece of information. A variable can contain one value.

A variable is created simply by making a reference to its name. The variable name stands for the value in the variable. A value is assigned to, taken from, and changed in the variable by referring to the variable name.

Variables (like constants) are either numeric or string. Numeric variables handle only numeric data, and string variables handle only string data.

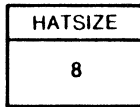
The following example shows a variable named HATSIZE. The example consists of a storage area (now empty) labeled HATSIZE. HATSIZE is the variable name that stands for a space in the computer's memory where a value can be stored.



Values can be assigned to variables with an assignment statement (the LET statement.) Refer to the LET statement in Chapter 5.

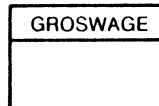
The following example shows that a value, 8, has been assigned to the variable HATSIZE. Note that the location receiving the value is on the left in the statement, and the value being assigned is on the right.

10 LET HATSIZE = 8



Variables can receive the result of calculations in a program. For example:

160 LET GROSWAGE=PAYRATE*HOURS



After creating a variable but before assigning a working value to it, PRO/BASIC sets a numeric variable equal to a value of zero and sets a string variable equal to a zero length string, which contains no characters. This process is called initialization.

2.5.1 Identifiers

An identifier is used to name variables, arrays, and functions.

All identifiers start with an alphabetic character, which is followed by a series of alphabetic and numeric characters. The underscore character (`_`) can be used in variable name identifiers. Identifiers can contain up to 32 characters including a suffix to indicate data type. (Refer to Section 2.5.2, Variables and Data Type.)

Do not use keywords as identifiers. Do not use the DEC Supplemental Graphic Set characters in identifiers. An identifier preceded by `FN` is taken to be a user-defined function.

2.5.2 Variables and Data Type

When a variable is created the type of data it can store is specified.

Variables can store either string or numeric constants; and if the constants are numeric, they can be either integer or real.

Use a suffix in the variable name to indicate the data type of the variable, and therefore, the data type of the value which the variable can store. PRO/BASIC uses three types of variables:

- *String*—Indicated by a dollar sign (\$) in the last position of the variable name. A string variable stores string data up to a maximum length of 255 characters. An example of a string variable follows:

FIRSTNAME\$

- *Integer*—Indicated by a percentage sign (%) in the last position of the variable name. An integer variable stores a whole number. An example of an integer variable follows:

NUM1%

- *Real*—No suffix. A real variable stores a real number of either single or double precision, depending upon the setting of the `DOUBLE` switch (see the `SET` command in Chapter 4) or the use of the `DECLARE` statement (see the `DECLARE` statement in Chapter 5). An example of a real variable follows:

NUM1

2.5.3 Conversion of Numbers of Mixed Data Types

When you are handling numeric data in variables, program execution will be faster if all variables are of the same data type and precision; however this is not essential. Conversion is automatic if data assigned to a variable is not of the same type as the variable.

The conversion favors the highest precision of any of the values in the calculation: if an integer value is paired with a real variable, the conversion is to real; if a real value is paired with a double precision value the conversion is to double. These conversions are shown in the following example, where I% indicates a value of integer data type, and S and D indicate real number values of single and double precision in the calculation:

I% = (S * D) * (I% - S)	calculation
D S	after operations
D	result becomes double precision
I% =	result stored in integer variable

During assignment, the conversion is to the data type of the variable that holds the final value. In the example above the result is in double precision but is finally assigned to an integer variable. During assignment, the value is truncated, as shown in the following example:

```
10 DECLARE DOUBLE A
20 A=12.3456789
30 B%=12.3456789
40 PRINT USING '##.#####', A,B%
RUN
12.3456789
12.0000000
Ready
```

In line 20, the double precision value is assigned to a double precision variable and displays 9 digits. In line 30, the same value is truncated to a whole number when assigned to an integer variable.

Double precision values are not rounded when assigned to single precision variables. For example:

```
10 DECLARE DOUBLE A
20 A=12.3456789
30 C=A
40 PRINT USING '##.#####', A,C
RUN
12.3456789
12.3456000
Ready
```

During assignment of a low precision result to a higher precision variable, the final result stored can be only as accurate as the lowest precision permits. For example:

```
10 DECLARE DOUBLE B
20 A=PI
30 B=A
40 PRINT USING '#.#####',PI,A,B
RUN
3.141592653589793
3.1415900000000000
3.14159
```

In the preceding example, the value of *PI* is assigned to *B* after being stored in *A*. Note the value of *PI* as printed from variable *B*. The double precision representation in variable *B* is accurate only to the fifth digit after the decimal point, since *A* is a single precision variable.

2.5.4 Subscripted Variables

A series of variables of the same data type can be given the same name. This is called an array. With an array you can refer to many variables by one variable name, called an array-name.

The array-name is common to all variables in the series. A number is used to distinguish one variable from the next. This number is called a subscript. A subscript appears enclosed in parentheses immediately after the array name. To identify a particular variable (an element) in an array, use a subscripted variable name.

An array-name can be string, integer, or real data type and indicates the data type the array can store. The example below shows three arrays, each of a different data type. Element 6 of each array is referenced in each of the examples:

A\$(6) A%(6) A(6)

Refer to Chapter 3 for more information and examples about arrays.

2.6 OPERATORS AND EXPRESSIONS

An operator is a symbol used in an expression to specify an action to be performed on operands.

An expression can have one or more operands, and zero or more operators. Operations in an expression are executed until a single value is produced.

Operands are the constants, variables, or functions that are acted upon by the operators.

There are four groups of operators. The four groups and associated symbols are as follows:

arithmetic	(^ + - * /)
relational	(< > = >= <= <>)
logical	(NOT AND OR XOR)
concatenation	(+)

Numeric expressions use arithmetic, relational, or logical operators. String expressions use relational operators and concatenation. An expression with no operators takes its data type from the value it contains.

2.6.1 Numeric Expressions Use Arithmetic Operators

Numeric expressions use integer or real data type operands and arithmetic operators.

PRO/BASIC uses the standard arithmetic operations and their operators. When PRO/BASIC evaluates an expression, it does some operations before others, and works through the expression from beginning to end a number of times.

The operators below are listed in order of precedence, that is, according to which operator will be executed during the first pass PRO/BASIC makes through the expression, and which operators are then executed in subsequent passes.

Operator	Operation	Example
-	unary minus	-3
^(or **)	exponentiation	2^3
*	multiplication	2*3
/	division	2/3
+	addition	2+3
-	subtraction	2-3

When both values in a division operation are integer data type, integer division is used and the result is truncated to an integer.

```

10 A,A%=3.25
20 B,B%=2.75
30 PRINT A;'/';B,A;'/';B%,A;'/';B,A;'/';B%
40 PRINT
50 PRINT A/B,A/B%,A%/B,A%/B%
RUN
3.25 / 2.75      3.25 / 2      3 / 2.75      3 / 2
1.18182          1.625          1.09091        1
Ready

```

In the preceding example real number division is used in the first three division operations, where at least one of the values are not integer, and integer division is used in the last division operation, where both operands are integers.

PRO/BASIC does not support exponentiation performed on negative single or double precision bases to real number powers.

In order to perform exponentiation on negative integer bases the power must also be an integer. For example:

```

-2% ^ 1    is an invalid operation
-2% ^ 1%   is a valid operation

```

2.6.2 Numeric Relational Expressions Use Relational Operators

Numeric relational expressions compare two integer or real values. If the comparison is correct, a true value is returned. If the comparison is incorrect, a false value is returned. (Refer to Appendix C for more information on values returned by relational operators.)

The relational operators are as follows. All relational operators have the same order of precedence, that is, after other operators of higher precedence are executed, relational operators are evaluated as they are encountered.

Operator	Operation	Example
<	less than	2<3
>	greater than	2>1
=	equal to	2=2
>=	greater than or equal to	3>=3
<=	less than or equal to	2<=3
<>	not equal to	2<>3

Because all the examples above are true, they would all return a true value.

Relational expressions are often used with IF statements which test for certain conditions and make decisions based on the result. For example:

```
10 INPUT A
20 IF A<=0 THEN PRINT 'Value must be greater than 0, reenter';\GOTO 10
30 PRINT 'OK'
RUN
? -1
Value must be greater than 0, reenter ? 0
Value must be greater than 0, reenter ? 1
OK
Ready
```

In the preceding example a numeric value is received from the keyboard and assigned to variable A. In line 20 the contents of A are compared to 0; if the value in A is equal to or less than 0 a message is displayed and program control returns to line 10. There the prompt is repeated. When a value greater than 0 is entered, the message 'OK' is displayed.

2.6.3 Logical Expressions Use Logical Operators

Logical expressions use logical operators to perform logical operations on operands. Operands can be relational expressions, numeric values, or other logical expressions.

Logical expressions can combine relational expressions and evaluate their true/false values to produce a single true/false result.

The logical operators are as follows, in order of precedence.

- **NOT**—The logical opposite of *A*. If *A* is true, *NOT A* is false. If *A* is false, *NOT A* is true.
- **AND**—The logical product of *A* and *B*. *A AND B* is true only if both *A* is true and *B* is true.
- **OR**—The logical sum of *A* and *B*. *A OR B* is true if either *A* is true or *B* is true.
- **XOR**—The logical exclusive *OR* of *A* and *B*. *A XOR B* is true if either *A* or *B* is true, but not both.

The following truth tables show the results the operator produces depending upon the values of operands *A* and *B*. A true value is indicated by *T*, and a false value by *F*.

NOT

A	NOT A
T	F
F	T

AND

A	B	A AND B
T	T	T
F	T	F
T	F	F
F	F	F

OR

A	B	A OR B
T	T	T
T	F	T
F	T	T
F	F	F

XOR

A	B	A XOR B
F	F	F
T	F	T
F	T	T
T	T	F

In the following examples the IF statement uses a logical operator to compare two relational expressions. Logical operators are frequently used for testing a number of relational operands, though this is not their only use: an IF statement can perform a test on a logical operand to return a result. (Refer to Appendix C for more detail on logical expressions.)

In the example below, program control will pass to line 20 if a true value is produced by the AND operator.

```
IF A>B AND C<D THEN GOTO 20
```

The relational operands ($A > B$), ($C < D$) each evaluate to a true or false value, determined by the values in the variables A, B, C , and D . If both operands are true, the AND operator produces a true value. Refer to the truth table for AND to see the possible combinations of true and false values.

Another example of relational operands in a logical expression follows. If the value tested is less than 18 or greater than 65, control will pass to the subroutine at line 2000.

```
IF AGE < 18 OR AGE > 65 THEN GOSUB 2000
```

2.6.4 Order of Precedence in Numeric Expressions

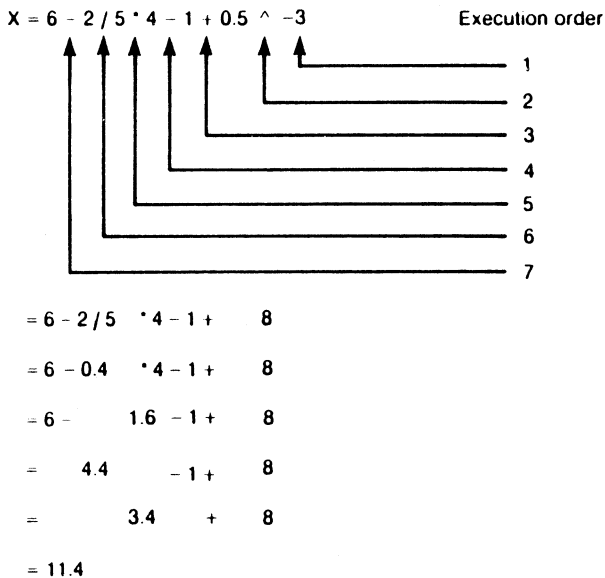
When many operators appear in an expression they are evaluated in an established order. The order of execution within each group of operators is listed in the treatments of the operators. There is also an order of execution among the groups of operators which is as follows:

ARITHMETIC	unary minus
	exponentiation
	multiplication/division
	addition/subtraction

RELATIONAL	all operators have same order of precedence
LOGICAL	NOT
	AND
	OR
	XOR

The order of precedence can be changed using parentheses. Anything enclosed in parentheses will be executed first.

The operations are executed one step at a time, from the highest to the lowest priority. Operators with the same precedence are executed as they are encountered, moving from left to right. An example of the execution of an expression according to the order of precedence is shown in Figure 1:



If we modify the order in Figure 1, with parentheses, we find that the expression no longer evaluates to the same number, as shown in Figure 2:

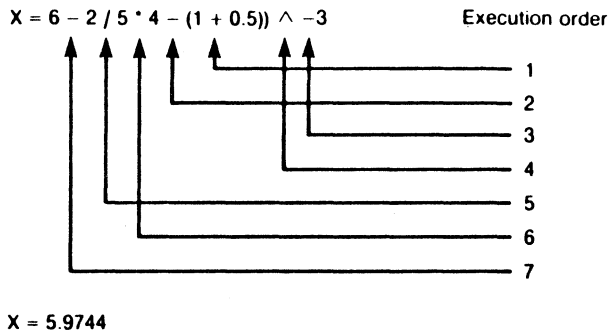
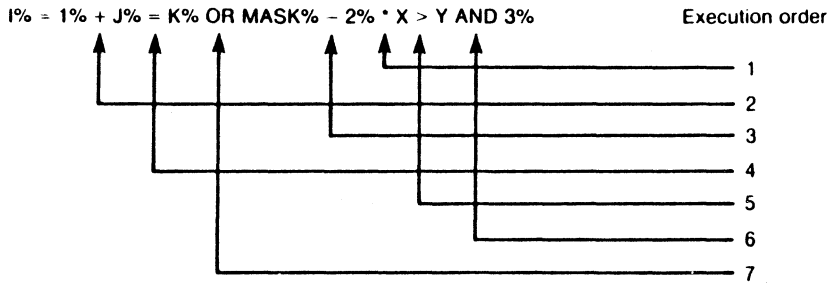


Figure 3 shows another example of expression evaluation with a combination of arithmetic, relational, and logical operators.



Refer to Appendix C for more information on values returned by relational and logical operators.

2.6.5 String Relational Expressions Use Relational Operators

Relational operators can also be used to compare two strings to determine which string precedes the other alphabetically. Two factors are considered: string contents and string length.

Strings are compared character by character until a difference is found. The comparison uses the numeric equivalent of the characters in the character set. (Appendix A contains a table of all DEC Multinational Characters.) For example:

```
10 IF 'a' < 'b' THEN PRINT 'Y'
```

In the preceding example 'a' is less than 'b' because the numeric value of 'a' is less than the numeric value of 'b'. Therefore, the relational expression is true.

The other factor is string length. In a comparison of two strings, identical except that one is longer than the other, the shorter string is 'filled' with spaces (DEC Multinational value 32) to generate strings of equal length. PRO/BASIC compares the remaining characters in the longer string against these spaces. For example:

```
10 A$ = 'ABCDE'
20 B$ = 'ABC'
30 IF B$ < A$ THEN PRINT "B$ COMES FIRST" ELSE PRINT "A$ COMES FIRST"
40 END
RUN
B$ COMES FIRST
Ready
```

2.6.6 String Concatenation

String concatenation joins character strings: end of the first to front of the second. The plus sign (+) indicates concatenation. For example:

```
10 A$ = 'PRO/' + 'BASIC'
20 PRINT A$
RUN
PRO/BASIC
Ready
```

2.7 FUNCTIONS

A function is a series of statements that accepts values and returns a single result. Functions can do complex computations for you and save program writing time and effort.

To use a function, specify the name of the function, and pass values to it. A function's type is the same as that of the result it returns.

PRO/BASIC includes many functions for handling numeric data or for handling string data. These are called library functions and are at the end of this section.

In addition, the user can write functions to complement the library functions. These are called user-defined functions. Refer to the DEF statement in Chapter 5 for more on user-defined functions.

The table below lists by category all the PRO/BASIC library functions.

Library Functions by Category

Name	Effect
Trigonometric	
PI	The PI function returns the value of pi
SIN	The sine function returns the sine of an angle
COS	The cosine function returns the cosine of an angle
ATN	The arctangent function returns the arctangent of an angle
Algebraic	
SQR	The SQR function returns the square root of a number
EXP	The EXP function returns the value of e, an algebraic constant, raised to any power
LOG	The LOG function returns the natural logarithm of a number
LOG10	The LOG 10 function returns the common logarithm of a number
INT	The INT function returns the largest integer equal to or less than the number provided
ABS	The ABS function returns the absolute value of a number
FIX	The FIX function returns the integer portion of an expression

Library Functions by Category (cont.)

Name Effect

String

- LEN** The LEN function determines the length of a string
- POS** The POS function searches for the position of a set of characters in a string
- MID\$** The MID function extracts segments from a string
- EDIT\$** The EDIT function edits a string

Conversion

- ASCII** The ASCII function converts from DEC Multinational character to the equivalent decimal value
- CHR\$** The CHR\$ function converts from decimal value to the DEC Multinational character equivalent
- NUM** The NUM function returns a string of numeric characters formatted as it would be by a PRINT statement
- VAL** The VAL function returns the number represented by the specified string

Error Handling

- ERL** The ERL function stores the line number where the last error occurred
- ERR** The ERR function stores the number of the latest error
- ERT\$** The ERT function accesses the error text associated with each error number

File Handling

- TAB** The TAB function moves the print position to the specified column
- CCPOS** The CCPOS function stores the current position of the cursor on the current line

DATE and TIME

- DATE\$** The DATE\$ function returns the current date
- TIME\$** The TIME\$ function returns the current time

Random Number

- RND** The RND function returns a random number between but not including 0 and 1
-

3

Using PRO/BASIC

Chapter 3

Using PRO/BASIC

This chapter explains how some PRO/BASIC statements are used to write programs.

3.1 ARRAYS

An array is a series of variables referred to by the same name. The variables are numbered, and are referred to by that number, called a subscript.

An array handles a series of data values more efficiently than a number of variables would; an array can be moved or processed as a unit, without the need to handle many separate variables.

3.1.1 Creating Arrays

You can create an array with the DIM statement by specifying the name and the size of the array. The array name can be any valid variable name (with the proper suffix to indicate the type of data it stores). Subscript(s) determine the size of the array.

One dimension is established in the array for each subscript used. The dimension's size is determined by the number which established it. In the example below, AARDVARK is the name of the array and 3 is the subscript.

```
200 DIM AARDVARK(3)
```

You can think of array AARDVARK as looking like this:

AARDVARK(0)	AARDVARK(1)	AARDVARK(2)	AARDVARK(3)

Dimensions are numbered from 0 to the subscript value, so array AARDVARK, created with subscript value 3, has 4 storage locations, or 'elements'; one more than the specified size of the dimension.

3.1.2 Assigning Values

Use the LET statement to assign values to and receive values from an element in an array. (Refer to the LET statement in Chapter 5.) The variable and the value assigned can either both be numeric or both be string data types.

To assign a value to an element of an array, locate the array element on the left in the assignment statement; the element (identified by its subscripts) is assigned the value on the right. For example:

```
10 LET TABLE$(6)='apple strudle'
```

Line 10 in the example assigns the value 'apple strudle' to array element TABLE\$(6).

To assign a value to a variable from an array, locate the variable on the left and the array element on the right. The variable is assigned the contents of the array element. For example:

```
20 LET DESSERT$=TABLE$(6)
```

The preceding example assigns the contents of TABLE\$(6) to the variable DESSERT\$.

In line 10 in the example below a two dimensional array is created. Line 20 causes the contents of element (1,0) to be increased by the value 1 each time it is executed.

```
10 DIM A(2,3)
20 A(1,0) = A(1,0) + 1
```

You can think of array A as looking like this. Note that there are three rows and four columns, because arrays are created beginning with zero.

A(0,0)	A(0,1)	A(0,2)	A(0,3)
A(1,0)	A(1,1)	A(1,2)	A(1,3)
A(2,0)	A(2,1)	A(2,2)	A(2,3)

The effect of line 20 above is to add 1 to the value in element (1,0).

Elements of arrays which store numeric values and reside in memory are given values of 0 (or null for string arrays) when the array is created. Element (1,0) of array A will have value 0 when the array is created, and value 1 after line 20 is executed.

line 10 executed

A(1,0)
0

line 20 executed

A(1,0)
1

Arrays usually reside in memory, however the DIM# statement creates an array located in a file, called a virtual array. (See Section 3.4, File Input and Output, in this chapter for more information on virtual array files.)

The advantage to locating an array in a file rather than in memory is that you can store more data on disk than could be stored in memory. However, there are disadvantages. These are treated under the DIM statement in Chapter 5.

3.1.3 Implicit Arrays

This section has only shown arrays created with the DIM statement, though arrays can be created by merely referencing the array name and subscript(s). This is called implicit creation. When an array is created in this fashion, each dimension is automatically created with size of 10 (11 elements, starting from 0). For example:

```
10 NUMBERS (4,3)=250
```

The program line above creates an array called numbers, to store integer data, with two 11-element dimensions (because there are two subscripts). This statement also assigns the integer value 250 to element 4,3 in array numbers.

3.2 FILES

A file is a collection of information stored on disks and diskettes instead of in the random-access memory of the computer. A file can be as long as needed up to the size of the storage medium. A file is referred to by a name assigned when it is created. It can contain different types of information—for instance text, or data for a program to use, or a PRO/BASIC program. Let's look briefly at the way in which data is stored.

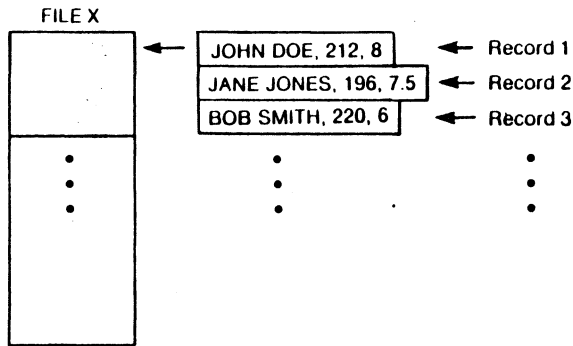
The file is the largest unit of disk storage of information; files contain smaller units of information called records and fields. We will start with the smallest unit, the field.

A field is an indivisible amount of information consisting of a character or a number of characters. A field can be numeric or string and can store data of its own type.

A record is a collection of related fields. An employee record is a typical example. An employee record could consist of such fields as employee name, employee address, social security number, job title, and pay rate.

A file is a group of related records. Each record in a particular file can hold the same amount of information. For example, an employee file would contain one record for each employee of a company.

Think of a file as looking like this:



3.2.1 File Organizations

PRO/BASIC supports two types of files: sequential and virtual array.

- ❑ *Sequential Files*—Records in a sequential file are stored in the order in which they were entered. In order to reach a specific record, all the records that go before it must be read.
- ❑ *Virtual Array Files*—Virtual array files are arrays located on disk. Each virtual array element is one record. Virtual arrays can contain integer, real, or string data. To access the contents of any element of the array just specify the subscript value(s) of the array element you want.

3.2.2 Naming Files

A file needs identification to specify its location and to distinguish it from other files.

The full identification of a file is called a file specification (filespec for short). It is as follows:

device:[directory]filename.type;version

3.2.2.1 Name and Type

A file is identified at a minimum by its file name and type. Only alphabetic (A through Z) and numeric (0 - 9) characters are valid in the file name and type. The DEC Supplemental Graphic characters are not valid in file names.

- ❑ The name identifies the file. The file name can have from one to nine characters.
- ❑ The type identifies the contents of the file. The type appears after the name and is separated from the name by a period. One to three characters may be used.

Below are some examples of PRO/BASIC files:

NEWPROG.BAS

TEST.DAT

MILEAGE.BAS

The two columns below show the P/OS file types, with the equivalent file type identifications used by PRO/BASIC.

PRO/BASIC	P/OS
.BAS	BASIC program
.DAT	Data
.DOC	Document
.TXT	Text

Use the file types listed on the left when in PRO/BASIC. You will use the file types listed on the right with P/OS File Services and other P/OS services. Refer to the *Professional 350 Series User's Guide* for more information.

3.2.2.2 Versions

The version number is used to distinguish between versions of the same file. The highest version number identifies the latest version of file. This number is incremented by 1 and associated with a new file each time a modification is made. Version numbers use the octal number system.

When a file is referenced without a specific version number, the latest version of the file is assumed. Previous versions are kept until deleted.

Include a version number to access a particular version of the file. The version number follows the file type and is separated from the file type by a semicolon (;) or a period (.).

3.2.3 Specifying Directory and Device

When a file is to be input from or output to the current device or directory you need not specify the device name or the directory. PRO/BASIC uses the cur-

rent device, as selected by the user. The default device is the hard disk if the user has not explicitly changed the current directory.

To change the default directory, select "Specify new current directory" from the File Services Menu.

By adding the directory name to the file specification you can access a file on a directory other than the default directory. Some information on directories follows:

On a storage medium, files are grouped by directories. A directory is an index of files. Each diskette has at least one directory. The directory has an entry for each file it contains. The entry contains information on the characteristics and activity of the file. Directory names are assigned by the user. Directory names are from one to nine alphabetic or numeric characters and are enclosed either in square brackets ([]) or in angle brackets (<>).

Add the device name to the file specification to access a file on a device other than the current device. Some information on storage devices follows.

Storage disks and diskettes have a logical name, which is associated with the disk or diskette drives. Volume names are properties of the media (diskette or hard disk) and do not change. Diskettes can be mounted in either diskette drive slot, and therefore the logical name may be different each time you use the diskette. Either the logical name of the device or the volume name of the storage media can be used as the device name in the file specification.

The possible names for the Professional 350 are listed in the table below. The logical name must be followed by a colon (:).

Logical Name	Device Type
DISKETTE1:	RX50 diskette
DISKETTE2:	RX50 diskette
BIGDISK:	RD50 disk

3.3 KEYBOARD INPUT/SCREEN OUTPUT

Keyboard input is data input to the program from the keyboard. Screen output is data from the program that is displayed on the screen.

The PRINT statement is frequently used to write data to the screen. In this section we will see the PRINT statement used to handle string values and to do

arithmetic operations that resolve numeric expressions to their simplest value. We will also look at the characteristics of PRINT when displaying data on the screen.

Data is received from the keyboard with the INPUT and LINPUT statements. We will look at the use of INPUT to assign a single data value to a variable, and the use of LINPUT to assign a line of string data to a variable. We will also introduce the READ and DATA statements, used for storing data in a program.

3.3.1 Displaying Data on the Screen (PRINT statement)

You can use PRINT in immediate mode to display string data on the screen. For example:

```
PRINT "Be Here Now"
Be Here Now
```

PRINT can also be used as a statement in program mode (preceded by a line number). In either case the effect is the same.

The contents of a variable can be displayed by specifying the name of the variable in the PRINT statement. In the following example, the LET statement is used to assign values to string variables S1\$ and S2\$:

```
10 LET S1$ = "Hi"
20 LET S2$ = " There"
30 PRINT S1$,S2$
RUN
'Hi' There
Ready
```

Note in line 10 that the string is enclosed in two matched pairs of quotation marks.

String data must be delimited by quotation marks, either single or double. Data enclosed in quotation marks is printed literally, that is, with no change to the string value. In this example, the outermost set of quotation marks delimit string S1\$; the inner quotation marks are part of the string. In line 20 a blank space appears as the first character in string S2\$ and is printed as part of the string.

A semicolon used between a number of items to be printed causes the items to be printed with no additional spaces between them. (Refer to Section 3.3.2, Formatting with the PRINT statement.)

The PRINT statement can also display numeric data. In the following example, line 10 handles string data, line 20 prints a blank line, and line 30 prints a numeric constant:

```
10 PRINT "Madam I'm Adam"
20 PRINT
30 PRINT 44
RUN
Madam I'm Adam

44
Ready
```

The PRINT statement also resolves a numeric expression to its simplest value and prints only the result. For example:

```
10 PRINT 8*7/4
RUN
14
Ready
```

In the next example, the LET statement assigns numeric values to two numeric variables and the PRINT statement performs the addition operation indicated in line 30. Notice that the commas used to separate the items to be printed cause additional spaces to print between the items.

```
10 LET N1 = 25
20 LET N2 = 75
30 PRINT N1,N2,N1 + N2
RUN
25      75      100
Ready
```

3.3.2 Formatting with the PRINT Statement

3.3.2.1 Print Format

A line on a display screen consists of print zones that are each 14 columns wide. The PRINT statement can use commas or semicolons to separate items on the list to be printed; each has different properties with regard to print zones. A comma causes a data item to print at the beginning of the next print zone. A semicolon used to separate data items will not print any extra spaces between them.

The example below shows the effect of each of the separators between data items to be printed:

```
10 PRINT 'cat','dog'
20 PRINT 'cat';'dog'
RUN
cat          dog
catdog
Ready
```

Numbers separated with a semicolon (;) are printed with a space before and after the number. If the number is negative, a minus sign (–) takes the place of the space before the number.

PRO/BASIC uses exponential notation to represent numeric values greater than 6 digits long. For example:

```
PRINT 74*3.5
.348587E+07
```

Trailing zeros to the right of the decimal point in a number are not printed.

```
PRINT 57.00
57
```

3.3.2.2 *Print Margins*

When printing numeric or string data, PRO/BASIC displays data together on a line when possible, but starts a new line if a line of data on the screen is longer than 80 characters, or longer than 132 characters for file output.

PRO/BASIC tests to see how many spaces remain on a line and prints the remaining data if it will fit, or goes on to the next line if it will not fit. The example below shows that two lines of 42 characters will not fit within the 80 columns of the display screen:

```
10 A$="HERE'S A CHARACTER STRING OF 42 CHARACTERS"
20 PRINT A$;A$
30 B$=" HERE'S ONE WITH FEWER CHARACTERS"
40 PRINT B$;B$
RUN
HERE'S A CHARACTER STRING OF 42 CHARACTERS
HERE'S A CHARACTER STRING OF 42 CHARACTERS
HERE'S ONE WITH FEWER CHARACTERS HERE'S ONE WITH FEWER CHARACTERS
Ready
```

3.3.2.3 PRINT USING

The PRINT USING statement is a variation of PRINT. This statement gives you more control over the appearance and location of data on the output line. It can be used for printing both numeric and string data. When printing numbers it is possible to truncate and round values, to locate a decimal point, and to specify special symbols, for instance floating-dollar signs or asterisk-filled fields, trailing minus signs, and commas.

3.3.3 Receiving Data from the Keyboard

3.3.3.1 INPUT Statement

You can assign data to a PRO/BASIC program using the INPUT statement. The INPUT statement assigns data typed at the keyboard to a variable. When this statement is executed, it asks that a value be assigned to the variable it specifies. For example when the statement below is executed it will ask for a value to be assigned to the variable B:

```
10 INPUT B
```

PRO/BASIC uses a question mark (?) to indicate that it is waiting for a value to be entered. The question mark is called a 'prompt'. When this statement is executed, it prompts for a value to be entered as shown below:

```
10 INPUT B
RUN
?
```

You respond by entering a number because B is a numeric variable. Variables and the data types of the values assigned to them must always agree.

If you make a mistake in typing when responding to an INPUT or LINPUT statement's prompt, use the DELETE key to erase any incorrect characters typed before you press RETURN to enter the response.

The value assigned to B can be used by referencing B, and can be changed at any time by assigning a new value to B.

The following example demonstrates the action of the LET and INPUT and the PRINT statements.


```

10 LET A=5
20 INPUT B
30 PRINT A*B
RUN
? 2
10
Ready

```

In line 10, the value 5 is stored in A. In this program the value of A will always be the same. In line 20 the value input to the variable B is received from the terminal when the user responds to the prompt (?). The PRINT statement in line 30 multiplies the values of A and B and prints the result.

The INPUT statement can accept values for more than one variable at a time. To do this, list any variables you want to input values for after the INPUT statement. Separate each variable with a comma. When an INPUT statement such as this is executed, you must provide a value for each variable. Be certain that values provided are of the same data type as the variables they are assigned to.

The following example shows an INPUT statement with a list of variables of string and numeric data types to receive values. PRO/BASIC will accept values in two ways: each value can be entered after an individual prompt, or all values, each separated by a comma, may be entered to a single prompt. The following example shows the same program line with the responses made in the two ways.

10 INPUT A\$,B,C\$,D,E\$	10 INPUT A\$,B,C\$,D,E\$
RUN	RUN
? hippopotami	? hippopotami,66,apple pie,12,hippopotamus
? 66	Ready
? apple pie	
? 12	
? hippotamus	
Ready	

In both cases, the first value is stored in the string variable A\$, the second value is stored in the numeric variable B, and so on. The commas in the second example serve only to separate each data item from the other.

3.3.3.2 LINPUT Statement

You can also move data to a PRO/BASIC program with the LINPUT statement. The LINPUT statement is much like the INPUT statement, except that LINPUT is used exclusively for string data.

LINPUT accepts and stores all characters including quotation marks and commas, up to the RETURN that enters the line. An example of the LINPUT statement follows:

```
10 LINPUT EMP_ADDR$
20 PRINT EMP_ADDR$
RUN
? 56 Your street, Anytown USA
56 Your street, Anytown USA
Ready
```

In this example, commas do not separate data items: the comma is part of the string.

3.3.3.3 Storing Data in the Program (READ and DATA)

Another way to get data into a program is through the READ and DATA statements. Using this pair of statements, you can enter at one time data that is to be frequently used. They do the same work as the LET statement, but make for better organization when handling many values. Data is written in as part of the program; execution does not stop to receive data.

Data is stored in the DATA statements until it is moved to variables. The READ statement assigns the data to the variable(s) during program execution.

Consider the example below:

```
10 DATA 1,3,5,7,9
20 PRINT, 'radius', 'area'
30 READ R
40 A=PI*R 2
50 PRINT ,R,A
60 GOTO 30
RUN
radius    area
1          3.14159
3          28.2743
5          78.5398
7          153.983
9          254.469
```

Error 57 at line 30: End of Data

The RESTORE statement allows data to be reread by the READ statement. (Refer to the RESTORE statement in Chapter 5.)

3.4 FILE INPUT AND OUTPUT

File input and output is the transfer of data between a file and the PRO/BASIC program.

In this section we will look at the statements used to perform operations on files: the OPEN and CLOSE statements. We will look at the statements used in reading data from and writing data to sequential files, the PRINT #, INPUT #, and LINPUT # statements, and the use of the assignment (LET) statement to read and write data to and from virtual array files.

3.4.1 Opening and Closing Files (OPEN and CLOSE)

These two operations are similar for sequential and virtual array files.

3.4.1.1 OPEN Statement

The OPEN statement can either access an existing file or create a new one. It also associates the file with a channel number that is used to access the file. An example of the OPEN statement follows:

```
10 OPEN "PAYROLL.DAT" FOR INPUT AS FILE #1
```

The purpose of each portion of the OPEN statement is summarized below:

❑ **OPEN "PAYROLL.DAT"**

This portion of the OPEN statement identifies the file to be opened. The file specification must be in enclosed in quotation marks or be a string expression.

❑ **FOR INPUT**

If you specify FOR INPUT, PRO/BASIC opens an existing file. If the file does not exist, the OPEN operation fails and an error message is displayed.

You can also specify FOR OUTPUT. If you do, PRO/BASIC creates a new version of the file, whether a version currently exists or not.

If neither FOR INPUT nor FOR OUTPUT is specified, PRO/BASIC searches for a file with the specified name. If there is a file by that name, it is opened for access. If no file by that name exists, PRO/BASIC creates a new file.

❑ **AS FILE #1**

The AS FILE portion uses a number, called a channel number to identify the file for input/output operations. PRO/BASIC establishes #channelnum as equivalent to the file specification. The channel num-

ber can be a number from 1 to 15. A file may be open on one channel only, but a number of files can be open (one file per channel) at the same time.

3.4.1.2 CLOSE Statement

The CLOSE statement closes the specified file and disassociates it from its channel number. The CLOSE statement looks like this:

```
CLOSE #1
```

Use the CLOSE statement to close all open files when finished transferring data in order to recover memory. If you do not do it explicitly, all open files are automatically closed by a subsequent RUN statement or when you exit PRO/BASIC.

3.4.2 Writing to a Sequential File (PRINT #)

You use sequential files in much the same way that you do keyboard input and screen output. Just as the PRINT statement displays data on the screen, PRINT # writes data to the file. The following is an example of the PRINT # statement:

```
PRINT #1,EMP_NAME$
```

The expression after the # (number sign) is the channel number. The channel number must be the same value as the expression specified in an OPEN statement.

Any numeric or string value or numeric or string variable to be written to the file can be included after the channel number. A comma separates the print items from the channel number, and a comma or semicolon is used to separate print items.

3.4.3 Reading from a Sequential File (INPUT #,LINPUT #)

The INPUT # statement retrieves data from a file and places that data in specified variables. Just as the INPUT statement requests a value from the keyboard, INPUT # requests a value from a sequential file. Once the data is in the variables, it can be operated upon. The following is an example of the INPUT # statement:

```
INPUT #1,SSN
```

The expression after the # (number sign) is the channel number. The channel number must be the same value as the expression specified in an OPEN statement.

If you specify the channel number of an opened file, PRO/BASIC reads data from the file. If you specify 0, PRO/BASIC reads data from the keyboard. (When you specify 0, the question mark prompt (?) is overridden, so you should provide your own.) List variables after the channel number. A comma must separate the channel number from any variables listed; and when more than one variable is listed, each variable must be separated by a comma. For an example of the use of PRINT # and INPUT # with sequential files, refer to the first example in Section 3.8, Chaining.

When using PRINT # and INPUT # to write data to and read data from sequential files, be certain that: the variables are of the same data type as the file data, and that the INPUT # statement uses no more variables than there are data items present in the record. Also be sure that if INPUT # is used to read file data, the data items written to the file with the PRINT # statement are each separated by commas enclosed in quotes so that the data items will be properly separated when input.

The following program lines from two programs that access the same file illustrate these points:

.	.
.	.
100 PRINT #1, ITEM_1,',':ITEM_2%	100 INPUT #1, A, B%
.	.
.	.

In this example, both lines use the same number of variables of the same data type, and the data items written to the file in the PRINT # statement are separated by commas.

The LINPUT # statement inputs an entire line from a file as a string. PRO/BASIC treats LINPUT # just as it treats LINPUT; all characters on the input line, including commas and quotation marks, are assigned to the string. The following is an example of the LINPUT # statement:

```
100 LINPUT #1,EMPL_NAME$
```

3.4.4 Appending Records to a File

When handling sequential files, you can read records from the file until the end of the file is reached, and then you can write more records onto the bottom of the file. This is called appending.

A good method of appending records to the end of the file is to use an error-handling routine. When end-of-file is detected, control can pass to a routine that

writes new records to the file. (See Section 3.5, Error-handling, for an example of such an error-handling routine.)

If after appending records to the end of a file, you need to return to the top of the file, close the file, and then open it again. This will locate program control at the first record in the file.

3.4.5 Creating a Virtual Array File (DIM # Statement)

To create a file containing a virtual array, place a DIM # statement and an OPEN statement with the same channel in a program.

There is one significant difference in the use of the OPEN statement for virtual array files: the use of the VIRTUAL option. Include the optional VIRTUAL clause after the channel number to create the array in a file rather than in memory. In the following example the DIM # statement creates an array and the OPEN statement, with the VIRTUAL option, associates the array with a file:

```
10 DIM #1,A(20,20)
20 OPEN 'TABLE.DAT' FOR INPUT AS FILE #1,VIRTUAL
```

If no file type is provided in the name of the virtual array file, the default file type .DAT is used.

After the file is opened, the elements of the virtual array can be used in the same way as elements of an array in memory.

3.4.6 Writing to and Reading from a Virtual Array File

You access virtual array files the way you access arrays in memory: use the LET statement to assign information (write) to and reference information (read) from the virtual array file. (Refer to Section 3.1 Arrays.)

To assign a value to an element of a virtual array file, locate the element on the left in the assignment statement; the element (identified by its subscripts) is assigned the value on the right. For example:

```
100 A(5,7) = HAT_SIZE
```

To assign a value to a variable from a virtual array element, locate the variable on the left in the assignment statement; the variable is assigned the value of the virtual array element on the right. For example:

```
100 EMP_NUM = A(20,1)
```

3.5 PROGRAM ERRORS AND ERROR HANDLING

During execution of a program some things may happen that are not anticipated by the program. For example, the end of a file may be reached so a program has no more data to work on, or an invalid mathematical operation may occur. Some of these errors—for instance the end-of-file condition—can be detected by the system.

When the system detects a program error it displays a message about the error. Depending on the severity of the error, PRO/BASIC may display a warning message and continue processing, or display a fatal error message and halt the program's execution. When an error occurs, you can press the **HELP** key to display more information on the error.

Other errors may not be discerned by the system until later, or perhaps not at all—for example, if a value entered is of the correct data type but is outside the valid range.

The **ON ERROR GOTO** statement allows you to include statements in the program that catch errors before the system does and then transfer program control to recover from the error or correct it before proceeding. The **ON ERROR GOTO** statement looks like this:

```
100 ON ERROR GOTO 1000
```

When an error occurs after line 100, the **ON ERROR** statement directs program control to the specified line, line 1000. It also places the error's number, and the line the error occurred at, in two special variables.

At the specified line number there are statements to test the contents of the variable that contains the error code number (**ERR**) and the line that was executing when the error occurred (**ERL**). Program control is then transferred accordingly.

You can specify a line number in the **ON ERROR** statement to handle errors within the program, or you can specify 0 to allow PRO/BASIC to handle errors.

The example below opens file **JUNK.DAT**, prints each record on the display screen, and—when the end of the file is reached—goes to the error handler. The error handler is simply the **RESUME** statement, which transfers control to a loop to input new records to the bottom of the file. If you wish to run the example program below, use an immediate mode statement to create the file **'JUNK.DAT'**:

```

OPEN 'JUNK.DAT' AS FILE #1

10 ON ERROR GOTO 1000
20 OPEN 'JUNK.DAT' FOR INPUT AS FILE #1
30 LINPUT #1,P$
40 PRINT 'OLD RECORD:';P$
50 GOTO 30
100 PRINT 'Now type in new lines to append to the file'
105 PRINT 'Press RETURN to stop'
106 PRINT 'NEW RECORD:'
110 LINPUT #0,P$ \IF LEN(P$)=0 THEN 200
120 PRINT #1,P$
130 GOTO 106
200 CLOSE #1
210 STOP
1000 REM Error handler
1010 IF ERL=30 THEN RESUME 100
RUN
Now type in new lines to append to the file
Press RETURN to stop
NEW RECORD:first line in file
NEW RECORD:RETURN
STOP at line 210
RUN
OLD RECORD:first line in file
Now type in new lines to append to the file
Press RETURN to stop
NEW RECORD:now add another record to the file
NEW RECORD:RETURN
STOP at line 210

```

For simplicity of discussion all exceptional conditions encountered in processing are termed “error”; some are errors and some are not. For example, an end-of-file error is not really an error, it indicates no faulty processing. Error-handling routines, which can respond to end-of-file and other exceptional conditions, make the term error still less accurate. The term “error” is only appropriate in a general sense and should not be construed to mean “mistake.”

3.6 PROGRAM CONTROL

In a PRO/BASIC program, control ordinarily passes from statement to statement and from line number to line number in ascending order. The normal order of execution can be altered to:

- ❑ Repeat a group of statements.
- ❑ Continue processing elsewhere in the program.
- ❑ Terminate the program.

The following sections describe the statements that affect program control.

3.6.1 Unconditional Transfer (GOTO)

The GOTO statement is the simplest method of changing the normal lowest-to-highest line number execution of a program. Control may be directed to any existing line number.

When a GOTO is executed, program control branches to the specified line number. In the example below, program control gets to line 40, and then goes to line 10: this process continues until the READ statement in line 10 reads all the values in the DATA statement.

```

10 READ A$
20 PRINT A$
30 DATA a, little, example
40 GOTO 10
RUN
a
little
example

```

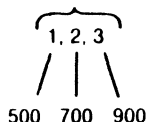
Error 57 at line 10: End of Data

A GOTO should be the only or last statement on a line; any statements appearing after the GOTO will not be executed. Transfer of control is to the first statement on the specified line.

3.6.2 Multiple Branching (ON GOTO)

ON GOTO is a more powerful form of the GOTO statement. ON GOTO allows you to specify several line numbers as alternatives to branch to. Which line is chosen depends on the value in a variable or the result of an expression. An example of the ON GOTO statement follows:

```
100 ON CODE GOTO 500,700,900
```



The expression can result in as many values as there are line numbers listed. In the preceding example, CODE is a variable that can validly contain values 1,2, or 3 because there are three line numbers listed.

When PRO/BASIC executes an ON GOTO statement it first evaluates the numeric expression. The value is then truncated to integer if necessary. If the value is equal to 1, PRO/BASIC passes control to the first line number in the list; if the value is equal to 2, PRO/BASIC passes control to the second line number in the list, and so on. Consider the following:

```
500 ON A GOTO 100,200,400,300
```

if value in A equals:	PRO/BASIC transfers control to:
less than 1	causes an error
1	line 100
2	line 200
3	line 400
4	line 300
greater than 4	causes an error

If the value is less than 1, or greater than the number of line numbers listed, the error message, "Expression in ON statement out of bounds" is displayed. An error message is also displayed if a line number specified in the ON GOTO statement does not exist.

3.6.3 Conditional Transfer (IF)

The IF statement tests a conditional expression and executes a statement or series of statements if the expression is true. If the conditional expression is false, the statement or statements are not executed. For example:

```
10 PRINT 'ENTER A VALUE LESS THAN 6'
20 INPUT N
30 IF N >= 6 THEN PRINT "WRONG"
.
.
.
```

In the preceding example lines 10 and 20 request and assign a value to N. In line 30, if the value of N is equal to or greater than 6, the expression is true, and the message WRONG is displayed. If the value of variable N is less than 6, the expression is false and processing will continue after line 30.

The IF statement also tests relational expressions. Following are two examples of the IF statement:

```

10 INPUT A,B
20 IF A<>B THEN PRINT 'A is not equal to B'
RUN
? 3,4
A is not equal to B
Ready

10 INPUT A,B
20 IF A<>B THEN PRINT 'A<>B'\PRINT 'A=';A, 'B=';B
RUN
? 3,4
A<>B
A = 3          B = 4

```

The IF statement can be followed with an ELSE clause, which is executed only when the conditional expression is false. The word ELSE can be followed by either a line number or statement(s). For example:

```

10 INPUT A,B
20 IF A<>B THEN PRINT 'A<>B' ELSE PRINT 'A=B'
RUN
? 3,4
A<>B
Ready
RUN
? 4,4
A = B

```

Note that IF THEN ELSE is all one statement. The THEN clause and optional ELSE clause can only be used with the IF and must appear on the same line as the IF.

Logical operations can be performed with IF statements. The following example uses an IF statement in line 30 to perform a logical operation on a value entered from the keyboard. If the number is less than 1 or greater than 2, a message is displayed, the prompt is repeated, and the IF statement transfers program control to line 10.

If the value is within the valid range the IF statement in line 40 does a relational test on it and then transfers program control to one of two lines.

```

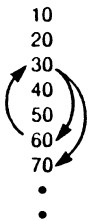
10 PRINT 'ENTER 1 OR 2'
20 INPUT A
30 IF A<1 OR A>2 THEN PRINT 'Value must be either 1 or 2, reenter?'\GOTO 10
40 IF A=1 GOTO 300 ELSE GOTO 400
.
.
.

```

3.6.4 Loops (FOR and NEXT)

If it were not possible to return to a section of your program you would have to rewrite the statements of that portion of the program each time the same operation was needed. The technique of returning to the same statements for repeated execution is called looping.

The loop below repeatedly executes the statements between lines 30 and 60 until a condition is met and program control goes to line 70.



PRO/BASIC provides the FOR and NEXT statements for constructing loops. A loop can also be created with a GOTO statement and an IF statement. As an introduction to loops made with the FOR and NEXT statements, we will look first at loops you make yourself.

3.6.4.1 How a Loop Works

We'll look at a loop constructed of familiar components that you can handle yourself, without the FOR/NEXT statements.

```

10 LET COUNTER = 1                (assigns beginning value to variable)
20 IF COUNTER>=10 GOTO 99          (establishes ending value)
30 PRINT 'EXECUTION ';COUNTER;' OF THE LOOP' (statement to execute)
40 LET COUNTER = COUNTER+1         (increments counter)
50 GOTO 20
99 END

```

When this program is executed, line 20 checks **COUNTER** to see if it is equal to 10 or greater. If it is not, the **PRINT** statement is executed. In line 40 the value 1 is added to **COUNTER**. Line 50 transfers control back to line 20, where **COUNTER** is tested again. When the value of **COUNTER** equals 10 or more, control is transferred to line 99, and the program ends. To make a loop the following are needed:

- ❑ A counter to keep track of the number of executions of the loop
- ❑ A value that is the starting point for the counter
- ❑ The ending value, that is, the value of the counter that will stop execution

Loops are so frequently used that **PRO/BASIC** has a pair of statements which simplify the process.

3.6.4.2 Using **FOR** and **NEXT**

The **FOR/NEXT** statements handle the details of constructing a loop. These statements must be used together. The **FOR** statement defines the beginning of the loop, the **NEXT** statement defines the end of the loop. Anything appearing between the **FOR** and the **NEXT** is executed at each pass through the loop.

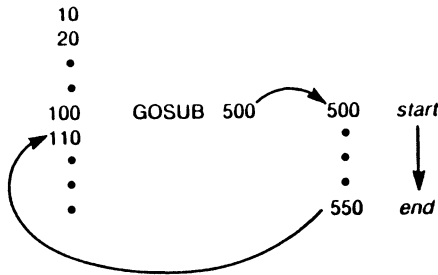
A counter variable is specified in the **FOR** statement which is incremented by the **NEXT** statement at each pass through the loop. The following example shows two loops: the loop on the left uses an **IF** statement and a **GOTO** statement; the loop on the right uses the **FOR** and **NEXT** statements.

10 I=1	10 FOR I = 1 TO 10
20 PRINT I	20 PRINT I
30 I=I+1	30 NEXT I
40 IF I<=10 THEN 20	40 END
50 END	

(Refer to the **FOR** and **NEXT** statements in Chapter 5 for more information and examples.)

3.6.5 Subroutines (**GOSUB**, **ON GOSUB**, **RETURN**)

A subroutine is a section of statements that does one thing whenever needed. In many programs there are sections of statements which do the same thing or nearly the same thing a number of times. To use subroutines can save the need to use unique program statements each time an execution is done. Subroutines in a program help organize the program and decrease the amount of memory required. Programs containing subroutines can be thought of like this:



The following statements are used when using subroutines:

- The GOSUB statement contains the beginning line number of a subroutine and, when executed, transfers program control to that line. The GOSUB statement also records the location from which control was transferred. The statements of the subroutine are then executed until the RETURN statement at the end of the subroutine is reached. The GOSUB statement looks like this:

GOSUB 1000

- The ON GOSUB statement transfers program control to one of a number of subroutines (referenced by their first line numbers), depending upon the value of a numeric expression. The GOSUB statement also records the location from which control was transferred. The ON GOSUB statement looks like this:

ON CLASS GOSUB 500,1000,1500

- A RETURN statement appears after the last program statement of all subroutines. Its purpose is to transfer program control back to the statement after the particular GOSUB or ON GOSUB statement last executed. (That location was recorded by the GOSUB or ON GOSUB statement.) The RETURN statement looks like this:

RETURN

A small example of a program with a subroutine follows.

```

10 INPUT A,B,C
20 GOSUB 40
30 PRINT D
35 GOTO 70
40 REM THIS IS A SUBROUTINE
50 D=A*B-C
60 RETURN
70 END
RUN
? 5,10,15
35
Ready

```

3.7 HALTING PROGRAM EXECUTION

Four methods to halt program execution are the following:

- ☐ Using the END statement
- ☐ Execution of the program through its highest line number
- ☐ Using the STOP statement
- ☐ INTERRUPT/DO

The END statement is optional. If you include an END statement, it should be the last statement in the program. Transferring control to an END statement with a GOTO or an IF statement terminates program execution.

The STOP statement causes PRO/BASIC to halt and print a message:

STOP at line n

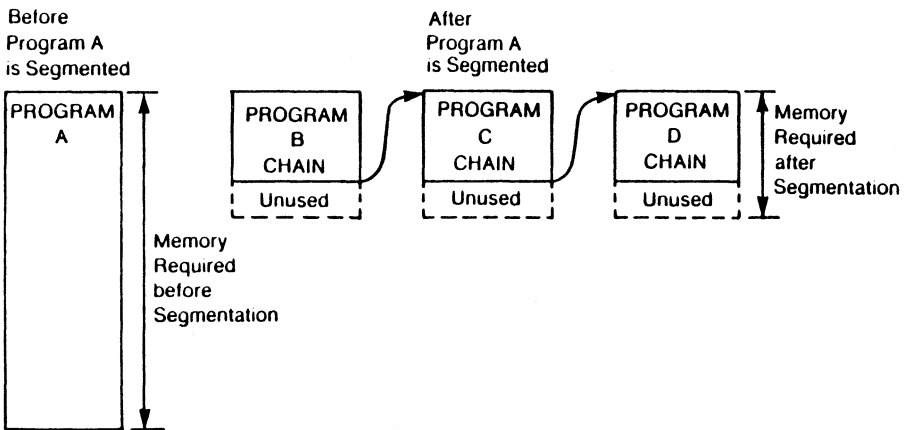
where *n* is the line number location of the STOP statement. STOP can be useful when looking for problems in your program. By placing one or more STOP statements throughout your program you can check the contents of variables at any intermediate point in processing. Continue execution with the CONTINUE statement or an immediate mode GOTO statement.

The STOP statement does not close files. Use the END statement to close files. If the program does not execute an END statement, the execution of the last statement of the program ends the program and closes all open files.

To use INTERRUPT/DO press the INTERRUPT key and the DO key. INTERRUPT/DO terminates the currently executing program and returns to PRO/BASIC.

3.8 CHAINING

Chaining, or program segmentation is the process of breaking a large program into two or more smaller programs. A large program, when made into smaller programs, will use less space in memory as shown below:



After program A is segmented into smaller programs, only program B is in memory at the start of execution. After program B runs, program B chains to program C, that is, PRO/BASIC erases program B, replaces it with program C, and runs program C. After program C runs, program C chains to program D. Less memory is required because smaller programs, each in their turn, occupy memory.

Use the CHAIN statement to segment a program. To use the CHAIN statement, create the program as usual, but include a CHAIN statement with a file specification as the last statement in the program. The CHAIN statement looks like this:

```
200 CHAIN "PRGRM2.BAS"
```

Include a CHAIN statement in each program that is to be followed by another program. When PRO/BASIC executes the CHAIN statement, it closes any open files, erases the program just executed, and transfers the program specified in the CHAIN statement into memory. Consider the following example:

The file specified by 'SEG1' contains:

10 PRINT 'program seg1 is working'	Prints identifying message
20 OPEN 'DATA1' FOR OUTPUT AS FILE #1	Creates output file
30 FOR I=1 TO 100	Writes out all even numbers
40 PRINT #1,2*I	2 to 200
50 NEXT I	to the file
60 CLOSE #1	Closes the file
70 CHAIN 'SEG2'	Chains to the next
80 END	program

The file specified by 'SEG2' contains:

10 PRINT 'program seg2 is working'	Prints identifying message
20 OPEN 'DATA1' FOR INPUT AS FILE #1	Opens existing file
30 FOR I=1 TO 100	Inputs the numbers
40 INPUT #1,J	from the file and adds
50 T=T+J	them, storing the total
60 NEXT I	in T.
70 PRINT 'THE TOTAL IS ';T	Prints the total.
80 CLOSE #1	Closes the file.
90 END	

A run of these programs produces the following output:

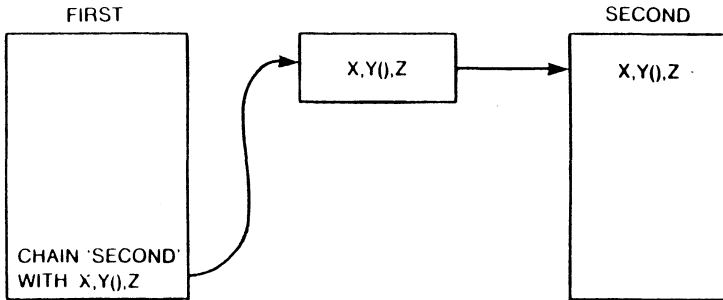
RUN SEG1	Run the first segment
program seg1 is working	Segment 1 executes and
program seg2 is working	chains to Segment 2
THE TOTAL IS 10100	Prints the total.
Ready	

Remember to use the SAVE command to save a program containing a CHAIN statement before running it. If you don't save the program, it is lost when PRO/BASIC transfers the next program into memory.

Using the LINE option in the CHAIN statement, you can begin execution of the chained-to program at any specified line number location. For example:

```
100 CHAIN 'SEG3' LINE 200
```

It is also possible to save values from the chaining program for use by the chained-to program.



At the left program FIRST is in memory. When PRO/BASIC executes the CHAIN statement with the WITH option in FIRST, it erases all of FIRST except the values in the variables X and Z, and the array Y as illustrated in the center. At the right, PRO/BASIC brings program SECOND into memory and assigns the preserved values to variables X and Z and array Y in program SECOND.

Use the WITH option of the CHAIN statement to preserve the values of variables and arrays. The WITH option of the CHAIN statement looks like this:

```
100 CHAIN 'SEG3' WITH S_TOTAL,TOTAL
```

The PROGRAM statement is used in the chained-to program if the WITH option is used in the CHAIN statement to pass values from the chaining program. The PROGRAM statement looks like this:

```
10 PROGRAM SEG3 (A$,B,C)
```

Consider the following example:

The file specified by 'PROG1' contains:

```

10 C=9
20 DIM A(C),B(C)
30 PRINT 'EXECUTING PROG1'
40 FOR D=0 TO C
50 A(D)=D*2
60 B(D)=D*3
70 TOTAL_A=TOTAL_A+A(D)
80 TOTAL_B=TOTAL_B+B(D)
90 NEXT D
100 PRINT C,TOTAL_A,TOTAL_B
110 CHAIN 'PROG2' WITH C,A()
120 END
    
```

Assigns 9 to C
 C used in DIM statements
 Prints identifying message

 Multiplies value in D and 2
 and 3
 stores D*2 in TOTAL_A
 stores D*3 in TOTAL_B
 Does loop nine times
 Prints C,TOTAL_A,TOTAL_B
 Passes variable C, array A

The file specified by 'PROG2' contains:

10 PROGRAM PROG2(SIZE%,TABLE())	Variable C renamed to SIZE%,
20 REM SIZE% now DIMs TABLE	Array A renamed to TABLE
30 PRINT 'EXECUTING PROG2'	Prints identifying message
40 FOR X=0 TO SIZE%	
50 T1=T1+TABLE(X)	Sums values in TABLE
60 T2=T2+B(X)	(passed from PROG1) and in new array B
70 NEXT X	(original B not passed)
80 PRINT SIZE%,T1,T2	Prints SIZE%,T1,T2
90 END	

A run of these two programs produces the following:

```

RUN
EXECUTING PROG1
  9                90                135
EXECUTING PROG2
  9                90                0
Ready

```

3.9 USING CONTROL FUNCTIONS

Control functions are non-display single- or multiple-character codes that you can use to give direct instructions to your Professional computer. The single character codes are the control characters that appear in columns 0, 1, 8, and 9 of the DEC Multinational code table. The multiple-character codes are either escape sequences or control sequences. Escape sequences are unique sequences of ASCII characters preceded by the ESC control character (decimal code 27). Control sequences are unique sequences of ASCII characters preceded by the CSI control character (decimal code 155). Note that PRO/BASIC uses 8-bit mode. For more information on character codes refer to the *Terminal Subsystem Manual*.

All control functions and their use are explained in detail in the *Professional 300 Series Terminal Subsystem Manual*. Two examples are given below to show how to enter control functions in a PRO/BASIC program.

3.9.1 Control Character

The following program demonstrates the use of the “Bell” control character (BEL, decimal code 7). The program asks you for a number from 1 to 10. If the number is invalid, the keyboard beeps (“bell” sound) three times and asks for more input. Note the use of CHR\$ (line 40) to return the character associated with the numeric code 7. This is so the BEL character can be printed in lines 140–160.

```

10 REM demonstrates use of Bell control character
20 REM
30 REM Define B$ to be the BEL control character
40 LET B$=(CHR$(7))
50 REM
60 PRINT "Type a number from 1 to 10"
70 INPUT NUMBER
80 IF NUMBER<1 GOTO 140
90 IF NUMBER>10 GOTO 140
100 PRINT "Number entered (";NUMBER;") is valid"
110 GOTO 180
120 REM
130 REM "Print" the BEL character 3 times to ring the bell 3 times.
140 PRINT B$
150 PRINT B$
160 PRINT B$
170 GOTO 60
180 END

```

3.9.2 Escape Sequence

The following program demonstrates the use of an escape sequence to print text at a specified line and column of the screen. The escape sequence used is

ESC [P1 ; Pc H

It places the cursor (where the next character will appear) at line P1 and column Pc. P1 and Pc are variable parameters. All characters shown are part of the sequence, including the semicolon, and are interpreted as ASCII codes.

In line 70 of the program, the user-defined function FNAT\$ is used to represent the escape sequence. FNAT\$ places the cursor at line X and column Y (X and Y are temporary variables). The library function EDIT\$, with the parameter 2, removes any spaces embedded in the sequence. NUM\$, which expresses a number as a string, is used because EDIT\$ operates only on strings.

As with control characters, you use a PRINT statement to enter the escape sequence. The first part of each PRINT statement on lines 90 and 100 places the cursor at the specified position. The second part prints the message.

```
10 REM Program demonstrates use of an escape sequence to print text
20 REM at a specified line and column of the screen.
30 REM
40 REM first clear the screen
50 CLEAR
60 REM
70 DEF FNAT$(X,Y)=EDIT$(CHR$(27)+'['+NUM$(X)+';' +NUM$(Y)+'H',2)
80 REM
90 PRINT FNAT$(0,0); 'hello there!'
100 PRINT FNAT$(12,8); 'Where are the snows of yesteryear?'
110 PRINT
120 PRINT
130 PRINT
140 END
```

4

Commands

Chapter 4

Commands

This chapter lists the commands of PRO/BASIC. Commands generally perform an action on a program, for example, start execution, move or copy a program, make a change to a program, or display information on a program.

The presentation of each command consists of four parts:

- ☐ *Syntax* – the syntax required by the command
- ☐ *Purpose* – what affect the command can have on the program
- ☐ *Comments* – explanations and suggestions for use
- ☐ *Example* – typical or explanatory example

CATALOG

Syntax

CAT[ALOG] [filespec]

where:

filespec is a file specification or a directory name.

Examples of Syntax

```
CAT
CAT*.BAS
```

Purpose

CATALOG displays the files in a directory.

Comments

There are a number of different methods of using CATALOG:

- ❑ Omit filespec to display all files in the current directory.
- ❑ Include a directory name to display files in a directory other than the present directory. For example:

```
CAT <MEMOS>
```

- ❑ Use the wildcard character, denoted by an asterisk (*). The wildcard character used in a portion of the file specification (file name, extension, or version number) means “all”. The wildcard allows the CATALOG command to display any file specification that matches the information provided, while ignoring the element(s) replaced by the wildcard.

For instance, a wildcard in the file name will display any file specification that matches the file type named. A wildcard in the file type will display all file specifications that match the specified file name.

The example below uses a wildcard instead of the file name, and specifies the file type as .BAS to display all files with the extension .BAS;

```
CAT *.BAS
AP.BAS;1      GL.BAS;5      PROG1.BAS;13    PROG2.BAS;2
```

The example below substitutes a wildcard for a specific file type, and specifies PAYROLL as the file name, to display any files with file name PAYROLL, without regard to their types.

CAT PAYROLL.*

PAYROLL.BAS;3 PAYROLL.DAT;5 PAYROLL.DAT;4 PAYROLL.DAT;3

You can also specify a portion of a file name—for instance, PROG*.BAS to display any file specification where the file name has characters in common with characters appearing before the wildcard.

- ☐ PRO/BASIC assumes a wildcard for the version number unless a version is specified.
- ☐ CAT *.* is the same as CAT.

Example

CAT

AP.BAS;6	AP.BAS;11	AP.BAS;24	AR.BAS;12
AR.BAS;14	AR.BAS;15	CHECK.BAS;24	EXP.BAS;12
EXP.BAS;12	EXPENSE.DAT;11	EXPENSE.DAT;12	GLEDGER.BAS;2
GLEDGER.BAS;3	GLEDGER.DAT;17	GLEDGER.DAT;15	INVNTRY.BAS;15
INVNTRY.BAS;3	INVNTORY.DAT;2	JUN82.DAT;2	JUN82.DAT;3
MAY82.DAT;1	MEMO.DAT;3	REPORT.BAS;4	T.DAT;14

Ready

CONTINUE

Syntax

CONT[INUE]

Purpose

CONTINUE causes PRO/BASIC to resume execution of a program halted by a STOP statement or a STEP command.

Comments

- If execution was halted by a STOP statement or after a STEP command, the CONTINUE command transfers control to the statement immediately following the STOP or STEP. If execution was halted by pressing INTERRUPT, then DO, the CONTINUE command re-executes the interrupted statement.
- The CONTINUE command cannot be used if you have changed your program since the last time it was run. You can however, resume execution with an immediate mode GOTO statement.

Example

```

05 STOP
10 REM Demonstrate CONTINUE command
20 A = 15 \ B = 150
30 X = A/B
40 STOP
50 PRINT X
60 END
RUN

STOP at line 5
CONT

STOP at line 40
DELETE 5
CONT

Error 114 at line 40: Can not continue

GOTO 50
.1
Ready

```

DELETE

Syntax

```

[linenum          ]
DEL[ETE] [linenum1 – linenum2 ]
          [linenum1 –          ]
          [ – linenum2          ]

```

where:

linenum is a single line number

linenum1–linenum2 specifies all lines from line number 1 to line number 2, inclusive.

linenum – specifies all lines from line number to the end of the program.

– linenum specifies all lines from the beginning of the program to line number.

Example of Syntax

```
DEL 10
```

Purpose

DELETE erases one or more lines from the current program.

Comments

- ☐ You must specify a line number or range of lines.
- ☐ Not all program lines must exist within the range specified.

Example

```
DEL – 100
```

```
DEL 100–350
```

EDIT

Syntax

EDIT linenum

where:

linenum is the number of a line in your program that you wish to change.

Example of Syntax

EDIT 150

Purpose

EDIT allows you to modify your program by changing individual lines using the Line Editor.

Comments

- ☐ When you have finished editing your input line, press RETURN or ENTER. PRO/BASIC either accepts the line or, if invalid, prints an error message and reinvokes the editor.
- ☐ See Chapter 1 for more information on the Line Editor.

EXIT

Syntax

EXIT

Purpose

EXIT terminates PRO/BASIC and returns control to the Main Menu.

Comments

- ❑ If changes have been made to the program in memory and you type EXIT to leave PRO/BASIC, the following warning message is displayed:

Error 109: Unsaved changes, type EXIT again to exit.

- ❑ The message indicates that you should use the SAVE command if you wish to preserve the latest version of your program before you leave PRO/BASIC.
- ❑ If you do not want to save the latest version of the program, press EXIT to leave PRO/BASIC.
- ❑ Pressing EXIT has the same effect as the EXIT command.

LIST

Syntax

```

        [linenum          ]
LIST [linenum1 – linenum 2]
        [linenum –        ]
        [ – linenum       ]

```

where:

linenum	is a single line number.
linenum1–linenum2	specifies all lines from line number 1 to line number 2, inclusive.
linenum –	specifies all lines from line number to the end of the program.
– linenum	specifies all lines from the beginning of the program to line number.

Examples of Syntax

```

LIST 100
LIST 220-290

```

Purpose

LIST displays the program currently in memory.

Comments

- If you do not include a line number or range, PRO/BASIC lists your entire program preceded by a line showing the program name, the date, and the current time.
- Press INTERRUPT, then DO, or HOLDSCREEN to stop listing a long program.
- All lines specified within a range do not have to exist.

MERGE

Syntax

MERGE [filespec]

Examples of Syntax

MERGE MYFILE

MERGE Diskette1:SEG22

Purpose

MERGE transfers the specified program from disk or diskette and merges it with the program currently in memory.

Comments

- ❑ PRO/BASIC inserts each line (by line number) from the stored program. In the case of duplicate line numbers, PRO/BASIC erases the line in the current program and inserts the line from the specified program.
- ❑ A large program can be segmented and stored in a number of files. To run the program, use the MERGE command to load the needed segments.

When writing program segments to be merged, assign each one a discrete block of line numbers. The merged program will then have all its lines in the correct order regardless of the order in which you merge the segments.

- ❑ Be careful that all unwanted lines are replaced by lines in the specified file, or deleted by using the DELETE command.
- ❑ If you type MERGE with no file specification PRO/BASIC searches for a file named NONAME.BAS. If no file with that name is found an error message is displayed.
- ❑ If there is no program in memory, a MERGE command is equivalent to an OLD command, except that it does not change the current program name.
- ❑ You can use the MERGE command to modify the current program by replacing specific segments. For example, you can quickly replace a subroutine or one block of DATA statements with another (see the following example).

Example

```

200 REM *** DATA1 ***
210 DATA ONE,TWO,THREE

200 REM *** DATA2 ***
210 DATA FOUR
220 DATA FIVE
230 DATA SIX

```

Ready

OLD DEMO

Ready

LIST

```

100 REM *** Demonstrate the MERGE command ***
310 FOR I% = 1% TO 3%
320  READ D$ \ PRINT D$; ' ';
330 NEXT I% \ PRINT \ END

```

Ready

MERGE DATA1

Ready

LIST

```

100 REM *** Demonstrate the MERGE command ***
200 REM *** DATA1 ***
210 DATA ONE,TWO,THREE
310 FOR I% = 1% TO 3%
320  READ D$ \ PRINT D$; ' ';
330 NEXT I% \ PRINT \ END

```

Ready

RUN

ONE TWO THREE

Ready

MERGE DATA2

Ready

LIST

100 REM *** Demonstrate the MERGE command ***

200 REM *** DATA2 ***

210 DATA FOUR

220 DATA FIVE

230 DATA SIX

310 FOR I% = 1% TO 3%

320 READ D\$ \ PRINT D\$;' ';

330 NEXT I% \ PRINT \ END

Ready

RUN

FOUR FIVE SIX

Ready

NEW

Syntax

NEW [programe]

where:

programe is a program name of up to 9 alphabetic and/or numeric characters.

Example of Syntax

NEW MYPROG

Purpose

NEW clears memory, deletes all variables and user-defined function definitions, and saves the name of your new program.

Comments

- ☐ Supply only the program name. Do not supply a directory name, extension, or version number to the **NEW** command.
- ☐ If you later save the program with the **SAVE** command without specifying an extension, the program name you provide is used as the file name and is given the extension **.BAS**.
- ☐ If you do not include a program name, **PRO/BASIC** uses **NONAME** by default.
- ☐ You can always change the name you chose for your new program by using the **RENAME** command, and you can store it with a different name by specifying a filespec with the **SAVE** command.
- ☐ If you want to keep the program currently in memory, enter a **SAVE** command before you use the **NEW** command.

Example

NEW PROG1

OLD

Syntax

OLD [filespec]

Examples of Syntax

OLD MYPROG

OLD LABGEN.BAS;3

Purpose

OLD transfers the specified PRO/BASIC program from disk/diskette to memory.

Comments

- ☐ If you want to keep the program currently in memory enter a SAVE command before you use the OLD command.
- ☐ Before transferring the program, OLD clears memory, deletes all variables and function definitions, and closes all open files.
- ☐ If you do not provide a file specification, PRO/BASIC searches for file specification NONAME.BAS.
- ☐ If the specified file is not found, PRO/BASIC displays an error message and does not clear memory.
- ☐ OLD extracts the file name portion of the file specification provided and assigns it as the program name.
- ☐ If a filespec is included with the RUN command, RUN can perform the same function as the OLD command and also start program execution.

RENAME

Syntax

RENAME progname

where:

progname is a program name of up to nine alphabetic and/or numeric characters.

Example of Syntax

RENAME NEWNAM

Purpose

RENAME changes the name of the program in memory.

Comments

- When the program is saved with the SAVE command, the program name is used as the file name, and the extension .BAS is assigned. Refer to the SAVE command in this chapter for more information on saving files.

Example

RENAME SEG1

RENUMBER

Syntax

```
RENUMBER [linenum[,increment]]
```

where:

linenum	is the first line number of the new line number sequence; the default is 10.
increment	is the increment for the new line number sequence; the default is 10.

Example of Syntax

```
RENUMBER 100,10
```

Purpose

Renumbers program lines.

Comments

- The RENUMBER command organizes a program in a uniform line number sequence, for example after extensive additions or deletions of program lines.
- ERL references are not changed as a result of the RENUMBER command.

Example

This example shows how PRO/BASIC checks for line numbers during a RENUMBER operation.

```

2 REM This program demonstrates errors in a RENUMBER
13 REM
20 GOTO 9582 \ REM Line 5982 does not exist
56 GOT 30 \ REM Line 30 does not exist, but will after a RENUMBER
99 END
RENUMBER

```

Error 71 at line 30: Can not find specified line

Error 71 at line 40: Can not find specified line

Error 234 at line 40: Renumbered line matches specified line

LIST

```

10 REM This program demonstrates errors in a RENUMBER
20 REM
30 GOTO 9582 \ REM Line 9582 does not exist
40 GOTO 30 \ REM Line 30 does not exist, but will after the RENUMBER
50 END

```

```

7 REM This program demonstrates the RENUMBER command
8 REM
9 ON ERROR GOTO 30
10 PRINT "Enter N (1, 2 or 3)" \ INPUT N
18 ON N GOSUB 100,132,159
20 GOTO 10 \ REM Go back to line 10 for more input
24 REM
26 REM Note: The 'line 10' comment is not changed after a RENUMBER
28 REM
30 IF ERL=18 THEN IF N<0 THEN 999 ELSE PRINT "Invalid N" \ RESUME 10
31 ON ERROR GOTO 0
48 REM
49 REM Note: 'ERL=18' is not changed after a RENUMBER
50 REM
100 PRINT "1 squared = 1" \ RETURN
132 PRINT "2 squared = 4" \ RETURN
159 PRINT "3 squared = 9" \ RETURN
200 REM
999 END

```

*** After a RENUMBER 100,5 command ***

```
100 REM This program demonstrates the RENUMBER command
105 REM
110 ON ERROR GOTO 145
115 PRINT "Enter N (1, 2 or 3)" \ INPUT N
120 ON N GOSUB 170,175,180
125 GOTO 115 \ REM Go back to line 10 for more input
130 REM
135 REM Note: The 'line 10' comment is not changed after a RENUMBER
140 REM
145 IF ERL=18 THEN IF N<0 THEN 190 ELSE PRINT "Invalid N" \ RESUME 115
150 ON ERROR GOTO 0
155 REM
160 REM Note: 'ERL=18' is not changed after a RENUMBER
165 REM
170 PRINT "1 squared = 1" \ RETURN
180 PRINT "2 squared = 4" \ RETURN
180 PRINT "3 squared = 9" \ RETURN
185 REM
190 END
```

Note that line numbers in REM statements and the ERL function have not been changed.

RUN

Syntax

RUN [filespec]

Examples of Syntax

RUN MYPROG

RUN

Purpose

RUN begins program execution.

Comments

- ❑ The RUN command with no filespec included begins execution of a program in memory. If there is no program in memory, PRO/BASIC signals an error.
- ❑ The RUN command with a filespec included transfers a program currently on disk/diskette to memory and then begins execution.
- ❑ The RUN command closes all files, initializes all numeric variables to 0 and all string variables to zero length, and releases all function definitions before starting execution.
- ❑ The RUN command begins execution at the lowest numbered line.
- ❑ The GOTO and GOSUB statements in immediate mode also can begin program execution, but they do not initialize the program.

SAVE

Syntax

SAVE [filespec]

Examples of Syntax

SAVE

SAVE MYPROG

SAVE TESTBED.BAS;6

Purpose

SAVE stores the program currently in memory to a file.

Comments

- ☐ If you do not include a file name, PRO/BASIC uses the current program name as the file name in the file specification.
If you do not include a file type, .BAS is assigned by default.
- ☐ If a file of the same name already exists, PRO/BASIC creates a new file and assigns it the next higher version number.

SET

Syntax

SET [NO] mode

where:

mode

is one of the following:

DOUBLE STEP TRACE

NO

disables the specified mode.

Examples of Syntax

SET DOUBLE

SET NO STEP

Purpose

SET controls the settable modes of PRO/BASIC processing.

Comments

- ☐ Use the SHOW command to display the modes currently in effect.
- ☐ DOUBLE
- ☐ In DOUBLE mode, all real variables created are of double precision.
For example:

```
10 A=PI
SET DOUBLE
20 B=PI
30 PRINT USING '#.#####',A,B
RUN
3.14159000000000
3.14159265358979
Ready
```

Variable A is still single precision, even after the SET double command, because it was created with single precision. (Refer to the DECLARE statement in Chapter 5 for more on creating variables with single and double precision.)

- ☐ STEP

With STEP mode you can execute one statement of a program at a time.

When in STEP mode, press RETURN to execute one statement. The CONTINUE command also causes the execution of one statement.

To use STEP mode, first specify SET STEP, then start program execution with RUN. SET STEP has no effect if the program has not been run, or if the program has executed to completion. STEP mode can be set at any time. STEP mode will stop at a STOP statement in a program; press RETURN again to execute through the stop. STEP mode execution will also stop at an error; when this happens, you cannot continue.

SET STEP is similar to the STEP command, though the STEP command allows the execution of a number of statements at a time.

Refer to the STEP command in this chapter for examples comparing the effects of SET STEP and the STEP command.

□ TRACE

In TRACE mode, each line number and initial keyword executed is displayed. Each assignment executed is displayed, as are all REM statements.

FOR/NEXT loops display the FOR statement first, then each occurrence of the NEXT statement displays NEXT and the latest value of the counter variable. This mode is useful for debugging a program because it shows the current values in variables and displays events of program execution.

```

10 REM This is the first line
20 A=1+2
30 IF A=3 THEN PRINT 'A is equal to 3'
RUN
A is equal to 3
Ready
SET TRACE
RUN
10          REM This is the first line
20          A = 3
30          IF   PRINT
A is equal to 3

Ready
```

- The settable modes of PRO/BASIC are unrelated to Immediate and Program modes.

SHOW

Syntax

SHOW [identifier]

where:

identifier is a variable, array, or function in the current program.

Examples of Syntax

SHOW

SHOW I%

SHOW COS

Purpose

SHOW displays detailed information about the current program, memory use, and PRO/BASIC's settable modes.

Comments

- Type SHOW by itself to display information on all variables, arrays, and functions in the current program. SHOW displays the name and the data type of each existing variable, the subscripts of an array, and the arguments in a function. Individual array elements cannot be displayed with SHOW.

SHOW also displays the program size, the number of lines, the number of identifiers and the amount of available memory. SHOW displays a list of the currently enabled settable modes. (See the SET command in this chapter for more on settable modes.)

- Type SHOW, and the name of any variable, array, library or user-defined function to display its current value.

Example

Below are four unrelated program lines. SHOW is executed, displaying the contents of variables, arrays, etc., before and after the four lines are executed.

```
10 A$ = "abcdef"
20 DIM B(5,10)
30 C = SIN(6)
40 DEF FNAREA(D) = 2*PI**D \ D = FNAREA(22)
SHOW
```

```
A$ =
B(?,?)
C = 0
FNAREA(real) : real user defined function
D = 0
```

```
Program Size (bytes): 86
Number of lines :      4
Number of Symbols: 36
Free Memory:          5844
```

```
RUN
Ready
SHOW
```

```
A$ = abcdef
B(5,10)
C = -.279416
FNAREA(real) : real user defined function
D = 138.23
```

```
Program size (bytes) 86
Number of lines:      4
Number of Symbols: 36
Free Memory:          5564
```

STEP

Syntax

STEP [integer]

where:

integer is a positive integer constant indicating the number of statements to execute before stopping; the default is 1.

Example of Syntax

STEP 10

Purpose

STEP allows you to execute statements in your program, one or more at a time, according to the number specified.

Comments

- ❑ If the number provided is greater than the number of statements in the program, execution ceases at the last statement in the program.
- ❑ STEP implies a RUN statement, that is, it causes execution of a program to begin if the program has not been run or has executed to completion.
- ❑ Step will halt at a STOP statement. Use STEP or CONTINUE to continue execution through the STOP statement. STEP will also halt at an error. The cause of the error should be eliminated and the program restarted.
- ❑ SET STEP and STEP can be used together in a program. (Refer to the SET command in this chapter for more on SET STEP mode.)
- ❑ Use the SHOW command to display all identifiers and their values, or type the name of the identifier to display its contents.

Example

The statements of the program below are executed with SET STEP. A statement is executed by pressing RETURN.

```

10 PRINT 1
20 PRINT 2 \ PRINT '2A' \ PRINT '2B'
30 PRINT 3
40 PRINT 4
50 PRINT 5
55 STOP
60 PRINT 6
SET STEP
RUN
  1
RETURN
  2
RETURN
  2A
RETURN
  2B
RETURN
  3
RETURN
  4
RETURN
  5
RETURN
STOP at line 55
RETURN
  6
Ready

```

The statements of this program are executed with the STEP command. Any number of statements to execute can be specified in STEP.

```

10 PRINT 1
20 PRINT 2 \ PRINT '2A' \ PRINT '2B'
30 PRINT 3
40 PRINT 4
50 PRINT 5
55 STOP
60 PRINT 6
STEP 1
  1
STEP 1
  2
STEP 2
  2A
  2B
STEP
3
STEP 2
  4
  5
STEP
STOP at line 55
CONT
  6
Ready

```

5

Statements

Chapter 5

Statements

This chapter lists the statements of PRO/BASIC. Statements are the individual instructions that make up a program.

The presentation of each statement consists of four parts:

- ☐ *Syntax* – the syntax required by the statement
- ☐ *Purpose* – what the statement can do in a program
- ☐ *Comments* – explanation and suggestions for use
- ☐ *Example* – typical or explanatory example

CALL COLLATE

Syntax

CALL COLLATE (expression1, expression2, variable)

where:

expression1	is a string expression.
expression2	is a string expression.
variable	is an integer variable which receives the result.

Example of Syntax

```
100 CALL COLLATE(A$,B$,C%)
```

Purpose

CALL COLLATE compares two strings to determine the relation of one to the other based on the order of characters in the DEC Multinational Character Set.

Comments

- The DEC Multinational Character Set includes many characters specific to a variety of languages. **CALL COLLATE** provides for the correct alphabetic ordering of letters that are identical except for accents. For example, **CALL COLLATE** allows A, Å, Ä, and Å to sort between A and B, although the numeric equivalent of each of these letters would not by itself result in the proper alphabetic sequence.
- If expression1 is alphabetically sorted before expression2, the value 1 is returned to the integer variable; if the string expressions are equal, 0 is returned; if expression1 is greater than expression2, -1 is returned.
- Refer to Chapter 2 and Appendix A for more on character sets.

Example

```

1 REM Sort DEC Multinational Characters using CALL COLLATE
5 LAST=156
10 DIM A$(LAST)
11 FOR I%=32% TO 126% \ A$(I%-32%)=CHR$(I%) \ NEXT I%
12 FOR I%=192% TO 253% \ A$(I%-192%+95%)=CHR$(I%) \ NEXT I%
100 FOR I=0 TO LAST
110   MIN=I \ CH$=A$(MIN)
150   FOR J=I+1 TO LAST
200     CALL COLLATE (CH$,A$(J),I%)
250     IF I%<0 THEN MIN=J \ CH$=A$(MIN)
260   NEXT J
261   IF MIN=I THEN 270
262   TEMP$=A$(I) \ A$(I)=A$(MIN) \ A$(MIN)=TEMP$
270 NEXT I
300 FOR I=0 TO LAST
310   PRINT USING "L",A$(I);
320 NEXT I
330 END
RUN

```

```

! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A a B b C c D d E e F f G g H h I i J j
K k L l M m N n O o P p Q q R r S s T t U u V v W w X x Y y Z z [ \ ] ^ _ ` { | } ~ ¡ ¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾
Ready

```

CHAIN

Syntax

CHAIN filespec [WITH value [,value] ...] [LINE linenum]

where:

filespec	is a string expression that specifies a file containing a PRO/BASIC program.
value	is one or more constants or variables to be preserved for reference by the chained-to program.
linenum	is a line number in the chained-to program.

Examples of Syntax

CHAIN "MOD3.BAS" WITH A, B%, C\$ LINE 200

CHAIN FILENAME\$ WITH TABLE%(.)

Purpose

CHAIN transfers program control from one program to another.

Comments

- When a **CHAIN** statement is executed, all variables and functions are erased, memory is cleared, and the chained-to program is transferred into memory and executed.

If the **LINE** option is used, the same processes take place, but execution of the chained-to program starts at the specified line number. If no line number is specified, execution of the chained-to program starts at the lowest numbered line.

If the **WITH** option is used, the specified values are saved and all others are erased, memory is cleared, and the chained-to program is transferred into memory and executed. When the **PROGRAM** statement in the chained-to program is executed, values previously saved in the **CHAIN** statement are associated with the variables in the **PROGRAM** statement. (Refer to the **PROGRAM** statement in this chapter.)

- When you pass values between programs, the data types of the values in the CHAIN statement must agree with the data types of the variables in the PROGRAM statement (string values to string variables, numeric values to numeric variables). For example:

```
100 PROGRAM AAA
110 CHAIN 'BBB' WITH A, 2, C$
```

Program AAA chains to program BBB, and passes the value in variable A, the integer constant 2, and the value in variable C\$.

```
100 PROGRAM BBB (X, Y%, A$)
```

Note that although the names of the variables have been changed in program BBB, variable X is real, variable Y% is integer, and variable Z\$ is string.

Numeric values listed in the WITH option are converted to the data type of the target variable in the PROGRAM statement of the chained-to program.

However, when you pass an array between programs, the array variables must be of the same type in both programs, that is, an integer array variable in one program must be integer in the program receiving the array variable, and so on.

- To specify an array in a CHAIN statement, use a pair of parentheses containing only the appropriate number of commas to indicate the number of dimensions in the array. For example:

```
100 DIM A(3),B(4,5),C(6,7,8)
110 CHAIN 'NEXT.BAS' WITH A(),B(,),C(,,)
```

In the preceding example, arrays A,B, and C, are real arrays of 1,2, and 3 dimensions respectively.

- You can pass an array, an element from an array, or an element from a virtual array, between programs; but you cannot pass an entire virtual array.

- If a CHAIN statement includes the WITH option, a PROGRAM statement must be executed in the chained-to program in order to assign values passed to new variables. Be certain to execute a PROGRAM statement before using passed variables. By convention, the PROGRAM statement is on the first line of the chained to program.

Note: When you type in or edit a program that contains a CHAIN statement, be sure to enter a SAVE command before running your program. Otherwise, you will lose the current program.

- Programs are chained to by their file specifications, not program names. (See the following example.)

Example

In this example, file AAA.BAS contains program A, file BBB.BAS contains program B, and file CCC.BAS contains program C. The chained-to programs are referenced by 'BBB' and 'CCC', which are file specifications.

```

100 PROGRAM A
110 PRINT 'Entering AAA'
120 A$='This is how you can pass data'
130 CHAIN 'BBB' WITH A$

100 PROGRAM B(B$)
110 PRINT 'Entering BBB'
120 PRINT B$\B$ = 'between program segments'
130 CHAIN 'CCC' WITH B$

100 PROGRAM C(C$)
110 PRINT 'Entering CCC'
120 PRINT C$
130 END

```

Running AAA produces the following results:

```

RUN

Entering AAA
Entering BBB
This is how you can pass data
Entering CCC
between program segments

Ready

```

CLOSE

Syntax

```
CLOSE [#channelnum[,#channelnum] ... ]
```

where:

#channelnum is a numeric expression specifying currently open channel.

Examples of Syntax

```
CLOSE #2%,#5%
```

```
CLOSE
```

Purpose

CLOSE closes a file and ends its association with a channel.

Comments

- ❑ Closing files when they no longer need to be open: frees memory, protects the data in the files, and allows the re-use of the channel number. In any case, all open files are closed by the execution of EXIT, OLD, and RUN commands or a CHAIN or by an END statement.
- ❑ If you do not specify a channel number in the CLOSE statement, all open files are closed when it is executed. Valid channel numbers are 1 – 15.
- ❑ No error is produced if the CLOSE statement is used with no channel numbers when no files are open.
- ❑ The screen and keyboard (channel 0) cannot be closed.

Example:

```
200 PRINT 'File name'; \ INPUT FILNAM$
210 OPEN FILNAM$ FOR INPUT AS FILE #1%
   ( process file )
500 CLOSE #1%
510 GOTO 200
```

DATA

Syntax

DATA value[,value] ...

where:

value is real, integer, or string, quoted or unquoted.

Examples of Syntax

100 DATA 1,0,"ABC","1,2",0,0

300 Data January,February,March,April,May,June,

Purpose

DATA stores values for use by the READ statement.

Comments

- ❑ Values are read from the DATA statements from left to right. When the last value in a data statement is assigned to a variable, the first value in the next DATA statement is taken.
- ❑ A DATA statement can contain as many values as space on the line allows. As many DATA statements as needed can be used.
- ❑ A DATA statement must be the only statement on the line. PRO/BASIC handles all characters from the keyword DATA to the end of the line as data.
- ❑ Values can be numeric or string. The data type of the value must agree with the data type of the variable it is assigned to (string values to string variables, numeric values to numeric variables). Use exactly the same data type for numeric values (integer to integer, single to single and double to double). Strings that contain spaces before (leading spaces) or spaces after (trailing spaces) a printing value, or commas, must be quoted.

- ❑ If a **DATA** value consists only of space(s), the variable receiving that value is assigned zero if it is numeric and null if it is string.
- ❑ Use the **RESTORE** statement to reread data. (See the **RESTORE** statement in this chapter.)
- ❑ **PRO/BASIC** ignores excess data in **DATA** statements when there are more reads than data values.
- ❑ Do not follow the last value in a **DATA** statement with a comma.
- ❑ The **READ** and **DATA** statements must be used together. (Refer to the **READ** and the **RESTORE** statements in this chapter.)

DECLARE

Syntax

```
DECLARE { SINGLE } { #channelnum, array(subscript(s))
                  { DOUBLE } { array(subscript(s)) [,array(subscript(s))...] }
                              { variable           [,variable]... } }
```

where:

SINGLE/DOUBLE	is the precision selected.
#channelnum	specifies a virtual array file.
array	is an identifier of up to 32 characters that specifies a real array.
subscript(s)	is one or more numeric constants or variables, separated by commas and enclosed in parentheses.
variable	is a real variable.

Examples of Syntax

```
100 DECLARE DOUBLE TABLE1, TABLE2, TABLE4
```

```
100 DECLARE SINGLE A,B(3),C,D(5,4),E,F
```

Purpose

DECLARE creates and assigns a precision to one or more real data type variables or arrays.

Comments

- ❑ A variable or array created with single precision will contain a value of up to 6 digits; a variable or array created with double precision will contain a value of up to 16 digits. Use **PRINT USING** to display the contents of a double precision variable. (Refer to the **PRINT USING** statement in this chapter.)
- ❑ If you specify a subscripted variable, the **DECLARE** statement creates an array in the same manner that the **DIM** statement does. (Refer to the **DIM** or **DIM #** statements in this chapter for more on creating arrays.)

- ❑ The DECLARE statement establishes the precision of a program's real variables. Because the accuracy of a result depends upon the smallest amount of precision in any of the operands, all operands in a computation that requires double precision should be double precision.
- ❑ Because DECLARE initializes variables to zero when executed, including those that existed before, it is recommended that you locate this statement at the beginning of your program.
- ❑ It is not advisable to name a variable in one DECLARE statement and name it once again in another DECLARE.

Example

In this example, the real variable A is declared with single precision in line 10. The variable B is declared with double precision in line 40. The single precision variable A displays six digits of precision, while variable B, declared with double precision, displays 16 digits.

```

10 DECLARE SINGLE A
20 A=1234567890123456
30 PRINT A
40 DECLARE DOUBLE B
50 B=1234567890123456
60 PRINT USING '#####',B
RUN
      .123457E+16
1234567890123456

```

DEF

Syntax

DEF FNfunction-name [(variable[variable,...])]=expression

where:

FN function-name	is an identifier. The function-name must be preceded by FN. The function-name can have a suffix to indicate the type of value returned by the function; percent sign (%) for integer, dollar sign (\$) for string, or no suffix, indicating a real number is returned.
variable	is a temporary variable to reserve space for the arguments passed to the defined function.
expression	returns a value of the same data type as the function name.

Examples of Syntax

```
DEF FNSTRIP$(S$)=MID$(S$,2,LEN(S$)-1)
```

```
DEF FNADDR%(N%,P%)=N%*(N%+1%)/2
```

Purpose

DEF allows you to define functions that are used in the same way as library functions, such as LOG or SIN.

Comments

- The DEF statement defines the function, as in line 10 in the following example. When the function is referenced (line 20), the expression to the right of the definition statement is evaluated and the result returned.

```
10 DEF FNDEC$='Digital Equipment Corporation'
20 PRINT FNDEC$
RUN
Digital Equipment Corporation
Ready
```

You can also define a function that returns a value that depends on the value(s) passed to it. This is a more dynamic use of the DEF statement and is treated next.

In the DEF statement, the two following items are supplied:

- Zero or more temporary variable(s), to be replaced by an argument value when the defined function is referenced.
- The expression, which returns a value.

In the following example, line 10 is a function definition;

```
10 DEF FNAREA(A)=2*PI*A
```

When the defined function is referenced, as many argument values are supplied as there are temporary variables holding places for them.

In the following example, line 20 is a function reference, passing the value 1.9 to the temporary variable 'A' in the function definition, with a PRINT statement to display the results:

```
20 PRINT FNAREA(1.9)
```

The names of the temporary variables in a function definition have no purpose other than to act as place holders for argument values passed from a function reference. An intermediate step in the execution of a function would look like this:

```
FNAREA(1.9) = 2*PI*1.9
```

In the following example temporary variable 'A' is replaced with 'APPLE' with no change to the meaning of the statement:

```
10 DEF FNAREA(APPLE)=2*PI*APPLE
20 PRINT FNAREA(1.9)
Ready
RUN
11.9381
```

When PRO/BASIC encounters a user-defined function reference, it goes through the following sequence: it evaluates the expressions in the function reference, looks up the DEF statement expression, replaces the temporary variables with the corresponding argument values from the function reference; evaluates the DEF statement; and it returns the result.

- The expression on the right side of a function definition can contain variables, just like an assignment expression, in addition to temporary variables.

- ❑ No more than 5 temporary variables can be passed to the function definition.
- ❑ User-defined functions can reference library functions or other user-defined functions. For example, the following is a valid DEF statement:

```
DEF FNA=SIN(FNB)
```

A user-defined function reference is valid in immediate mode; however, the DEF statement is not.

- ❑ There can only be one function associated with a DEF statement.

Example

```
10 REM Function to convert an angle from degrees to radians
20 DEF FNANGLE(A)=A*2*PI/360
30 PRINT "ENTER ANGLE TO BE CONVERTED"
40 INPUT A
50 PRINT "THE RADIAN OF THAT ANGLE IS";FNANGLE(A)
60 PRINT
70 PRINT "WANT TO CONVERT ANOTHER ANGLE? ENTER Y"
80 PRINT "-- OR PRESS ENTER TO END"
90 INPUT X$|IF X$="Y" THEN GOTO 20
99 END
RUN

ENTER ANGLE TO BE CONVERTED
? 45

THE RADIAN OF THAT ANGLE IS .785398

WANT TO CONVERT ANOTHER ANGLE? ENTER Y
-- OR PRESS ENTER TO END
? RETURN
Ready
```

DIM

Syntax

`DIM[ENSION] array-name(subscript(s))[,array-name(subscript(s))]`...

where:

array-name	is an identifier. The array-name can have a suffix to indicate the type of data the array will contain; percent sign (%) for integer, dollar sign (\$) for string, or no suffix, indicating real numbers.
subscript(s)	is one or more numeric constants or variables, separated by commas and enclosed in parentheses.

Examples of Syntax

```
10 DIM A(6)
```

```
10 DIM B(100),C(15,3)
```

Purpose

DIM creates an array by specifying the array's name and dimensions.

Comments

- The array-name determines the type of data stored in the array. The array's dimensions are specified by a subscript or subscripts. Each subscript used creates one dimension in an array. The size of a dimension is equal to the number that created it. (Note that the use of a subscript both establishes a dimension and specifies its size.) The size of an array is determined by the number of dimensions and the sizes of the dimensions. Up to 7 dimensions can be defined in an array. In the first of the examples in the Syntax section above, A is an array of one dimension, with 7 elements (starting at zero, counting to 6; dimensions begin at zero). In the second example, one DIM statement defines two arrays: B is an array of one dimension, of 101 elements (starting at zero, count to 100); C is an array of two dimensions, 16 elements by 4 elements, that total 64 elements.

So far, only numeric constants have been used as subscripts to define arrays. An array with constants for all the subscripts is a static array. The size of a static array cannot be changed. An array defined as static cannot be referred to again in a subsequent DIM statement.

- You can also create an array that can be redimensioned by using variables for subscripts instead of constants. This is called a dynamic array.

If one or more variables are used to define an array, the size of the array can be changed by changing the values in the variables and repeating the DIM statement, or by using different variables and repeating the DIM statement. The number of dimensions cannot be changed, nor can all the original variable subscripts be replaced with constants.

In the following example, array A is first created with 2 dimensions of 6 and 8 elements. On line 40, variable I is assigned value 23. On line 50, another DIM statement is used to change the dimensions of array A.

```
10 X=5\Y=7
20 DIM A(X,Y)
30 REM array A is now 6 by 8 elements, or 48 elements in size
40 I=23
50 DIM A(I,15)
60 REM array A is now 24 by 16 elements, or 384 elements in size
70 REM array A is dynamic because one subscript is a variable
```

- An array can be created without the DIM statement by referencing the array-name. When you reference an array that does not exist (has not been created using a DIM statement), PRO/BASIC creates a new array with that name. This is called an implicit array. When you create an array implicitly, each dimension contains 11 elements numbered 0-10. In the following example, line 10 creates an 11-by-11 array that stores real numbers and assigns the value 3.14159 to element (5,5).

```
10 LET A(5,5)=3.14159
```

All implicitly created arrays are static and cannot be redimensioned.

- An array explicitly created using constants cannot be redimensioned.
- When an array is created (with the DIM statement or implicitly) its elements are initialized to zero for numeric arrays, or to null strings for string arrays. Arrays are also initialized when redimensioned.

- ❑ Use only a variable or a constant for a subscript; do not use expressions.
- ❑ See the DIM # statement in this chapter for information on creating arrays in files.

Example:

```

10 REM - PROGRAM TO CREATE A 4-BY-6 ARRAY AND FILL WITH NUMBERS
20 DIM A(3,5)
30 FOR I=0 TO 3 \FOR J=0 TO 5
40   A(I,J)=I+J*10
50 NEXT J \NEXT I
60 FOR I=0 TO 3 \FOR J=0 TO 5
70   PRINT A(I,J);
80   NEXT J
90   PRINT
100 NEXT I
110 END
RUN
0   10   20   30   40   50
1   11   21   31   41   51
2   12   22   32   42   52
3   13   23   33   43   53

```

Ready

DIM

Syntax

DIM[ENSION] #channelnum, array-name(subscript(s)) [=string length]

where:

#channelnum	specifies a virtual array file.
array-name	is an identifier. The array-name can have a suffix to indicate the type of data the virtual array will contain; a percent sign (%) for integer, a dollar sign (\$) for string, or no suffix, indicating real numbers.
subscript(s)	is one or more numeric constants or variables separated by commas and enclosed in parentheses.
string length	is a positive numeric constant which specifies the maximum length of the elements of a string array.

Examples of Syntax

```
10 DIM #1,A$(100)=80
```

```
10 DIM #3,TABLE%(15,30)
```

Purpose

DIM # creates a virtual array by specifying the array's name and dimensions.

Comments

- A virtual array stores data in a file, rather than in memory. When the file is open, you can access the elements by subscript. Because the array is stored in a file, its contents may be reused.
- **DIM#** declares the array's name and dimensions, and identifies the channel number of a file containing a virtual array. The array-name determines the type of data to be stored in the array. Each subscript used creates one dimension in a virtual array. The size of a dimension is equal to the number that created it. (Note that the use of a subscript both establishes a dimension and specifies its size.) Up to 7 dimensions can be defined in an array.

- A virtual array can be defined using variables in place of constants, for subscripts. (See the DIM statement in this chapter for more on the use of variables.) However, you cannot redimension a virtual array, even when variables are used.
- Because a virtual array is stored in a file, PRO/BASIC must execute an OPEN statement with the same channel number specified in the DIM # statement before any references are made to elements of the array. The OPEN statement must contain the qualifier "VIRTUAL." For example:

```
10 DIM #1, A$(1000)=132
20 OPEN 'VIRT.DAT' FOR OUTPUT AS FILE #1, VIRTUAL
30 LINPUT A$(47)
```

This program fragment opens a virtual array file and stores an input line in element 47 of the file VIRT.DAT. If you do not specify an extension with the file name when handling a virtual array file, the default file type .DAT is used.

- A virtual array can be much larger than a normal array because it uses disk space rather than memory. If your program needs a very large array or needs to store information between executions, a virtual array may be helpful.

Virtual array files have two major advantages over sequential files:

1. You don't have to read values in order. For example, you can access the last element in a virtual array as easily as the first.
2. You can store numeric data with no loss of precision. In a virtual array, numbers have the same binary format that PRO/BASIC uses internally. Storing numeric data in a sequential file results in a small loss of precision for some fractions because of the binary value to character value conversion.

Virtual array files also have two major disadvantages:

1. Virtual array accesses are slower than normal because each one requires a disk input/output operation.
2. You cannot directly examine a virtual array file. Unlike text files, virtual array files are stored in binary format.

- ❑ The size of elements of string data type virtual arrays can be set. Any size specified is rounded up to a power of 2 (1,2,4,8,16,32,64,256). If a value contains more characters than the specified size, characters are lost. If a value contains fewer characters than the specified size, disk storage space is wasted.

The maximum size you can specify for a string virtual array is 256.

- ❑ The default size of elements in a virtual string array is 16.
- ❑ For compatibility with other versions of BASIC, place each DIM # statement in a lower-numbered line than the corresponding OPEN statement.
- ❑ Virtual arrays are not initialized to zero, or null strings in the case of string arrays, during creation.
- ❑ Use only a variable or a constant for a subscript in the DIM # statement.

Example

```

10 REM This example uses a dynamic array to
20 REM store a sorted list of names on disk.
30 REM
40 DIM #1,NAMES$(4)=20 \ REM Maximum name length is 20
50 OPEN 'name.dat' FOR OUTPUT AS FILE #1, VIRTUAL
60 N_NAMES=4 \ REM Maximum number of names
70 REM Initialize the array
80 FOR N=1 TO N_NAMES
90   NAMES$(N)=" "
100 NEXT N
110 REM
120 I=1
130 IF I<=N_NAMES THEN PRINT "input name "; ELSE PRINT "no room" \ GOTO 230
140 LINPUT NAMES$(I)
150 IF NAMES$(I)=" " THEN GOTO 230
160 REM Do sort upon entry
170 X=I
180 FOR J=I TO 1 STEP -1
190   IF NAMES$(X)<NAMES$(J) THEN SWITCH=1 ELSE SWITCH=0
200   IF SWITCH=1 THEN T$=NAMES$(J) \ NAMES$(J)=NAMES$(X) \ NAMES$(X)=T$  X=J
210 NEXT J
220 I=I+1 \ GOTO 130
230 REM Print the array
240 FOR I=1 TO N_NAMES
250   PRINT I,NAMES$(I)
260 NEXT I
270 CLOSE #1
280 PRINT "done"
290 END
RUN
input name ? JOHN
input name ? PAUL
input name ? GEORGE
input name ? RINGO
no room
1      GEORGE
2      JOHN
3      PAUL
4      RINGO
done
Ready

```

END

Syntax

END

Example of Syntax

32767 END

Purpose

END terminates execution of a program and closes all open files.

Comments

- The END statement should be the last statement on the last line of your program, for compatibility with other versions of BASIC.

FOR/NEXT

Syntax

FOR variable = expression1 TO expression2 [STEP expression3]

where:

variable	is the counter variable, a numeric variable.
expression1	is the initial value of the counter variable, a numeric expression.
expression2	is the final value of the counter variable, a numeric expression.
expression3	is the STEP value, a numeric expression (the default is 1).

Examples of Syntax

```
10 FOR I=1 TO 10 STEP 2
```

```
•
```

```
program statements
```

```
•
```

```
50 NEXT I
```

```
200 FOR Z% = INT(T(I)*7.02) TO 0% STEP -10%
```

```
•
```

```
program statements
```

```
•
```

```
300 statements \NEXT Z%
```

Purpose

FOR/NEXT performs a series of program instructions a number of times.

Comments

- The FOR statement acts as a counter for the loop and defines the beginning of the loop. The NEXT statement adjusts the counter and defines the end of the loop. Any statements appearing between the FOR statement and the NEXT statement are executed until the final value of the counter variable is reached. The normal line number sequence is then resumed at the statement after the NEXT statement.

These two statements must be used together. (Refer to the NEXT statement in this chapter.)

- At execution, PRO/BASIC evaluates all expressions in the FOR statement before it assigns a value to the counter variable. After evaluation, the beginning value is assigned from expression1 to the counter variable.

Expression2 contains the final value for the loop. If the value in the counter variable is larger (with positive STEP value) or smaller (with negative STEP value) than the final value, the loop is ended. Otherwise, the loop is performed and the statement(s) executed. When the NEXT is reached, the value in the counter variable is changed by the value in STEP. This value is 1 unless you specify otherwise.

Expression3 contains the optional STEP value. STEP can have any numeric value, positive or negative. STEP defines the amount by which the counter variable is to be incremented or decremented each time the loop is executed. Each time the loop is executed, the value in STEP is added to the value of the counter variable, and the result compared with the final value. The type of comparison made to the final value depends upon the sign of the STEP value. If the STEP value is positive, the counter variable is tested to determine whether it exceeds the final value; if the STEP value is negative, the test is made to determine if the value of the counter variable is less than the final value.

- If the counter variable is incorrectly greater than or less than the ending value at the start of the loop, execution of the loop will not occur, as demonstrated in the following two examples:

10 FOR I=20 TO 2 STEP 2 50 FOR J=1 TO 3 STEP -1

Neither of the two program statements will execute properly. In the first example, the initial value of the counter variable is greater than the final value, so the loop is never executed. A negative value in STEP would allow this statement to work. In the second example, the negative STEP causes the NEXT statement to test for a counter value less than the final value. However, the counter value is already less than the ending value. A positive value in STEP would allow this statement to work.

- You can place a FOR/NEXT loop inside another FOR/NEXT loop. This is called “nesting.” For example:

```

10 FOR I=0 TO 4
20   FOR J=0 TO 4
30     PRINT I*10+J,
40   NEXT J
50   PRINT
60 NEXT I

```

RUN

0	1	2	3	4
10	11	12	13	14
20	21	22	3	24
30	31	32	33	34
40	41	42	43	44

Ready

Note the use of indentation in the example. Statements within loops are indented to show that they are within the same loop. Line 30 is indented twice because it is inside two loops. Indentation makes programs easier to understand.

When you use nested loops, each loop must be completed within any outer loops, as in the preceding example, where the FOR J and NEXT J are paired within the FOR I and NEXT I. (Also, each FOR and NEXT statement pair must have a unique counter variable.) The following example demonstrates an invalid use of nesting FOR/NEXT statements:

```

10 FOR M = 1 TO 10
20 FOR N = 1 TO 10
30 NEXT M
40 NEXT N

```

- After the completion of a loop, the counter variable has the last value that caused execution of the loop. Although PRO/BASIC increments the counter variable until it is greater than the ending value, PRO/BASIC subtracts the STEP value to return the counter variable to the value last used to execute the loop. For example:

```

10 FOR I = 1 TO 3
20   PRINT I
30 NEXT I
40 PRINT I
50 END
RUN
1
2
3
3

```

The variable I does not contain the value 4, which caused the end of the loop. It was subtracted back out. The result, 3, is the 'last good value'.

Example

In this example, when line 100 is executed, CONTROL% is assigned the value 1 and PRO/BASIC tests to determine whether it has exceeded the final value of 4. It has not and the loop (line 150) is executed.

When control gets to line 200, PRO/BASIC increments CONTROL% to 3 and tests to determine whether the value of CONTROL% has exceeded the final value of 4. It has not, and control is transferred back to line 150.

The next time control gets to line 200, PRO/BASIC increments CONTROL% to 5 and tests again. This time, it exceeds the final value and control passes out of the loop to line 210. The value 3 was the last valid value and remains in CONTROL% after the loop ends.

```

100 FOR CONTROL% = 1% TO 4% STEP 2%
150   PRINT CONTROL%;
200 NEXT CONTROL%
210 PRINT 'Done.' \ END
Ready

RUN
1 3 Done.

Ready

```

GOSUB

Syntax

GOSUB linenum

where:

linenum specifies the first line number of a subroutine.

Example of Syntax

```

100 GOSUB 500 \ REM Call subroutine to perform AAA
110 REM The next operation is BBB
.
.
.
program statements
.
.
500 REM *** Subroutine to perform A ***
.
.
.
program statements
.
.
590 RETURN

```

Purpose

GOSUB saves the location of the statement following the GOSUB and transfers control to the specified line.

Comments

- In the example under the Syntax section above, the GOSUB at line 100 saves the location of the statement following the GOSUB (line 110), and transfers control to line 500. The RETURN statement at line 590 transfers control to the location after the GOSUB (line 110). (Refer to the RETURN statement in this chapter.)
- A subroutine may call another subroutine. This is called nesting. The maximum number of nesting levels depends on the amount of memory available.
- A GOSUB can transfer control to the first statement of a multistatement line, but not to subsequent ones.

- ❑ Do not allow execution of a subroutine other than by means of a GOSUB statement. If control “falls” into a subroutine, the RETURN statement will produce an error. You can use GOTO statements to transfer control around a subroutine.
- ❑ The GOSUB statement used in immediate mode can be a valuable debugging tool.

Example

```

10 A=0
20 GOSUB 60 \ A=5
30 GOSUB 60 \ A=6
40 GOSUB 60
50 GOTQ 120
60 REM This subroutine executes a loop with values from lines 20 and 30
70 FOR I = 1 TO 6
80   LET A=A+A
90   PRINT A,
100 NEXT I
110 RETURN
120 END
RUN

```

0	0	0	0	0	0
10	20	40	80	160	320
12	24	48	96	192	384

Ready

GOTO

Syntax

GOTO linenum

Example of Syntax

100 GOTO 300

Purpose

GOTO transfers program control to any line in your program.

Comments

- ❑ Execution resumes at the first statement on the line branched to, unless that line contains nonexecutable statements (such as a DATA statement). In this case, execution resumes at the first executable statement.
- ❑ Other statements must never follow a GOTO statement on the same line, since they will never be executed.
- ❑ The GOTO statement can be a valuable debugging tool in immediate mode. For example, you can put a STOP statement at the end of a problem area and execute the program. When PRO/BASIC executes the STOP, you can examine and modify variables or program statements and then transfer control with an immediate mode GOTO back into the problem area.
- ❑ Another spelling of GOTO is GO TO.

Example

The following example, taken from a program that prints out the contents of a document file, demonstrates a correct application of the GOTO statement. Because the print loop exits only when an end-of-file condition at line 210 produces an error, a simple GOTO loop is appropriate. (Assume that an ON ERROR statement is present elsewhere.)

```

200 REM *** Print loop—exit is caused by end of file ***
210 LINPUT #1%,LINE$ \ REM      Get a line
220 PRINT LINE$ \ REM           Print the line
230 GOTO 210 \ REM              End of print loop

```

IF

Syntax

IF expression

where:

expression	is any conditional, i.e., relational or logical expression.
statement	is a single statement or several statements, separated by backslashes.
linenum	is the line number in the current program to which control is transferred.

Examples of Syntax

```
IF A>B THEN 100 ELSE 200
```

```
IF A>B THEN PRINT A ELSE PRINT B
```

```
IF A>B THEN C=B \ B=0 ELSE C=A \ A=0
```

Purpose

IF tests a relational or logical expression and executes statements depending on the result.

Comments

- Use IF to perform an operation or series of operations only if a certain condition or set of conditions is satisfied. If the expression is true, PRO/BASIC executes the THEN (or GOTO) portion of the statement. If the expression is false, PRO/BASIC transfers control to the next numbered line, or executes the statement(s) in the ELSE clause, if an ELSE statement is present.

A conditional expression can test the relation of two operands. The pair of operands can be either both string or both numeric. The expression is said to be true if the relation between the two operands is true, or false if the relation between the operands is false.

A conditional expression can also test a number of relational expressions by separating the relational expressions with the logical operators AND, OR. The expression results in a true or false value depending upon the relations tested and the effect of the logical operator.

- Note that IF THEN ELSE is all one statement. The THEN clause and optional ELSE clause can only be used with the IF and must appear on the same line as the IF. More than one statement can be included after the word THEN for execution if the IF condition is true. For example:

```

10 PRINT 'WHAT DAY IS TODAY?'
20 INPUT DAY$
30 IF DAY$='FRIDAY' THEN PRINT 'HIP' \ PRINT 'HIP' \ PRINT 'HOORAY'
RUN
WHAT DAY IS TODAY?
? FRIDAY
HIP
HIP
HOORAY
Ready

```

The ability to execute multiple statements also applies to statements appearing in the ELSE clause. Note in the preceding example that the statements after THEN are separated by a backslash (\). When multiple statements are used in the IF statement, the backslash character must separate each one. A GOTO clause can end the line, or be followed by an ELSE or by a REM statement.

- An IF statement can include other IF statements (as many as will fit on one line). This is called a “nested IF” statement. For example:

```

10 IF A<B THEN PRINT '<' ELSE IF A=B THEN PRINT '=' ELSE PRINT '>'

```

An ELSE clause belongs to the closest preceding IF statement that does not already have an associated ELSE clause. In the preceding example, each ELSE belongs to the IF statement that immediately precedes it.

- An IF statement can be used to execute a loop until a certain condition is met. For example:

```

10 INPUT A$
20 IF LEN(A$)=0 THEN 100
.
.
.
90 GOTO 10
100 REM Processing continues after user enters a null string

```

Example

```

10 REM Program does an insertion sort to demonstrate IF statement
20 DIM A(12)
30 FOR I=1 TO 12
40   READ A(I)
50 NEXT I
60 HIGH=12
70 FOR I=1 TO HIGH-1
80   FOR J=I+1 TO HIGH
90     IF A(I)<A(J) THEN T=A(I) \ A(I)=A(J) \ A(J)=T
100    NEXT J
110 NEXT I
120 FOR I=1 TO HIGH
130   PRINT A(I)
140 NEXT I
150 DATA 2,3,66,55,33,44,43,21,43,32,16,78
Ready
RUN
78
66
55
44
43
33
32
21
16
3
2

```


CALL INKEY

Syntax

CALL INKEY(variable)

where:

variable is a string variable.

Example of Syntax

100 CALL INKEY(CH\$)

Purpose

CALL INKEY allows a program to respond immediately to a user's keystroke.

Comments

- CALL INKEY gets a single character from the keyboard if one is typed. If no key is pressed, it returns a null string (string of length zero), and processing continues.
- It allows escape sequences and control characters to be input directly to your program.
- The character is accepted as it is typed; it is not necessary to type RETURN.

Example

This example illustrates how CALL INKEY can filter input. In this case, only numeric keys can be entered. All other keys (except RETURN) will beep when pressed. The program stops accepting input when RETURN is pressed.

```

10 CALL INKEY(A$) \ IF LEN(A$)=0 THEN GOTO 10
20 IF A$=CHR$(13) THEN GOTO 70
30 IF A$<'0' OR A$>'9' THEN PRINT CHR$(7); \ GOTO 10
40 PRINT A$;
50 N$=N$+A$
60 GOTO 10
70 PRINT \ PRINT 'OK ';N$
RUN
123
OK 123
Ready

```

INPUT

Syntax

INPUT [#channelnum,] variable[,variable] ...

where:

#channelnum is a numeric expression specifying a valid channel number (0—15). Channel 0 is the default and indicates the keyboard.

variable receives a value from the keyboard or a file.

Examples of Syntax

```
10 INPUT A,B%,C$
```

```
10 INPUT $2,N,TABLE%(I,J)
```

Purpose

INPUT assigns data from the keyboard or a document file to one or more variables.

Comments

- When data is to be accepted from the keyboard; a “prompt” is displayed at execution indicating the program is waiting for data. For example:

```
10 INPUT N
RUN
? 5
Ready
```

If channel 0 is explicitly specified, data is to be accepted from the keyboard, but the PRO/BASIC “?” prompt is overridden. For example:

```
10 INPUT #0,N
RUN
5
READY
```

- You can use a **PRINT** statement before an **INPUT** statement to help explain the prompt when receiving data from the keyboard. This practice is particularly important when overriding the prompt, as above.

For example:

```
10 PRINT "ENTER A NUMBER" \ INPUT #0,N
RUN
ENTER A NUMBER
5
READY
```

- ❑ If a channel number from 1 to 15 inclusive is specified, data is to be accepted from a file. When receiving data values from a file, all data items required for assignment to variables must be satisfied from one record.
- ❑ Data types of values and the variables they are assigned to must agree.
- ❑ Leading and trailing blanks are deleted from data values. PRO/BASIC does not alter the contents of a quoted string.
- ❑ Data values accepted from the keyboard can be input one-by-one in response to individual prompts, or all values can be entered to one prompt for assignment to variables. Separate each value with a comma when entering multiple values to one prompt.
- ❑ Use the DELETE key to erase incorrect characters entered to an INPUT prompt. The full range of keys of the Line Editor is not available when responding to INPUT because you are entering data, not text.

Example

This example program adds a list of numbers together, printing out the sum of all the numbers entered after each number is entered.

```
10 SUM=0 \ N=0
20 SUM=SUM+N
30 PRINT SUM,
40 INPUT N
50 GOTO 20
RUN
0          ? 12
12         ? 137.98
149.98     ? .23
150.21     ? -50
100.21     ? INTERRUPT/DO
```

Error 28 at line 40 INTERRUPT-DO keys entered

KILL

Syntax

KILL filespec

where:

filespec is a string expression that indicates a file specification.

Example of Syntax

```
120 KILL 'TEMP.DAT'
```

Purpose

KILL deletes the specified file(s) from a storage device.

- ☐ When no file type is included, .DAT is the default.
- ☐ If a version number is not specified, all versions of the specified file are deleted.
- ☐ If no files match the file specification, an error message is displayed, and no files are deleted.

Example

```
100 OPEN 'TEMPXX.DAT' FOR OUTPUT AS FILE #1  
    (use the file)  
670 CLOSE #1 \ KILL 'TEMPXX.DAT'
```

LET

Syntax

[LET] variable[,variable] ... = expression

where:

expression is any valid value or variable or expression.

Examples of Syntax

```
10 LET A=482.5
```

```
10 A,B,C(7%)=0 \ E$=F$
```

```
10 LET TAX=PRICE*RATE
```

Purpose

LET assigns the value of the expression on the right of the equal sign to the variable or list of variables on the left of the equal sign.

Comments

- ☐ The equal sign, when used by the LET statement, assigns a value and does not indicate algebraic equality.
- ☐ The keyword LET is optional and can be omitted.
- ☐ All expressions are evaluated before assignment.
- ☐ You cannot mix numeric and string data types in an assignment statement (assign a numeric value only to a numeric variable, and a string value only to a string variable).

You can mix numeric values of all data types in calculations and assignment; integer, single and double precision values can be used together.

- ☐ You can assign a value to a user-defined function only if the assignment statement is the definition of that function (see the DEF statement in this chapter).

Example

```
LET A=1  
PRINT A  
1  
A=2  
PRINT A  
2
```

LINPUT

Syntax

LINPUT [#channelnum,] variable

where:

#channelnum is a numeric expression specifying a valid channel number (0—15). Channel 0 is the default and indicates the user's terminal.

variable receives a line from the keyboard or a file.

Examples of Syntax

```
10 LINPUT A$
```

```
10 LINPUT #2,NAMelist$(I,J)
```

Purpose

LINPUT assigns all characters on a line accepted from the keyboard or a document file to a string variable.

Comments

- LINPUT treats the entire input line as a string data item and does not change it in any way.
- When data is to be accepted from the keyboard; a prompt is displayed at execution indicating the program is waiting for data. For example:

```
10 LINPUT ITEM
?
```

If channel 0 is explicitly specified, data is to be accepted from the keyboard, but the PRO/BASIC “?” prompt is overridden. For example:

```
LINPUT #0,A$
```

- Use a PRINT statement before a LINPUT statement to help explain the prompt when receiving data from the keyboard. This practice is particularly important when overriding the prompt, as above.

```
30 PRINT "ENTER YOUR LAST NAME"; \ LINPUT LAST_NAMES$
```

- ❑ Use the DELETE key to erase characters incorrectly entered to a LINPUT prompt. The full range of keys of the Line Editor is not available when responding to LINPUT, because you are entering data, not text.
- ❑ If a channel number from 1 to 15 is specified, data is to be accepted from a file.
- ❑ When entering data from a document file, and end-of-file error occurs when there are no records left.

Example

This example prints out parts of a sentence, split at the spaces. Everything you type is accepted and stored in the variable (LINE\$ in this case).

```

10 LINPUT LINE$
20 P=POS (LINE$, ' ',1)
30 IF P>0 THEN PRINT MID$(LINE$,1,P) \ LINE$=MID$(LINE$,P+1,1000) \ GOTO 20
40 PRINT LINE$
RUN
? "What?", she said, thoughtfully.
"What?",
she
said,
thoughtfully.
Ready

```


NAME AS

Syntax

NAME filespec1 AS filespec2

where:

filespec1 is a string expression that specifies an existing file.

filespec2 is a string expression that specifies the new file specification.

Examples of Syntax

```
300 NAME A$ AS B$
```

```
950 NAME 'TEST.DAT' AS 'TEST.BAK'
```

Purpose

NAME AS changes the name of a file.

Comments

- ☐ If the second file specification already exists, PRO/BASIC creates a new file specification with the next higher version number.
- ☐ The first file specification will no longer exist after execution; the new file specification replaces the old one and the old one is deleted.
- ☐ The default file type .DAT is used if no file type is provided.
- ☐ Both files must be on the same device.

Example

```
100 REM *** File processor ***
110 PRINT 'File name' \ INPUT FILENAME$
120 OPEN FILENAME$ FOR INPUT AS FILE #1
130 OPEN 'TEMPXX.DAT' FOR OUTPUT AS FILE #2
    (process the file)
900 CLOSE \ KILL FILENAME$
910 NAME 'TEMPXX.DAT' AS FILENAME$
```

NEXT

Syntax

NEXT variable

where:

variable is the counter variable, and is the same numeric variable named in the associated FOR statement.

Examples of Syntax

50 NEXT I

Purpose

The NEXT statement defines the end of a FOR/NEXT and returns control to the FOR statement.

Comments

- The FOR statement sets the counter for the loop. The NEXT statement adjusts the counter.
- The NEXT statement must be used with the FOR statement. (Refer to the FOR/NEXT statement for more information and examples.)

ON ERROR

Syntax

ON ERROR GOTO linenum

where:

linenum is an existing line number or 0. 0 is the default and causes errors to be handled by PRO/BASIC.

Examples of Syntax

ON ERROR GOTO 0

ON ERROR GO TO 400

Purpose

ON ERROR allows you to control error handling.

Comments

- PRO/BASIC detects computational and input/output errors during execution. Normal error handling then displays a warning message and continues, or displays a fatal error message and stops program execution. For example:

```
110 A=1/0 \ REM Division by 0 causes fatal error
RUN
```

```
Warning 61 at line 110: Division by zero is not defined
Ready
```

User-defined error handling can be included in the program to replace normal error handling.

To enable user-defined error handling use ON ERROR GOTO with a line number. The line number is the location of an error-handling routine. When an error occurs after ON ERROR (linenum) is executed, PRO/BASIC transfers control to the line specified in the ON ERROR statement. (If an error occurs before the ON ERROR statement is executed normal error handling is used.)

In the following example, when an error occurs in line 110, program control is immediately transferred to the error-handling routine at line 200.

```

100 ON ERROR GOTO 200
110 A=1/0 \ REM Division by 0 causes fatal error
120 REM next line not executed due to the error in line 110
130 PRINT A=100
200 PRINT 'Entering error-handling routine'
210 RESUME 300
300 END
RUN

Entering error-handling routine
Ready

```

- Error handling can be returned to PRO/BASIC at any time in your program.

To return to normal error handling use ON ERROR GOTO 0.

In the following example, if an error occurs before line 300, PRO/BASIC handles it. If an error occurs after line 300 but before line 10000, the error is handled by the error-handling routine at 19000. If an error occurs between lines 10000 and 12000 it is handled by PRO/BASIC.

100		errors that
110	program statements	occur between
120		lines 100 and 300
.		are handled by
.		PRO/BASIC
300 ON ERROR GOTO 19000		
.		errors that
.		occur between
400	program statements	lines 300 and 1000
410		are handled by
.		user
.		
10000 ON ERROR GOTO 0		
.		errors that
.		occur between
12010	program statements	lines 10000 and
12015		14000 are handled
.		by PRO/BASIC
.		

```

14000 GOTO 32767
.
.
19000 REM      error handling statements
32767 END

```

- PRO/BASIC provides three special functions that contain status information (after execution of an ON ERROR statement) for use by your error-handling routine.

ERL contains the number of the line at which the last error occurred.

ERR contains the number of the error.

ERT\$ contains the error message associated with the specified error number.

Use IF statements to test the contents of these functions.

Place an ON ERROR GOTO 0 statement at the end of an error handler, after testing for likely errors, to allow PRO/BASIC to handle any errors your error handler does not test for. For example:

```

19000 REM – Error Handling Routine
19010 IF ERR=54 THEN PRINT 'Imaginary Square Root' \ RESUME
19020 IF ERR=55 THEN PRINT 'Subscript out of Range' \ STOP
19030 ON ERROR GOTO 0

```

Use the RESUME statement to leave the error handler and return to regular processing. See the RESUME statement in this chapter.

- Errors that occur during error handling are not recoverable.
- User-defined error handling can be used only in a program, not in immediate mode.
- A frequent use of ON ERROR is to detect the end of a file (as in line 1900 in the following example).

Example

```

10 ON ERROR GOTO 19000
20 OPEN 'FILE.LST' FOR INPUT AS FILE #1
30 LINPUT #1, A$
40 PRINT A$
50 GOTO 30
19000 IF (ERR=11) AND (ERL=30) THEN CLOSE #1 ELSE ON ERROR GOTO 0
32767 END

```

ON GOSUB

Syntax

On expression GOSUB linenum [,linenum] ...

where:

expression is a numeric expression.

Examples of Syntax

```
ON I GOSUB 100,200
```

```
ON INT(4*RND+1) GOSUB 300,500,700,900
```

Purpose

ON GOSUB branches to one of several subroutines and saves the location of the statement following the ON GOSUB to return to.

Comments

- The ON GOSUB statement branches to one of several subroutines (represented by the first program line of each) depending upon the value of an expression. The number of valid values the expression can result in starts with the value 1 and continues until equal to the number of line numbers listed.

The RETURN statement is located at the end of the program statements of the subroutine. When the RETURN is executed control branches back to the location previously recorded. (Refer to the RETURN statement for more information.)

- At execution of an ON GOSUB statement, PRO/BASIC first evaluates the numeric expression, and truncates the value to an integer if necessary. If the result of the expression is 1, control branches to the

first line number in the list, if the value is 2, control branches to the second line number in the list, and so on. For example:

```
50 ON A*B+1 GOSUB 100,200
```

when:

$A*B+1 = <1$ PRO/BASIC signals an error

$A*B+1 = 1$ PRO/BASIC goes to the subroutine on line 100

$A*B+1 = 2$ PRO/BASIC goes to the subroutine on line 200

$A*B+1 = >2$ PRO/BASIC signals an error

If the expression is less than 1 or greater than the number of line numbers in the list, PRO/BASIC displays an error message.

A negative or 0 value always causes an error.

It is a good practice to check the value of the expression before the ON GOSUB statement is executed.

- Subroutines using ON GOSUB can be nested. The maximum number of nesting levels depends on the amount of memory available.
- Do not allow execution of a subroutine except by GOSUB or ON GOSUB. If a subroutine is executed mistakenly, the RETURN statement at the end of the subroutine will cause an error.

Example

In this example, the third action (printing '-') is shared by the other two actions, and the second action (printing '<-->') is shared by the first action (printing '['<-->]').

```

10 RANDOMIZE
20 FOR I=1 TO 10
30     ON RND*3+1 GOSUB 70,100,130
40     PRINT
50 NEXT I
60 GOTO 160
70 REM First subroutine
80     PRINT '['; \ GOSUB 100 \ PRINT ']';
90 RETURN
100 REM Second subroutine
110     PRINT '<'; \ GOSUB 130 \ PRINT '>';
120 RETURN
130 REM Third subroutine
140     PRINT '-';
150 RETURN
160 END
RUN
[<-->]
-
<-->
-
[<-->]
[<-->]
[<-->]
-
[<-->]
<-->
Ready

```


ON GOTO

Syntax

ON expression GOTO linenum [,linenum] ...

where:

expression is a numeric expression.

Examples of Syntax

ON I% GOTO 100,200

ON INT(4*RND+1) GOTO 300,310,320,330

Purpose

ON GOTO branches to one of several possible line numbers depending on the value in the expression.

Comments

- When ON GOTO is executed, the numeric expression is evaluated and truncated to an integer if necessary.

If the value is 1, PRO/BASIC branches to the first line number in the list, if the value is 2, PRO/BASIC branches to the second line number in the list, and so forth. For example:

```
80 ON GROUP GOTO 300,400,500
```

when:

GROUP < 1	PRO/BASIC signals an error
GROUP = 1	branches to line 300
GROUP = 2	branches to line 400
GROUP = 3	branches to line 500
GROUP > 3	PRO/BASIC signals an error

- If the expression is less than one or greater than the number of line numbers in the list, PRO/BASIC displays an error message. Negative values and 0 cause an error.
- It is a good practice to check the value of the expression before the ON GOTO statement is executed.

Example

```

100 INPUT I% \ IF I%=0% THEN 300
110 ON I% GOTO 200,210,220
200 PRINT 'At line 200' \ GOTO 100
210 PRINT 'At line 210' \ GOTO 100
220 PRINT 'At line 220' \ GOTO 100
300 END
RUN
? 2
At line 210
? 1
At line 200
? 3
At line 220
? 0
Ready

```

The function of the ON statement is equivalent to a series of IF statements. The preceding program could have been written:

```

100 INPUT I% \ IF I%=0% THEN 300
110 IF I%=1% THEN 200
120 IF I%=2% THEN 210
130 IF I%=3% THEN 220
140 PRINT 'Error — ON statement out of range' \ STOP
200 PRINT 'At line 200' \ GOTO 100
210 PRINT 'At line 210' \ GOTO 100
220 PRINT 'At line 220' \ GOTO 100
300 END

```

OPEN

Syntax

```
[FOR INPUT ]
OPEN filespec [FOR OUTPUT ] AS FILE #channelnum [,VIRTUAL]
```

where:

filespec is a string expression that specifies a file.

#channelnum is a number or an expression specifying a valid channel number (1-15).

VIRTUAL indicates that the file is a virtual array.

Examples of Syntax

```
OPEN NAME$ AS FILE #1
OPEN 'TT:' FOR INPUT AS FILE #3
OPEN 'VIRT.DAT' FOR OUTPUT AS FILE #2, VIRTUAL
```

Purpose

OPEN allows input and output from an existing file or creates a new file.

Comments

- Specify FOR INPUT to open an old file. Specify FOR OUTPUT to open a new file. If you do not specify FOR INPUT or FOR OUTPUT, PRO/BASIC:
 - opens a file if the file exists.
 - creates a file if the file does not exist.
- PRO/BASIC supports two types of files:
 1. Document (sequential) files can be on disk or on peripheral devices such as printers. Use the INPUT # statement and the PRINT # statement to read from and write to a document file. Remember that you cannot simultaneously read from and write to the same document file.

2. Virtual array files exist only on disk (see the DIM # statement for detailed information). You can read from or write to a virtual array file exactly as if it were in memory.

- Use VIRTUAL to create a virtual array in a file. Specify FOR INPUT to read and write data from an existing file. Specify FOR OUTPUT to read and write data from a newly created file.

If you do not specify FOR INPUT or FOR OUTPUT with VIRTUAL file organization, PRO/BASIC opens an existing file to read and write, or creates a new file to read and write if the file does not exist.

NOTE: If you open an existing document file with FOR OUTPUT, closing the file creates a new version of the file.

- A file opened FOR INPUT must exist.
- The valid range of channel numbers is 1 - 15. Channel 0 (the default) is the keyboard and display screen and is always open.

Example

The example below types the contents of a specified file. In this case, file ABC.BAS, with 2 lines, is printed.

```
10 PRINT 'File to Type';
20 LINPUT FILE$
30 OPEN FILE$ FOR INPUT AS FILE #1
40 LINPUT #1,LIN$ \PRINT LIN$ \GOTO 40
SAVE TYPER
RUN TYPER
File to Type? ABC.BAS
SET NO DOUBLE
10 PRINT 'abc'
20 END
```

Error 11 at line 40: End of file

PRINT

Syntax

PRINT [[#channelnum,]print list]

where:

print list	consists of one or more valid expressions to be printed, separated by commas or semicolons and optionally followed by a comma or semicolon.
#channelnum	is a valid channel number (0 - 15) or an expression that specifies a valid channel number. Channel 0 is the default and indicates the display screen.

Examples of Syntax

```
PRINT 2+2
```

```
600 PRINT 'The average is:';TOTAL% / ITEMS%
```

```
50 PRINT #4%,TTLINE$;
```

Purpose

PRINT displays data from your program on the screen or stores data in a document file.

Comments

- Use the PRINT statement to display the value of an expression.
For example:

```
10 A = 45\B = 55
20 PRINT A+B
30 END
RUN
100
Ready
```

- If no channel is specified, PRO/BASIC prints to the screen (channel 0). Printing to channel 0 does not require an OPEN statement beforehand because the display screen and keyboard are always open for input/output.
- To print to a channel other than 0, a document file must be open on that channel.

- A line on a screen consists of a series of print zones, each of 14 spaces. Commas and semicolons between expressions determine where on a line program data is printed. A comma causes PRO/BASIC to print program data at the beginning of a print zone and advance one print zone for each comma between expressions. When a semicolon separates expressions, PRO/BASIC does not print additional spaces between them. For example:

```

10 PRINT 10,20
20 PRINT 10,,20
30 PRINT 10;20
RUN
10                20
10                20
10 20
READY

```

You can advance print zones by using consecutive commas, as in line 20 in the preceding example.

- If you end a PRINT list with a comma or a semicolon, the cursor remains on the same line. Otherwise, the cursor moves to the beginning of the next line. For example:

```

10 PRINT 'AAA'
20 PRINT 'BBB';
30 PRINT 'CCC'
40 PRINT 'DDD',
50 PRINT 'EEE'
RUN
AAA
BBBCCC
DDD      EEE
Ready

```

- When printing numeric fields, PRO/BASIC precedes each number with a space or a minus sign, and follows it with a space. A PRINT statement with no expressions simply goes on to the next line. If an expression returns a string, PRO/BASIC prints no extra spaces at all. For example:

```

PRINT 'Hel'; 'lo'
Hello
Ready

```

- The TAB function is a special library function designed to be used with the PRINT statement. It moves the cursor directly to a specific character position. (Refer to the TAB function in Chapter 6.)
- To print to a line printer:
 1. to print a program file—use the SAVE command, followed by the line printer specification, to print the program currently in memory.

SAVE LP:

2. to print a print list—specify:

OPEN 'LP:' FOR OUTPUT AS FILE #channelnum

You can then print to hardcopy by specifying:

PRINT #channelnum, print list

where print list is any valid expression. It is possible to print a file by opening the file to be printed, opening the line printer as the output file, and then using the LINPUT# AND PRINT# statements to print each line from the file to the printer. The following program statements would print a column head and all the values in a file.

```

300 ON ERROR GOTO 380
310 OPEN 'FROM_FILE' FOR INPUT AS FILE #1
320 OPEN 'LP:' FOR OUTPUT AS FILE #2
330 PRINT 'COLUMN HEAD'
340 PRINT
350 LINPUT #1,TO_PRINT
360 PRINT #2,TO_PRINT
370 GOTO 350
380 PRINT 'DONE' \ END

```

PRINT USING

Syntax

PRINT [#channelnum,] USING format string, print list

where:

#channelnum	is a numeric expression specifying a valid channel number (1 - 15). Channel 0 is the default and indicates the display screen.
format string	is a string expression that describes the format used for printing.
print list	is one or more valid expressions to be printed, separated by commas or semicolons.

Examples of Syntax

```
100 PRINT USING FS3$,N%,NAME$,TAXRATE
```

```
100 PRINT #2% USING '**###,###.##-',-12.34
```

```
100 PRINT #3% USING "## 'E 19##",10,'January',81
```

Purpose

PRINT USING controls the appearance and location of data passed from your program to the display screen or a document file.

Comments

- The format string contains field descriptors and string data:

A field descriptor holds a place for formatted data and controls how it is output.

Any other character data in a format string is printed unchanged with the formatted output.

Refer to the format string (within double quotation marks) in the last example under Syntax in this section. The two pound signs (##) are a field descriptor reserving space for two digits: in this case the first data item in the list is the value 10. The 'E' in the format string is a field descriptor for handling strings: in this case, the string is 'January'.

The two digits (19) are string data, they are printed as they appear. There are also single spaces before the 'E' and the 19.

- There are field descriptors for numeric output and for string output. The field descriptors that control numeric output are:

The pound sign (#), which reserves one position for a digit or a minus sign. For example:

```
PRINT USING 'You have#### points.',-6
You have -6 points.
```

The period (.), which inserts a decimal point. The number of reserved positions on either side of the period determines where the decimal point appears in the output. (See SET RADIX in this chapter.) For example:

```
PRINT USING '###.###',2.22
2.220
```

The comma (,), which reserves one position (for a comma or a digit) and inserts commas before every third significant digit to the left of the decimal point. In the format string, place a comma anywhere to the left of the decimal point (if present) and to the right of the dollar sign or asterisk (if present). (See SET SEPARATOR in this chapter.) For example:

```
PRINT USING '####,####.##',3333333
3,333,333.00
```

The hyphen (-), which reserves one position (for a sign) and prints the numbers in the list with a trailing minus sign if negative or a trailing space character if positive. The hyphen must be the last character in a field in a format string. For example:

```
PRINT USING '###-',2,-17,100
2
17-
100
```

Two dollar signs (\$\$), which reserve two positions (for a dollar sign and a digit) and print the number with a dollar sign immediately to the left of the most significant digit. This produces floating-dollar-sign format. (See SET CURRENCY in this chapter.) For example:

```
PRINT USING '$$#####.##',250
$250.00
```

Two asterisks (**), which reserve two positions (for two digits) and fill the left side of the numeric field with asterisks. This produces asterisk-filled format. For example:

```
PRINT USING '*****.##',250
*****250.00
```

Four carets (^^^^), which reserve four positions (for an exponent) and print the numbers in the list in E notation format. The carets must be the rightmost characters in the format string. See Section 2.4.2 for a discussion of E notation. For example:

```
PRINT USING '###.##^',5,1000
500.00E-02
100.00E+00
```

PRO/BASIC prints the number in the format field in the furthest left position (left-justified) and adjusts the exponent to compensate, except when printing zero. When printing zero, PRINT USING prints leading spaces and zeros, if necessary, and the exponent E+00.

The field descriptors that control string output are:

A single quotation mark ('), which reserves one position (for one character) and marks the beginning of a character field. Therefore, a literal string containing a character field must be enclosed in double quotation marks. If the expression is too large for the field, PRO/BASIC truncates the expression from the right.

For example:

```
PRINT USING "-> '<-', 'PLASTIC', 'MAN'
-> P<-
-> M<-
```

An "L" or "l" reserves one position and prints the expression in the furthest left position of the field. The number of L's (plus one for the single quotation mark) determines the size of the field. If the expression is too large for the field, PRO/BASIC truncates the expression from the right. For example:

```
PRINT USING "->'LLLL<-', 'PLASTIC', 'MAN'
-> PLAST<-
-> MAN  <-
```

An "R" or "r" reserves one position and prints the expression right-justified. The number of R's (plus one for the single quotation mark) determines the size of the field. If the expression is too large for the field, PRO/BASIC truncates the expression from the right.

For example:

```
PRINT USING "->'RRRR <-', 'PLASTIC', 'MAN'
-> PLAST <-
->  MAN <-
```

A "C" or "c" reserves one position and prints the expression centered. If the expression can not be centered exactly, it is offset one character to the left. The number of C's (plus one for the single quotation mark) determines the size of the field. If the expression is too large for the field, PRO/BASIC truncates the expression from the right. For example:

```
PRINT USING "->'CCCC <-', 'PLASTIC', 'MAN'
-> PLAST <-
->  MAN  <-
```

An "E" or "e" reserves one position and expands the field to hold the entire expression. The number of E's (plus one for the single quotation mark) determines the size of the field. If the expression is smaller than the field, PRO/BASIC prints the expression left-justified. For example:

```
PRINT USING "->'EEEE <-', 'PLASTIC', 'MAN'
-> PLASTIC <-
->  MAN    <-
```

Comments

Below is a summary of the symbols used with PRINT USING:

Format Characters for Numeric Fields

#	number sign	Reserves place for one digit or minus sign.
.	decimal point	Determines location of decimal point.
,	comma	Reserves one position and inserts commas before every third digit left of the decimal.
**	two asterisks	Reserves two positions and fills the field left of the number with asterisks.
-	hyphen	Reserves one position and prints negative numbers with trailing minus sign.
\$\$	two dollar signs	Reserves two positions and prints a dollar sign before the first digit.

******** four circumflexes Reserves four positions for exponent and prints number in E format.

Format Characters for String Fields

' single quote Reserves one position for character and starts string field.

L Reserves one position for character and left-justifies the string.

R Reserves one position for character and right-justifies the string.

C Reserves one position for character and centers the string in the field.

E Reserves one position for character and expands the field to print the string.

- ☐ To print to a file other than 0, a document file must be open on that channel.
- ☐ Commas and semicolons separate items to be printed but do not specify print zones. If a comma or a semicolon follows the last expression in the list, a line terminator is not generated, that is, the cursor remains on the current line. For example:

```
PRINT USING '###',1,2,3 \ REM cursor moves to new line
PRINT USING '###',1,2,3 \ REM cursor remains on line
```

- ☐ If there is more data to print after expressions are entered to the format string, PRO/BASIC uses the format string again from the beginning and starts on a new line.
- ☐ You can use PRINT USING to align columns of numbers. For example:

```
PRINT USING '###.##-',100,-23.4,-.175,96.32
100.00
23.40-
0.17-
96.32
```

- You can use PRINT USING to insert blanks and other printing characters at precise locations. For example:

```
PRINT USING "Phone: (###) ###'####",617,123,'-',4567
Phone: (617) 123-4567
```

- To print negative numbers in floating-dollar-sign or asterisk-filled format, you must use trailing minus sign format. For example:

```
PRINT USING '***#####.##-',-12.34
*****12.34-
```

- E notation format cannot be combined with any other numeric format in a single field.
- Character control symbols (R, L, C, and E) cannot be combined in a single field.
- If a numeric field does not contain enough positions for the given number, PRO/BASIC prints a percentage sign followed by the number in normal PRINT format. For example:

```
PRINT USING '#', 22
% 22
```

- If a fractional value contains more digits than there are spaces provided in the format string, the value is rounded according to the first digit after the end of the format string. For example:

```
10 DECLARE DOUBLE N
20 INPUT N
30 IF N=0 THEN 50
40 PRINT USING '.##',N \ GOTO 10
50 END
RUN
? .223
.22
? .229
.23
?
```

Error 28 at line 20: INTERRUPT—DO keys entered

If a character field (other than the E format character) does not contain enough format string symbols for the given string PRO/BASIC truncates the string to fit and discards the excess characters without warning.

PROGRAM

Syntax

PROGRAM *programe* [(*variable*[,*variable*] ...)]

where:

programe is any combination of alphabetic and numeric characters, up to 32 maximum.

variable is a variable of the same data type as the value(s) specified in the CHAIN statement.

Examples of Syntax

PROGRAM MOD3(A,B%,C\$)

PROGRAM PEACHESNCREAM(TABLE%,,)

PROGRAM ABC123(LIST\$(),N%)

Purpose

PROGRAM is used when the chained-to program is to receive values from the chaining program.

Comments

- The **PROGRAM** statement should be the first statement in the chained-to program. Enclose the variables receiving values in parentheses. The names of the variables receiving values need not be the same as those passing values from the CHAIN statement.
- The data types of the values passed and the variables receiving them must agree—that is, string values can only be passed to string variables, numeric values to numeric variables. However, when you pass numeric values, integer, single or double precision, the value is converted to agree with the data type of the variable. Arrays, on the other hand, must agree precisely—that is, a single precision value must be assigned to a variable of single precision.
- To specify an array in a **PROGRAM** statement, use parentheses containing only the appropriate number of commas to indicate the number of dimensions. For example, in the second example under Syntax in this section, TABLE% has two dimensions; in the third example, LIST\$ has one dimension.

- ❑ You can pass an array, an element from an array, or an element of a virtual array between programs. You can not pass an entire virtual array.
- ❑ A warning message is displayed if the number of variables listed in the CHAIN statement is not equal to the number of variables listed in the PROGRAM statement.
- ❑ Use only one PROGRAM statement. Any PROGRAM statements appearing after the first PROGRAM statement are ignored by PRO/BASIC.
- ❑ Although the PROGRAM statement must be followed by a valid filename, (prognam), that value is ignored.
- ❑ See the CHAIN statement for more information and examples.

RANDOMIZE

Syntax

RANDOMIZE

Purpose

RANDOMIZE causes **RND**, the random number function, to create a new series of random numbers.

Comments

- Without the **RANDOMIZE** statement, the random number function, **RND**, always provides the same random number sequence. **RANDOMIZE** reshuffles the random number sequence to create a different series of numbers.
- When a program is run or chained to, execute **RANDOMIZE** to create different random number patterns.

Example

Two executions of this program produce the same sequence of numbers.

```
20 FOR I% = 1 TO 6
30   PRINT RND;
40 NEXT I% \ PRINT
RUN
.76308 .179978 .902878 .88984 .387011 .475943
Ready
RUN
.76308 .179978 .902878 .88984 .387011 .475943
Ready
```


Two executions of this program, which includes the RANDOMIZE statement, produce different number sequences each time.

```

10 RANDOMIZE
20 FOR I% = 1 TO 6
30   PRINT RND;
40 NEXT I% \ PRINT
RUN
.243497 .127635 .617749 .321294 .442961 .888113
Ready
RUN
.207654 .160614E-01 .852302 .111175 .849465 .472174
Ready

```

READ

Syntax

`READ variable[,variable] ...`

where:

`variable` (or array element) is numeric or string

Examples of Syntax

```
100 READ A,B(10%),C$,D%(10%),E$,F$(10%)
```

```
200 READ A$,B$,C$,D$,E$,F$,
```

Purpose

READ assigns values from DATA statements to variables.

Comments

- ❑ The READ and DATA statements allow your program to contain information that otherwise would have to be typed in or read in from a file.
- ❑ When PRO/BASIC executes a READ statement, it assigns a value to each variable in the READ statement, from left to right. Values are assigned from DATA statements from left to right and from lower- to higher-numbered DATA statements. One READ can access values from one DATA statement or several DATA statements. More than one READ statement can be used.
- ❑ The data type of the value must agree with the data type of the variable it is assigned to—that is, you can only assign numeric values to numeric variables. However, when you assign numeric values, integer, single or double precision, the value is converted to agree with the data type of the variable in the READ statement.
- ❑ If there are more reads than data values, an error message is displayed. If there are fewer reads than there are values in a DATA statement, subsequent reads will start from the last value read.

- There must be at least one DATA statement if a READ statement is used in a program.
- Refer to the DATA and RESTORE statements in this chapter for more on the use of these statements.

Example:

```
110 READ I%,R,A$,B$
120 PRINT I%,R,A$,B$
130 DATA 335,209.38,'disk drive',terminal
RUN
335          209.38          disk drive          terminal
Ready
```

REM

Syntax

REM [text]

where:

text is any string of printing characters

Example of Syntax

```
700 REM *** This subroutine performs a bubble sort ***
```

Purpose

REM permits you to include comments, notes, or messages in your program.

Comments

- A REM statement must be the last statement on a line, since anything that follows it is assumed to be a comment.

Example

```
740 FOR I=1 TO N-1 \ REM Sort one less element each time
```

RESTORE

Syntax

RESTORE [linenum]

where:

linenum is the line number of a DATA statement.

Examples of Syntax

```
100 RESTORE
```

```
100 RESTORE 150
```

Purpose

RESTORE causes values in a DATA statement to be reread from a specified line number.

Comments

- ❑ Use the RESTORE statement to select data items starting from a specified line number. If you do not specify a line number, RESTORE selects the lowest-numbered DATA statement in your program.
- ❑ See the READ and DATA statements in this chapter.

Example

```
100 DATA 1
110 DATA 2
120 DATA 3
130 READ A%,B%,C% \ PRINT A%;B%;C%
140 RESTORE
150 READ A%,B%,C% \ PRINT A%;B%;C%
160 RESTORE 110
170 READ A%,B% \ PRINT A%;B%
RUN
1 2 3
1 2 3
2 3
Ready
```

RESUME

Syntax

RESUME [linenum]

where:

linenum is the number of the line where program execution continues after processing an error.

Example of Syntax

RESUME

Purpose

RESUME marks the end of an error-handling routine and returns program control to a specified line.

Comments

- If a line number is specified, program control returns to the first statement on that line. If no line number is specified, program control is returned to the first statement on the line where the error occurred.
- When PRO/BASIC executes a **RESUME** statement, it clears the error condition.
- This statement is used with user-defined error handling. In the absence of user-defined error handling, PRO/BASIC detects and handles errors by itself. (See Chapter 3 for more on error handling.)
- The **RESUME** statement is not valid in immediate mode.

Example

```
10    ON ERROR GOTO 19000
.
program statements
.
19000 PRINT ERT$(ERR) \ REM THIS IS THE ERROR HANDLER
19010 PRINT "RESUME?";
19020 INPUT A$
19030 IF A$='Y' THEN RESUME ELSE RESUME 32767
32767 END
```

RETURN

Syntax

RETURN

Example of Syntax

RETURN

Purpose

RETURN terminates a subroutine and transfers control to the statement immediately following a GOSUB.

Comments

- ☐ The last line in a subroutine must be a RETURN statement.
- ☐ A RETURN statement can be followed by the ELSE portion of an IF statement (conditional RETURN) or a REM statement.
- ☐ To execute a subroutine without branching from a GOSUB or ON GOSUB statement causes a fatal error and displays the error message, "Return without GOSUB."
- ☐ See the GOSUB and ON GOSUB statements in this chapter for more information and examples.

SET CURRENCY

Syntax

SET CURRENCY expression

where:

expression is a string expression, a symbol which identifies a currency.

Example of Syntax

SET CURRENCY "\$"

Purpose

SET CURRENCY specifies the character that identifies a currency and allows the character to 'float,' or appear at a variable location, depending upon the size of the currency value.

Comments

- ❑ The specified value is substituted for the dollar sign (\$) in the PRINT USING string.
- ❑ If no currency symbol is specified with SET CURRENCY, the dollar sign (\$) is used.

Example

```

10 SET CURRENCY 'fr'
20 INPUT A
30 PRINT USING '$$###'###.##',A
40 GOTO 20
RUN
? 4321
fr 4,321.00
? 45
fr 44.00
? 2126
fr 2,126.00
?
Error 28 at line 20: INTERRUPT-DO keys entered

```


SET RADIX

Syntax

SET RADIX expression

where:

expression is a string expression, a punctuation mark inserted to the currency value.

Example of Syntax

SET RADIX "."

Purpose

SET RADIX specifies the character that separates the fractional part of a number.

Comments

- ☐ The character specified is substituted for the decimal point in the PRINT USING string.
- ☐ If no character is specified with SET RADIX, the decimal point (.) is used.

Example

```
10 SET RADIX ""
20 INPUT A
30 PRINT USING '$$###.##',A
40 GOTO 20
RUN
? 4
$4*00
?
```

Error 28 at line 20: INTERRUPT—DO keys entered

SET SEPARATOR

Syntax

SET SEPARATOR expression

where:

expression is a string expression, a character that separates groups of digits in a large number.

Example of Syntax

SET SEPARATOR " "

Purpose

SET SEPARATOR specifies the character used to separate groups of digits in a large number.

Comments

- ☐ The character specified is substituted for the comma in the PRINT USING string.
- ☐ If no character is specified with SET SEPARATOR, a comma (,) is printed.

Example

```
10 SET SEPARATOR '!'  
20 INPUT A  
30 PRINT USING '###,###.##',A  
40 GOTO 20  
RUN  
? 1234.56  
!1234.56  
?  
Error 28 at line 20: INTERRUPT—DO keys entered
```

STOP

Syntax

STOP

Purpose

STOP suspends program execution, reports the last line number executed, and waits for input.

Comments

- ❑ The STOP statement allows you to halt your program at any intermediate point in processing to examine and modify variables and program statements. Program processing can be resumed by entering the CONTINUE command or an immediate mode GOTO or GOSUB statement.
- ❑ The STOP statement does not close files; therefore, if your program opens an output file, you must either allow it to complete execution or execute a CLOSE statement to preserve the contents of the file.

Example

```

10 REM program calculates the sum of the first N integers
20 PRINT 'this program calculates the sum of the first N integers'
30 PRINT 'enter N' \ INPUT N
40 FOR I=1 TO N
50   S=S+1
60   T=T+S
70 STOP
80 NEXT I
90 PRINT 'sum is';T
RUN
this program calculates the sum of the first N integers
enter N
? 2

```

```

STOP at line 70
CONT

```

```

STOP at line 70
SHOW

```

```

N = 2
I = 2
S = 2
T = 3

```

```

Program size (bytes):  207
Number of lines:      9
Number of Symbols:    34
Free Memory:          10343

```

```

CONT
sum is 3
Ready

```

6

Library Functions

Chapter 6

Library Functions

This chapter lists the library functions of PRO/BASIC.

A function is a series of statements that accepts one or more values (called arguments), performs an operation and returns a result. All numeric functions return a single precision number.

To use a function, specify the function name and the arguments.

The presentation of each library function consists of four parts:

- ☐ *Syntax*—the syntax required by the function
- ☐ *Purpose*—the result the function returns
- ☐ *Comments*—explanation and suggestions for use
- ☐ *Example*—typical or explanatory example

ABS

Syntax

ABS (expression)

where:

expression is a numeric expression.

Example of Syntax

A%=ABS(B% * -2%)

Purpose

ABS returns the absolute value of a numeric expression.

Comments

- ☐ Absolute value is the magnitude of a number, without regard for its sign.
- ☐ You can use ABS to find the difference between variables with unknown values.

Example

```

10 REM This program asks for two values to be entered
11 REM and then displays the difference between them
20 PRINT 'POSITION 1'
30 INPUT X1
40 PRINT 'POSITION 2'
50 INPUT X2
60 PRINT 'THE DIFFERENCE IS';ABS(X1-X2)
70 END
RUN
POSITION 1
? -2.4
POSITION 2
? 3.5
THE DIFFERENCE IS 5.9

```

ASCII

Syntax

ASCII (expression)

where:

expression is a string expression.

Example of Syntax

ASCII (BNAME\$)

Purpose

ASCII returns the decimal value of a DEC Multinational character.

Comments

- ASCII allows you to identify and manipulate characters by numeric value, rather than by string value.
- If the string expression contains more than one character, the ASCII function returns the decimal value of the first (leftmost) character in the string.
- The ASCII function is the opposite of the CHR\$ function, which converts a decimal value into its character equivalent.
- See Appendix A for a complete list of the DEC Multinational characters and their decimal values.

Example

```

100 REM *** Program to Demonstrate ASCII Function **
110 REM
120 PRINT 'Line'; \ LINPUT TLINE$ \ PRINT
130 FOR I=1 TO LEN(TLINE$)
140   CH=ASCII(MID$(TLINE$,I,1))
150   IF CH<>32 THEN PRINT USING '####', CH; ELSE PRINT
160 NEXT I \ PRINT
170 END
RUN

```

Line? The quick brown fox jumped over the lazy white dog.

```

84 104 101
113 117 105 99 107
98 114 111 119 110
102 111 120
106 117 109 112 101 100
111 118 101 114
116 104 101
108 97 122 121
119 104 105 116 101
100 111 103 46

```

Ready

ATN**Syntax**

ATN (expression)

where:

expression is a numeric expression representing the tangent of an angle.

Example of Syntax

A=ATN(1)

Purpose

ATN returns the arctangent of the specified tangent in radians.

Comments

The result is expressed in radians in the range $-\pi/2$ to $\pi/2$.

Example

```
10 REM *** Program to Print the Arcsine of an Angle ***
20 PRINT "angle in radians"; \INPUT ANGLE
30 PRINT "arcsine =";ATN(ANGLE/SQR(1-ANGLE^2))
40 END
RUN
angle in radians? .5
arcsine = .523599
Ready
```

CCPOS

Syntax

CCPOS (channelnum)

where:

channelnum is a numeric expression specifying a valid channel number (0 - 15) for a document file. The default is 0, indicating the display screen.

Example of Syntax

C%=CCPOS(2%)

Purpose

CCPOS returns the current cursor position in the current line on the specified channel.

Comments

You can use CCPOS to find out if there is enough room on the current line for what you want to print.

Example

```
10 FOR I=1 TO 10
20   FOR J=1 TO I
30     PRINT ' ';
40   NEXT J
50   PRINT CCPOS(0)
60 NEXT I
70 END
```

RUN

```
1
 2
  3
   4
    5
     6
      7
       8
        9
         10
```

Ready

CHR\$

Syntax

CHR\$ (expression)

where:

expression is a numeric expression which is the decimal value of a DEC Multinational character.

Example of Syntax

BELL\$=CHR\$(7%)

Purpose

CHR\$ returns the DEC Multinational character associated with the provided decimal value.

Comments

- ☐ The **CHR\$** function accepts any number through 255 and returns the associated character
- ☐ **CHR\$** is the opposite of the **ASCII** function, which converts character data to numeric data.
- ☐ See Appendix A for a complete list of the DEC Multinational characters and their decimal equivalents.

Example

This program uses the CHR\$ function to print the DEC Multinational Character Set.

```

10 FOR J=0 TO 7
20   IF J=0 OR J=4 THEN GOTO 80
30   FOR I=0 TO 31
40     C=I+J*32
50     PRINT CHR$(C)+' ';
60   NEXT I
70   PRINT
80 NEXT J
RUN

```

```

! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
? i f f f f f f f f f f f f f f f f f f f f f f f f f f f f f
A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A
a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
Ready

```

COS

Syntax

COS (expression)

where:

expression is a numeric expression representing the angle in radians.

Example of Syntax

C = COS(N)

Purpose

COS returns the cosine of an angle.

Comments

- ☐ The argument is expressed in radians.
- ☐ The result is between -1 and 1.
- ☐ COS and SIN may return inaccurate results when handling large numbers.
- ☐ You can use the COS function (with the SIN function) to determine the tangent of an angle.

Example

```

100 REM *** Program to Print the Tangent of an Angle ***
110 REM
120 DEF FNTAN(ANGLE)=SIN(ANGLE)/ COS(ANGLE)
130 REM—Angle is input in radians in line 140
140 PRINT 'Angle';\ INPUT A
150 PRINT 'Tangent =';FNTAN(A)
160 END
RUN

Angle? .5
Tangent = .546302

Ready

```

DATE\$

Syntax

DATE\$ (0%)

Example of Syntax

TODAY\$=DATE\$(0%)

Purpose

DATE\$ returns a string containing the current date.

Comments

- DATE\$ returns the date in the format:

dd-mmm-yy

where:

dd is the day.

mmm is the first three letters of the month.

yy is the year.

- The only valid argument for the DATE\$ function is 0. An argument is included only for compatibility with other versions of BASIC.
- If the day of the month is less than ten, DATE\$ returns a leading 0 in the first character position.

Example

PRINT DATE\$(0%)

12-Jun-81

Ready

EDIT\$

Syntax

EDIT\$ (string expression, numeric expression)

where:

string expression	is a string or a variable containing a string to be edited.
numeric expression	is the value 2,8,16,32,64, or 128 or a sum of these values. The value indicates which editing operations to do.

Examples of Syntax

```
LINE$ = EDIT$(LINE$,32%)
```

```
PRINT EDIT$(NAME$,128+16+8)
```

Purpose

EDIT\$ performs one or more editing operations on a string.

Comments

- ☐ You can select editing operations by specifying the corresponding value(s) from the following table:

Value	Edit Operation
2	Discards all spaces and tabs
8	Discards leading spaces and tabs
16	Converts multiple spaces and tabs to a single space
32	Converts lower case letters to upper case
128	Discards trailing spaces and tabs

- ☐ You can perform several editing operations at once by adding up the values representing edit operations.
- ☐ The **EDIT\$** function is useful for “cleaning up” input lines.

Examples

In the following example, the value 2 is specified in the EDIT\$ function so that all spaces and tabs will be removed.

```
10 S$ = '   aaa bbb   ccc   ddd   eee   fff'
20 PRINT EDIT$(S$,2)
RUN
aaabbbcccddeeefff
Ready
```

In this example, the value 48 specified in the EDIT\$ function represents the sum of the values of two edit operations: conversion of multiple spaces and tabs to a single space (16), and conversion of lowercase to uppercase (32))

```
10 S$ = '   aaa bbb   ccc   ddd   eee   fff'
20 PRINT EDIT$(S$,48)
AAA BBB CCC DDD EEE FFF
Ready
```

ERL

Syntax

ERL

Example of Syntax

```
IF ERL=2000 THEN RESUME 2000
```

Purpose

ERL returns the line number of the statement executing when the last error occurred.

Comments

- ☐ Use IF statements to test the contents of ERL to determine where the error occurred.
- ☐ The ERL function is to be used after an error occurs and before a RESUME statement is executed.
- ☐ The ERL function is used for user-defined error handling. In the absence of user-defined error handling, PRO/BASIC itself detects and handles errors. (Refer to Chapter 3, Section 5 for more on error handling.)

ERR

Syntax

ERR

Example of Syntax

```
IF ERR=9 THEN RESUME 730
```

Purpose

ERR returns the number of the latest error.

Comments

- ☐ Use IF statements to test the contents of ERR to determine which error occurred.
- ☐ The ERR function is to be used after an error occurs and before a RESUME is executed.
- ☐ The ERR function is used for user-defined error handling. In the absence of user-defined error handling, PRO/BASIC itself detects and handles errors. (Refer to Chapter 3, Section 5 for more on error handling.)

ERT\$

Syntax

ERT\$(expression)

where:

expression is a numeric expression.

Example of Syntax

```
2020 ERROR_TX$=ERT$(11)
```

Purpose

ERT\$ returns the error message text associated with the specified error number.

Comments

- ❑ The error number must be between 0 and 255, inclusive.
- ❑ The ERT\$ function can be used at any time to return the text associated with a specified error number.
- ❑ The ERT\$ function is used for user-defined error handling. In the absence of user-defined error handling, PRO/BASIC itself detects and handles errors. (Refer to Chapter 3, Section 5 for more on error handling.)

EXP**Syntax**

EXP (expression)

where:

expression is a numeric expression.

Example of Syntax

A=EXP(B)

Purpose

EXP returns e, an algebraic constant approximately equal to 2.71828, raised to the power specified.

Comments

- The value of e raised to a power must be in the range -88 to 88, inclusive.
- The EXP function is the inverse of the LOG function.

Example

```
10 REM-PROGRAM TO DISPLAY HYPERBOLIC SIN OF X FOR 1-10
20 DEF FNSINH(X)=(EXP(X)-EXP(-X))/ 2
30 FOR I%=1 TO 10
40 PRINT 'SINH',I%,FNSINH(I%)
50 NEXT I%
60 END
RUN
```

SINH	1	1.1752
SINH	2	3.62686
SINH	3	10.0179
SINH	4	27.2899
SINH	5	74.2032
SINH	6	201.713
SINH	7	548.316
SINH	8	1490.48
SINH	9	4051.54
SINH	10	11013.2

FIX

Syntax

FIX (expression)

where:

expression is a numeric expression.

Example of Syntax

N = FIX(N)

Purpose

FIX truncates a real number at the decimal point.

Comments

- ☐ **FIX** returns a real number.
- ☐ When handling negative numbers **FIX** may return a value whose absolute value is smaller than the argument. For example:

```
10 READ A,B
20 PRINT A;FIX(A)
30 PRINT B;FIX(B)
40 DATA 9.9,-9.9
RUN
  9.9                9
 -9.9               -9
Ready
```

In the example above, the first line of results shows the value 9 returned from 9.9. The second line of results shows the value -9 returned from -9.9. Note that when handling a negative number, **FIX** returned a value larger than the argument.

- ☐ Refer to the Example in the **INT** function in this chapter.

Example

```
10 REM FIX Returns the whole number portion of a real number.
20 FOR I=1 TO 5
30 READ N
40 PRINT FIX(N)
50 NEXT I
60 DATA 50.2, -2.12, .056, 17.63, 100.5
70 END
RUN
    50
   -2
    0
   17
  100
Ready
```


INT

Syntax

INT (expression)

where:

expression is a numeric expression

Example of Syntax

N = INT(N)

Purpose

INT returns the real number value of the largest integer less than or equal to the expression provided.

Comments

- The INT function always returns the value of the largest integer that is less than or equal to the argument. As a result, when handling negative numbers INT can return a value whose absolute value is larger than that of the expression provided. For example:

```
10 READ A,B
20 PRINT A;INT(A)
30 PRINT B;INT(B)
40 DATA 9.9,-9.9
RUN
9.9          9
-9.9         -10
```

In the example above, the first line of results shows the value 9 returned from 9.9. The second line of results shows the value -10 returned from 9.9. Note that when handling a negative number, INT returns a value smaller than the argument.

- When handling positive numbers, the FIX and the INT functions produce the same results.
- The INT function does not round off numbers. However, you can use the INT function to define a ROUND function.

Example

```

100 REM *** Demonstrate rounding off with INT ***
110 DEF FNROUND(N)=INT(N + 0.5)
120 PRINT 'INT','ROUND','FIX'
125 PRINT
130 FOR I%=10% TO -10% STEP -2%
140 TEMP=I%+I%/9
150 PRINT TEMP,INT(TEMP),FNROUND(TEMP),FIX(TEMP)
160 NEXT I% \ END
RUN

```

	INT	ROUND	FIX
11.1111	11	11	11
8.88889	8	9	8
6.66667	6	7	6
4.44444	4	4	4
2.22222	2	2	2
0	0	0	0
-2.22222	-3	-2	-2
-4.44444	-5	-4	-4
-6.66667	-7	-7	-6
-8.88889	-9	-9	-8
-11.1111	-12	-11	-11

Ready

LEN

Syntax

LEN (expression)

where:

expression is a string expression.

Examples of Syntax

```
PRINT LEN(NAME$)
IF LEN(A$)=20 THEN 400
```

Purpose

LEN returns the number of characters in a string expression.

Comments

If the string expression is null, LEN returns 0.

Example

```
10 REM PROGRAM RIGHT-ALIGNS PRINT OUTPUT
20 READ N
30   FOR J=1 TO N
40     READ P$
50     PRINT TAB(20-LEN(P$));P$
60     PRINT
70   NEXT J
80 DATA 5,'EMPLOYEE NAME','JOB NO.','JOB CATEGORY','REGULAR HOURS'
90 DATA 'OVERTIME HOURS'
RUN
```

EMPLOYEE NAME

JOB NO.

JOB CATEGORY

REGULAR HOURS

OVERTIME HOURS

Ready

LOG

Syntax

LOG (expression)

where:

expression is a numeric expression greater than zero.

Example of Syntax

A=LOG(B*C)

Purpose

LOG returns the natural logarithm of the specified number.

Comments

- Natural logarithms are exponents of the base e, where e is a mathematical constant approximately equal to 2.71828. That is, the natural logarithm of a number n is the power to which e must be raised to equal n. For example, the natural logarithm of 100 is 4.60517, because e raised to the power of 4.60517 equals 100.
- The LOG function is the inverse of the EXP function.

Example

```

100 REM *** Find the log of a number to any base ***
110 REM
120 DEF FNLOGX(BASE,X)=LOG(X)/ LOG(BASE)
130 REM
140 PRINT 'Number'; \INPUT N \ PRINT 'Base'; \INPUT B
160 PRINT 'The log of';N; 'to the base';B; 'is';FNLOGX(B,N)
170 END
RUN
Number? 81
Base? 3
The log of 81 to the base 3 is 4
Ready

```

LOG10

Syntax

LOG10 (expression)

where:

expression is a numeric expression greater than 0.

Example of Syntax

A=LOG10(B)

Purpose

LOG10 returns the common logarithm of a specified number.

Comments

A logarithm is the exponent of another number (called a base). Common logarithms use base 10. The common logarithm of a number n therefore is the power to which 10 must be raised to equal n . For example, the common (base 10) logarithm of 100 is 2, because 10 raised to the power 2 equals 100.

Example

```

10 REM PROGRAM TO DEMONSTRATE LOG10 FUNCTION
20 FOR I = 10 TO 100 STEP 10
30   PRINT i;LOG10(i)
40 NEXT I
50 END
RUN
10  1
20  1.30103
30  1.47712
40  1.60206
50  1.69897
60  1.77815
70  1.8451
80  1.90309
90  1.95424
100 2
Ready

```

MID\$

Syntax

MID\$ (string, expression1, expression2)

where:

string	is the string expression from which a substring is extracted.
expression1	is a numeric expression indicating the starting position of the substring.
expression2	is a numeric expression indicating the number of characters in the substring.

Examples of Syntax

PRINT MID\$ (NAME\$, 1%, 4%)

VOWEL\$ = MID\$ ('AEIOUY', N%, 1%)

Purpose

MID\$ extracts a substring (section) from a string.

Comments

- ☐ Starting with the character at expression1, a copy of a substring with length of expression2 is returned.
- ☐ If the starting position (expression1) is less than 1, PRO/BASIC assumes a value of 1.
- ☐ If the number of characters in the substring (expression2) is greater than the number of characters from the starting position to the end of the string, PRO/BASIC returns the remainder of the string.
- ☐ If the number of characters in the substring (expression2) is less than or equal to 0, or if the starting position is greater than the length of the string, PRO/BASIC returns an empty string.

Example

```

100 REM *** Program to demonstrate MID$ ***
110 REM
120 REM Define a function to return a substring containing
130 REM the rightmost N% characters of the string S$
140 REM
150 DEF FNRIGHT$(S$,N%)=MID$(S$,LEN(S$)-N%+1%,256%)
160 REM
170 PRINT 'Line'; \INPUT TTLINES$
180 FOR I% = 1% TO LEN (TTLINES$)
190     FOR J% = LEN(TTLINES$)-1% TO I% STEP -1%
200         PRINT ' ';
210     NEXT J%
220     PRINT FNRIGHT$(TTLINES$,I%)
230 NEXT I% \END
RUN

```

Line? Can you write FNLEFT?

```

      $?
      T$?
      FT$?
      EFT$?
      LEFT$?
      NLEFT$?
      FNLEFT$?
      FNLEFT$?
      e FNLEFT$?
      te FNLEFT$?
      ite FNLEFT$?
      rite FNLEFT$?
      write FNLEFT$?
      write FNLEFT$?
      u write FNLEFT$?
      ou write FNLEFT$?
      you write FNLEFT$?
      you write FNLEFT$?
      n you write FNLEFT$?
      an you write FNLEFT$?
      Can you write FNLEFT$?

```

Ready

NUM\$

Syntax

NUM\$ (expression)

where:

expression is a numeric expression.

Example of Syntax

N\$=NUM\$(N)

Purpose

NUM\$ returns numeric characters in string data type, formatted as they would be by the PRINT statement.

Comments

- PRO/BASIC does not allow direct conversion of numeric data to string data. For example, the following line is invalid:

```
10 A$=B%
```

To convert numeric data to string data, you must use the NUM\$ function, as shown in the following example:

```
10 A$=NUM$(B%)
```

- The NUM\$ function used with the EDIT\$ function allows you to modify the standard print format for numbers. For example, suppose that you want to create a file with line numbers beginning in the first character position. You could write:

```
100 OPEN 'TEST.DAT' FOR OUTPUT AS FILE #1
110 FOR I = 10 TO 1000 STEP 10
120   PRINT #1, EDIT$(NUM$(I),2)
130 NEXT I
140 END
```

- NUM\$ allows you to perform string operations on numbers. For example, by converting a number to a string you can search for and extract a substring with the POS or MID functions.
- If the specified number is negative, NUM\$ places a minus sign in the returned string.

- NUM\$ converts the specified number to E notation if necessary. For example:

```
PRINT NUM$(10000000000)
      .1E+10
Ready
```

- The NUM\$ function is the inverse of the VAL function, which returns the numeric equivalent of a specified string.

Example

```
100 REM *** Program to Demonstrate NUM$ ***
110 REM
120 REM Find the number of nines in a number.
130 REM
140 PRINT 'Number'; \ INPUT N
150 N$=NUM$(N) \ NINES%=0%
160 FOR I% = 1% TO LEN(N$)
170   IF MID$(N$,I%,1%)='9' THEN NINES%=NINES%+1%
180 NEXT I%
190 PRINT 'The number'; N; 'contains'; NINES%; 'nines.'
RUN
Number? 89419.3
The number 89419.3 contains 2 nines.
Ready
```

PI**Syntax**

PI

Example of Syntax

```
PRINT 'PI ='; PI
```

Purpose

PI returns the value 3.141592653589793.

Comments

- ☐ Use PI in any numeric expression that requires the constant PI.
- ☐ Use PRINT USING to display the double-precision value of PI.
- ☐ You cannot change the value returned by PI.

Example

```

100 REM *** Program to convert radians to ***
110 REM *** degrees and degrees to radians ***
120 REM
130 DEF FNDEG(RAD)=RAD*(180/PI)
140 DEF FNRAD(DEG)=DEG*(PI/180)
150 REM
160 ON ERROR GOTO 300
170 PRINT '0 Exit program'
180 PRINT '1 Radians to Degrees'
190 PRINT '2 Degrees to Radians'
200 PRINT \ PRINT 'Command'; \ INPUT N% \ PRINT
210 ON N%+1% GOTO 32767,220,240
220 PRINT 'Radians'; \ INPUT V \ PRINT
230 PRINT V; 'Radians =';FNDEG(V);'Degrees' \ GOTO 200
240 PRINT 'Degrees'; \ INPUT V \ PRINT
250 PRINT V;'Degrees =';FNRAD(V);'Radians' \ GOTO 200
300 REM
310 REM *** Error-handling routine ***
320 REM
330 IF ERL=200 THEN PRINT \ RESUME 170
340 IF ERL=210 THEN RESUME 170
350 IF ERL=220 OR ERL=240 THEN RESUME
360 ON ERROR GOTO 0 \ RESUME 32767
32767 END
RUN

```

0 Exit program

1 Radians to Degrees

2 Degrees to Radians

Command? 1

Radians? 1

1 Radians = 57.2958 Degrees

Command? 2

Degrees? 360

360 Degrees = 6.28319 Radians

Command? 0

Ready

POS

Syntax

POS (string1, string2, expression)

where:

string1	is a string to be searched (the searched-string).
string2	is a substring to search for (the search-for string).
expression	is a numeric expression indicating the position at which to begin the search (the begin-search position).

Examples of Syntax

```
P%=POS(ALPHA$,'NOP',1)
ON POS('AEIOU',LETTER$,1) GOTO 10,20,30,40,50
```

Purpose

POS searches a string for a substring and returns the position of the substring's first character.

Comments

- ☐ If the search fails, POS returns 0.
- ☐ If the begin-search position is less than 1, POS assumes a value of 1.
- ☐ POS returns the substring's position by counting from character position 1, regardless of the begin-search position. For example:

```
PRINT POS ('ABCDEFGH', 'FG', 3%)
6
```

Ready

- ☐ If the begin-search position is greater than the length of the searched string, POS returns a value of 0.

- If the search-for string is null:

and the begin-search value is greater than 1 but less than or equal to the length of the searched-string, POS returns the begin-search position. For example:

```
PRINT POS ('ABCDEFG', "", 3%)
3
```

Ready

and the searched-string is also null, POS returns a value of 1. For example:

```
PRINT POS ("", "", 2%)
1
```

Ready

If the begin-search position is greater than the length of the searched-string, POS returns the length of the searched-string plus 1. For example:

```
PRINT POS ('ABCDEFG', "", 10%)
8
```

Ready

Example

```
10 PRINT 'What is the current month';
20 INPUT MONTH$ \ M$ = MID$ (MONTH$, 1%, 3%)
30 N% = POS('JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC', M$, 1%)
40 IF N% = 0% THEN PRINT 'No such month' \ GOTO 60
50 PRINT MONTH$; ' is month number'; 1% + (N% - 1%) / 3%
60 END
RUN
```

What is the current month? JUNE
JUNE is month number 6

Ready

RND

Syntax

RND

Example of Syntax

```
I%=INT(RND*10)+1%
```

Purpose

RND returns a random number greater than or equal to 0 and less than 1.

Comments

- Each time PRO/BASIC executes an OLD, NEW, or RUN command or a CHAIN statement, RND is reset to produce the same random number or random number sequence.

However, when the RANDOMIZE statement is executed before the RND function, a different random number or series of random numbers is produced each time the RND function is used.

- You can generate random integers within a range using the following formula:

$$R\% = \text{INT}((\text{MAX_VAL}\%)*\text{RND})+1\%$$

where: MAX_VAL% is the largest integer you want to generate.

For example, to generate random integers between 1 and 10:

```
10 FOR I%=1% TO 10%
20   PRINT INT(10%*RND)+1%;
30 NEXT I%
RUN
```

9 3 5 2 7 4 10 3 8 5

Ready

Example

```

100 REM *** Program to Demonstrate the RND Function ***
110 REM
120 REM A simulation of the game of craps
130 REM
140 RANDOMIZE \ WON%, LOST% = 0%
150 PRINT 'Games'; \ INPUT GAMES%
160 FOR I% = 1% TO GAMES%
170   GOSUB 360|REM Roll the dice
180   IF ROLL% = 7% OR ROLL% = 11% THEN 270
190   IF ROLL% = 2% THEN PRINT ' Snakeyes!'; \ GOTO 280
120   IF ROLL% = 3% THEN 280
210   IF ROLL% = 12% THEN PRINT ' Boxcars!'; \ GOTO 280
220   POINT% = ROLL% \ PRINT ' Your point is'; POINT%
230   GOSUB 360 REM Roll the dice
240   IF ROLL% = 7% THEN 280
250   IF ROLL% = POINT% THEN 270
260   PRINT \ GOTO 230
270   PRINT ' You win!' \ WON% = WON% + 1% \ GOTO 290
280   PRINT ' You lose!' \ LOST% = LOST% + 1%
290   PRINT
300 NEXT I% \ PRINT
310 PRINT 'Your score was'; WON%; 'wins and'; LOST%; 'losses.';
320 GOTO 32767
330 REM
340 REM *** Subroutine to roll a pair of dice ***
350 REM
360 DIE1% = INT (6% * RND) + 1% \ DIE2% = INT (6% * RND) + 1%
370 ROLL% = DIE1% + DIE2%
380 PRINT USING 'Your rolled # and # for ##. ', DIE1%, DIE2%, ROLL%;
390 RETURN
32767 END
RUN

```

Games? 3

You rolled 6 and 5 for 11. You win!

You rolled 5 and 5 for 10. Your point is 10

You rolled 4 and 6 for 10. You win!

You rolled 5 and 2 for 7. You lose!

Your score was 2 wins and 1 losses.

Ready

SIN

Syntax

SIN (expression)

where:

expression is a numeric expression which represents an angle expressed in radians.

Example of Syntax

A=SIN(B)

Purpose

The SIN function returns the sine of an angle.

Comments

- ☐ The argument is expressed in radians.
- ☐ The value returned is between -1 and 1.
- ☐ COS and SIN may produce inaccurate results when handling large numbers.

Example

The following example program uses SIN to plot a sine curve on the display screen. Type it in and run it to see the results. Press INTERRUPT then DO to stop the program.

```
10 PRINT TAB (INT (30% * (SIN (I) + 1%) + 0.5)); ""
20 I = I + 0.2 \ GOTO 10
```

Press INTERRUPT/DO to stop.

SQR

Syntax

SQR (expression)

where:

expression is a numeric expression.

Example of Syntax

$C = \text{SQR}(A^2 + B^2)$

Purpose

SQR returns the square root of a specified number.

Comments

The number specified must be greater than or equal to 0. If a negative number is specified, an error message is displayed.

Example

PRINT SQR(2)

1.41421

Ready

TAB

Syntax

TAB (expression)

where:

expression is a numeric expression indicating the printing position.

Example of Syntax

```
PRINT TAB(10%);'Name';TAB(30%);'Address'
```

Purpose

TAB is used with a PRINT statement to move the cursor or print position rightward to a specified column on the display screen or in a file.

Comments

- When TAB is used to move the cursor on the display screen, it moves the print position to the specified column. A line on the screen is 80 columns long.

When TAB is used to move the print position in a file, it moves the print position to the specified column on the line in the file. A line in a file is 132 columns long.

- Use semicolons with the TAB function. If commas are used, PRO/BASIC goes on the the next print zone before executing the TAB function.
- The first column at the left margin is column 0. The print position can be anywhere from 0 to the right margin of the screen. The print position can be moved only from the left to the right. If the cursor position is greater than the number specified in TAB, the print position is not changed.

Examples

```
10 PRINT "NAME";TAB(15);"ADDRESS";TAB(30);"PHONE NO."
20 END
RUN
NAME          ADDRESS      PHONE NO.
Ready
```

Without TABS 15 and 30, PRO/BASIC prints:

```
NAMEADDRESSPHONE NO.
```

TAB formats numbers as well:

```
10 PRINT "Column 0";TAB (15) "Column 16";TAB (30);"Column 31"
20 PRINT 100;TAB(15);29;TAB(30);35
30 END
RUN

Column 0      Column 16      Column 31
100           29           35
Ready
```

TIME\$

Syntax

TIME\$ (0%)

Example of Syntax

NOW\$=TIME\$(0%)

Purpose

TIME\$ returns the current time.

Comments

- **TIME\$** returns the time in the format:

hh:mm

where:

hh is the hour in 24 hour format.

mm is the minutes in 00 to 59 format.

- The only valid argument for the **TIME\$** function is 0. An argument is included only for compatibility with other versions of BASIC.

Example

PRINT TIME\$(0%)

11:21

Ready

VAL

Syntax

VAL (expression)

where:

expression is a string expression representing a number in either standard notation or E notation.

Examples of Syntax

A=VAL(B\$)

TEN%=VAL('10')

Purpose

VAL returns a real number equivalent to the number represented by a string expression.

Comments

- ❑ PRO/BASIC does not allow direct conversion of string data to numeric data. For example, the following program is invalid:

```
10 A%=B$
```

- ❑ To convert string data to numeric data, you must use the VAL function, as shown in the following example:

```
10 A%=VAL(B$)
```

- ❑ The string cannot contain embedded spaces or tabs. For example, the following program will cause an error:

```
PRINT VAL(' 1 2 3 ')
```

- ❑ If the string is null, VAL returns 0.
- ❑ The VAL function is the opposite of the NUM\$ function, which returns the string equivalent of a numeric argument.

Example

```
10 NOW$=TIME$(0%)
20 HOURS%=VAL(MID$(NOW$,1,2))
30 MINUTES%=VAL(MID$(NOW$,4,2))
35 T$="The time at the tone will be: ##:##"
36 LET B$=CHR$(7)
40 IF HOURS%<12 GOTO 50 ELSE GOTO 70
50 PRINT USING T$+"AM",HOURS%,MINUTES% \ PRINT B$ \ GOTO 75
70 HOURS%=HOURS%-12 \ PRINT USING T$+"PM",HOURS%,MINUTES% \ PRINT B$
75 END
RUN
The time at the tone will be: 2:30 PM
Ready
```

7

PRO/BASIC Graphics

Chapter 7

PRO/BASIC Graphics

INTRODUCTION

PRO/BASIC graphics statements allow you to draw lines, curves, and polygons, and print characters in different sizes, styles, and at different angles, using a palette of 256 colors.

To describe these statements fully, we must introduce coordinates, and the viewport and window coordinate systems used in PRO/BASIC graphics.

But first, let us outline the possible hardware configurations and the capabilities of each.

7.1 GRAPHICS HARDWARE

PRO/BASIC graphics capabilities are determined by two components, the monitor and the extended bit-map option (EBO). The two types of monitors available are color and monochrome. The extended bit-map option can be purchased for your Professional 350.

If you do not have the extended bit-map option you can display graphics in black and white.

If you have the extended bit-map option and a monochrome monitor, you can use eight shades of gray, ranging from black to white.

If you have the extended bit-map option and a color monitor, you can choose from 256 colors and display any eight of them on the screen at one time.

		MONITOR	
		COLOR	MONOCHROME
Yes		256 Colors Any 8 at One Time	8 Shades of Grey
EBO			
No		Black and White	

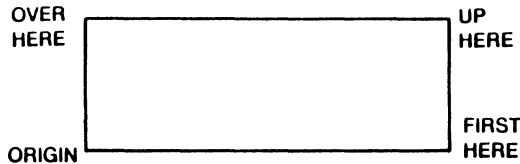
Figure 7-1

7.2 COORDINATES

Coordinates are used to specify a point in space, on a wall or on a computer's display screen. To hang a painting on a wall we say, hang the painting 10 feet out from the corner, and 8 feet up from the floor. The horizontal distance paired with the vertical distance specifies a single location.

To locate a point we need to identify its horizontal location and its vertical location. These measurements are made from the same starting point, as the bottom corner is the starting point for measurements to hang a painting.

The following example suggests a rectangle by describing a line drawn from a start point, or origin, first horizontally to 'first here,' then vertically to 'up here,' then horizontally to 'over here,' and finally back to the origin.



Numeric values provide a more realistic method of describing the rectangle. Coordinates are pairs of numbers which determine the location of a point. By specifying a pair of coordinates, one horizontal, one vertical, and giving the value 0 to the origin, we can identify a location relative to the origin.

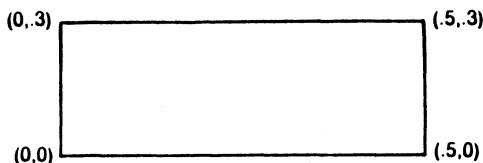
The method used is:

horizontal displacement from origin, vertical displacement from origin

Horizontal and vertical displacement relative to the origin are represented by x and y. A value in x represents horizontal location; a value in y represents vertical location. The shorthand notation is:

(x,y)

We can now use numeric values to define a rectangle. Using (x,y) to identify a location, let the origin be at (0,0). From the origin move horizontally to (.5,0), then vertically to (.5,3), then horizontally to (0,3), and finally back to the origin.

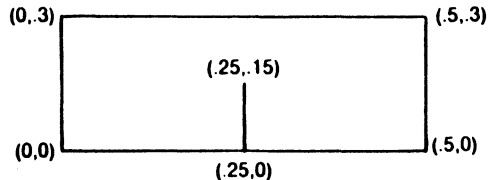


To locate the center of the rectangle, find the midpoints of the horizontal line and vertical line by dividing the lengths of the horizontal line ($x=.5$) and vertical line ($y=.3$) by 2.

$$.5/2 = .25$$

$$.3/2 = .15$$

From the origin (0,0), move to the horizontal midpoint at (.25,0), then move to the vertical midpoint at (.25,.15). This identifies the middle point of the rectangle.



Having introduced the notion of coordinates, we can proceed to a discussion of two coordinate systems that are important in PRO/BASIC graphics: viewport coordinates, which allow us to choose a part of the screen to display graphics, and window coordinates, which allow us to choose a convenient scale for that portion of the screen.

7.3 VIEWPORT COORDINATES

The viewport is the portion of the screen available for graphics. The viewport can be as large as the physical screen, but no larger. The size of the viewport can be set by specifying four points.

Viewport coordinates are used to specify the size of the viewport; however, because four values are needed to do this, a range of x values and a range of y values are provided instead of a pair of coordinates. Ranges for x and y are identified in the following format:

minimum x, maximum x, minimum y, maximum y

This format says, create a region on the screen for the display of graphics from min x to max x, and from min y to max y. The default values for the viewport setting are as follows:

.375,1,0,.625

These values set the viewport equal to a square region at the right of the screen with the origin at its lower left, as shown in Figure 7.2.

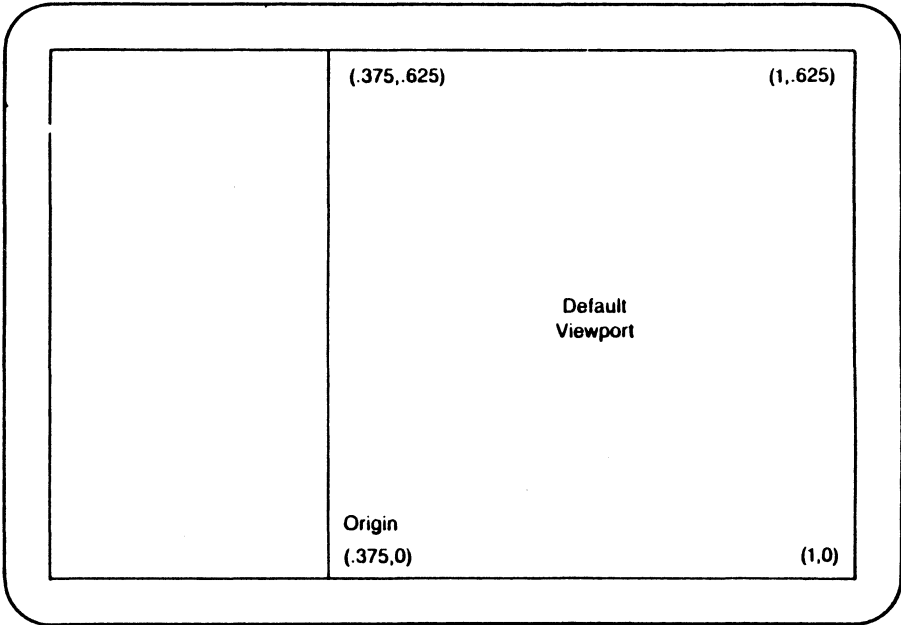


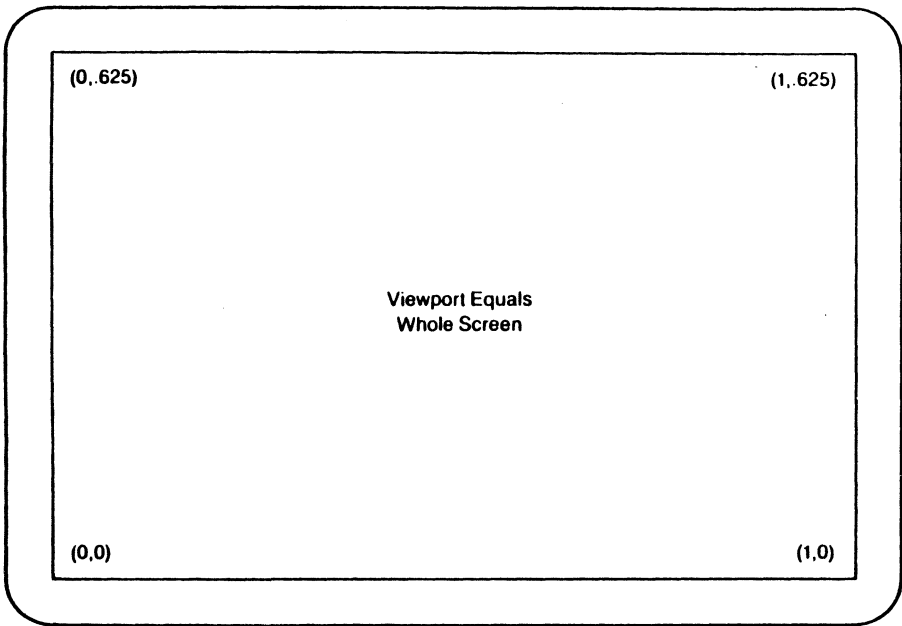
Figure 7-2

The viewport setting shown is the default setting; the viewport is reset to these values each time you run a program.

The following settings are used to set the viewport equal to the entire screen:

0,1,0,.625

This viewport setting causes the lower left corner of the viewport, that is, the origin, to be in the lower left corner of the screen, as shown in Figure 7.3:

**Figure 7-3**

Because of the inequality of the display screen's vertical and horizontal sizes, a proportional adjustment is made to reconcile the y dimension and the x dimension.

The horizontal dimension is defined as extending from 0 to 1. The y dimension is 62.5% as long as the x dimension, and extends from 0 to .625.

To review viewport coordinates, the viewport coordinate system is used to identify the region on the display screen used for graphics, called the viewport. Four points, within the range of 0 to 1 horizontally, and 0 to .625 vertically, define the viewport.

The default viewport setting provides for equal length in the two dimensions (a square of .625 by .625 in viewport coordinates).

7.4 WINDOW COORDINATES

The viewport coordinate system defines the portion of the screen to use for graphics: the window coordinate system specifies a scale to use within that region of the screen.

The window coordinate system specifies a range of values for the horizontal (x) dimension and a range of values for the vertical (y) dimension. Points are plotted with window coordinates in the range specified for each dimension.

You determine the scale that points are plotted on. Ranges for x and y are identified in the same format as for the viewport:

min x, max x, min y, max y

The values provided create vertical and horizontal scales within which points are plotted. The default window coordinates are as follows:

0,1,0,1

The measurements range from 0 to 1 horizontally and 0 to 1 vertically, as shown in Figure 7-4.

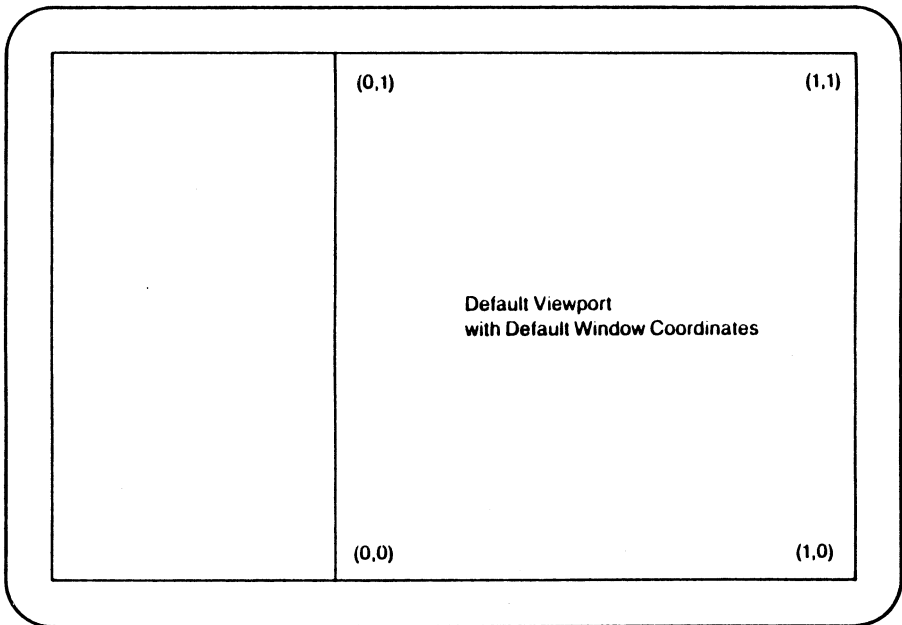


Figure 7-4

The origin is at the lower left corner and has value 0,0. The window coordinates are reset to these values each time a program is run.

To review the distinction between viewport coordinates and window coordinates; the viewport specifies which part of the screen to use for graphics, the window specifies which scale to use for plotting points. Points are plotted in window coordinates, not viewport coordinates.

7.5 TWO GRAPHICS STATEMENTS

The SET VIEWPORT statement receives values for the viewport setting. The SET WINDOW statement determines the scale to be applied to the viewport dimensions. Consider these two statements which set up a graph:

```
10 SET VIEWPORT 0,1,0,.625
20 SET WINDOW 1900,1985,0,15
```

The values in SET VIEWPORT set the viewport to equal the entire screen. The values in SET WINDOW apply a scale ranging from 1900 to 1985 horizontally and 0 to 15 vertically to plot average monthly rainfall, as shown in Figure 7-5.

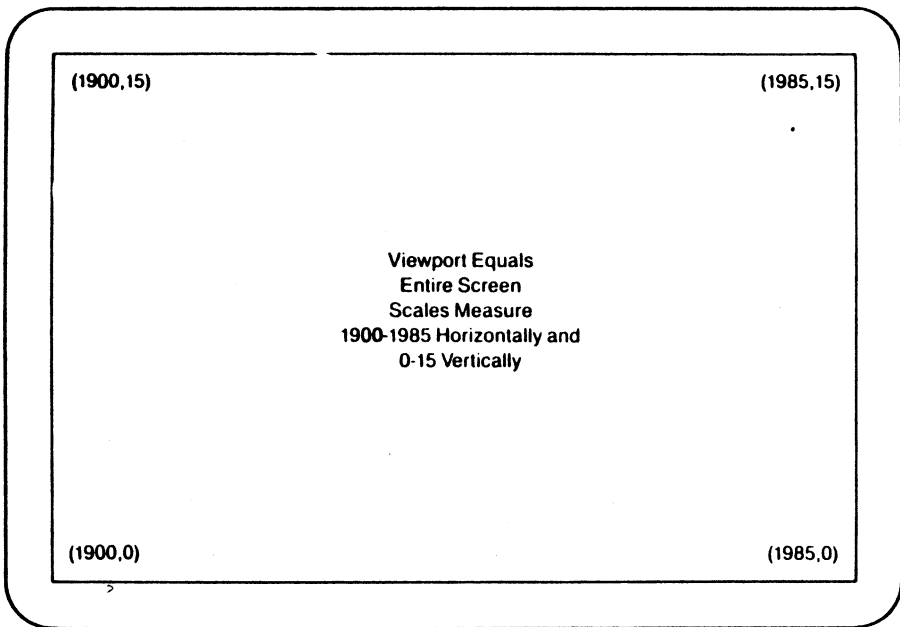


Figure 7-5

Any points that we wish to plot on the screen, then, must have x values between 1900 and 1985, and y values between 0 and 15.

ASK POSITION

Syntax

ASK POSITION (x,y)

where:

x,y are real variables which receive the
coordinates of the current location.

Example of Syntax

ASK POSITION (TEMP_X,TEMP_Y)

Purpose

Stores the horizontal coordinate of the current location in the x variable and the vertical coordinate of the current location in the y variable.

Comments

- The position is given in terms of the window coordinate system.
See SET WINDOW.
- ASK POSITION is useful if you do not wish to constantly keep track of the current position but need to know it at times. For example, a subroutine may need to alter the current location but must restore the previous position before it returns.

Example

```

.
.
340 SET POSITION (.5,.5)
350 GOSUB 1000
360 PLOT ARC (.6,.5,360)
.
.
.

1000 REM subroutine that changes position, but must restore it
1010 ASK POSITION (TEMPX,TEMPY)
1020 SET POSITION (0,0)
1030 GRAPHIC PRINT 'Hello.'
1040 SET POSITION (TEMPX,TEMPY)
1050 RETURN

```


CLEAR

Syntax

CLEAR

Purpose

Clears the screen.

Comments

- ☐ The CLEAR statement causes the entire screen to become the background color.
- ☐ It also moves the graphics cursor to the origin ((0,0) in the default window coordinates) and moves the text cursor to the top left corner of the screen.
- ☐ CLEAR also negates the effect of a separator at the end of the last PLOT statement.

GRAPHIC PRINT

Syntax

GRAPHIC PRINT print list

GRAPHIC PRINT USING format string, print list

where:

print list is one or more valid expressions to be printed, separated by commas or semicolons.

format string is a string expression which describes the format used for printing.

Examples of Syntax

```
100 GRAPHIC PRINT "Hello ";NAME$;"|"
```

```
200 GRAPHIC PRINT USING 'The answers are ##.##',N1;N2
```

Purpose

Prints graphics characters.

Comments

- The separator used in the print list (“;” or “,”) has the same effect as separators used in PRINT and PRINT USING statements. If a semicolon (“;”) is used to separate print list items, no extra spaces are inserted when the items are printed. If a comma (“,”) is used to separate print list items, spaces are inserted to cause the next print list item to start at a multiple of 14 spaces from the original position.
- The appearance of printed text depends upon other previous settings such as position, font, character spacing, character size, italic angle, text angle, color, and writing mode.
- If no trailing separator is used, the graphics cursor is restored to the position it had prior to execution of the GRAPHIC PRINT statement. To leave the graphics cursor at the end of the item printed, use a separator at the end of the GRAPHIC PRINT statement.

Example

```

10 REM Demonstrate the GRAPHIC PRINT statement
20 REM (Note the use of separators).
30 REM
40 SET POSITION (0,.25)
50 GRAPHIC PRINT ' a';'bc','def';
60 GRAPHIC PRINT '*' \ REM Here is the graphics cursor position
70 REM
80 SET POSITION (0,.2)
90 GRAPHIC PRINT ' a';'bc','def'
100 GRAPHIC PRINT 'E' \ REM Cursor restored to original position
110 END
run

```

Ready

abc	def*
@abc	def

PLOT

Syntax

PLOT [(x,y) [separator(x,y)] ... [separator]]

where:

x,y are numeric expressions indicating the drawing position.

separator is a comma or a semicolon.

Examples of Syntax

PLOT (.1,0)

PLOT (0,0),(1,0),(1,1),(1,0),(0,0)

PLOT (.5,.5),(.75,.5),

Purpose

Draws a point at the specified position, or draws lines between successive specified positions.

Comments

- The positions are specified in window coordinates.
- If the last PLOT statement had a separator at the end, a line will be drawn from the current position to the point given in the PLOT statement.

If no separator was used, a point is drawn at the first point given in the PLOT statement. Consider the example below:

10 PLOT (0,0),

20 PLOT (1,1)

Plots a line from (0,0) to (1,1), whereas

10 PLOT (0,0)

20 PLOT (1,1)

just plots the points (0,0) (1,1)

Remember that many graphic statements change the current position.

- A CLEAR statement negates the effect of a separator at the end of a PLOT statement.

- A PLOT statement without any coordinates is a way to end a line. The following example draws lines through 50 random points. The PLOT statement in line 340 has no separator, and ends the line.

```

310 FOR I = 1 TO 50
320 PLOT (RND,RND);
330 NEXT I
340 PLOT \ REM End the current line
350 PLOT (0,0),(1,0),(1,1),(0,1),(0,0) \ REM Plot a border

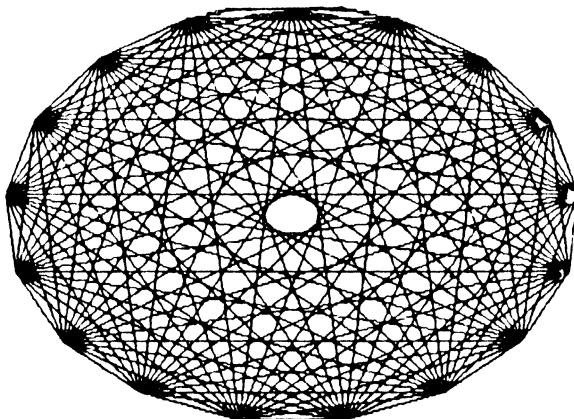
```

Example

```

100 N=17
110 FOR I=0 TO N
120   RAD=2*PI*I/N
130   X=(SIN(RAD)+1)/2
140   Y=(COS(RAD)+1)/2
150   FOR J=I+1 TO N
160     RAD2=2*PI*J/N
170     X2=(SIN(RAD2)+1)/2
180     Y2=(COS(RAD2)+1)/2
190     PLOT (X,Y),(X2,Y2)
200   NEXT J
210 NEXT I
220 PRINTSCREEN
run

```



PLOT ARC

Syntax

PLOT ARC (x,y, angle)

where:

x,y are real expressions indicating the center of the arc.

angle is a numeric expression which specifies the angle
(in degrees) of the arc.

Examples of Syntax

PLOT ARC (.1,.1,360)

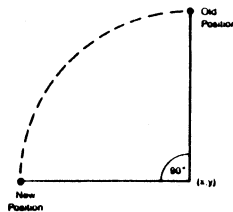
PLOT ARC (x,edge -.5,180)

Purpose

Plots an arc, i.e., a section of a circle, using the current position as a point on the circumference.

Comments

- ☐ The current location is specified in window coordinates.
- ☐ The arc will be drawn from the current position along an arc, the center of which is at the specified position. For example:



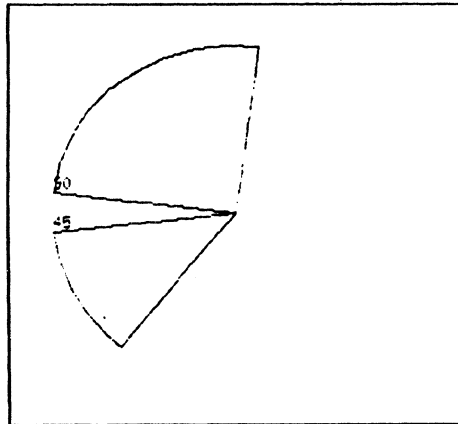
- ☐ If the angle is positive, the arc will be drawn counter-clockwise, and if the angle is negative, the arc will be drawn clockwise.
- ☐ PRO/BASIC draws the arc and changes the current position to the last position on the arc.
- ☐ You can draw a circle by specifying an arc of 360 degrees.

Example

```

10 DELTA=.05
20 X_CENTER,Y_CENTER=.5
30 X_CIRCUM=.1\Y_CIRCUM=.5
40 REM Plot a 45 degree slice
50 PLOT (X_CENTER,Y_CENTER),(X_CIRCUM,Y_CIRCUM-DELTA),
60 GRAPHIC PRINT '45'
70 PLOT ARC (X_CENTER,Y_CENTER,45)\ REM Positive angle -- CCW
80 PLOT (X_CENTER,Y_CENTER),
90 REM Plot a 90 degree slice
100 PLOT (X_CIRCUM,Y_CIRCUM+DELTA),
110 GRAPHIC PRINT '90'
120 PLOT ARC (X_CENTER,Y_CENTER,-90)\ REM Negative angle -- CW
130 PLOT (X_CENTER,Y_CENTER)
140 REM Plot a border
150 PLOT (0,0),(1,0),(1,1),(0,1),(0,0)
160 END
run

```

**Ready**

PLOT CURVE

Syntax

PLOT CURVE (x array, y array, expression, type)

where:

x array	is the one-dimensional single-precision array of x values of the curve to be plotted.
y array	is the one-dimensional single-precision array of y values of the curve to be plotted.
expression	is an integer expression that specifies the number of points, beginning with array element 0, that are to be plotted on the curve.
type	if a non-zero value, requires that the ends of the curve meet, or if equal to zero, does not require that the ends of the curve meet.

Example of Syntax

PLOT CURVE (X(),Y(),11,-1%)

Purpose

Draws a curve through the specified points.

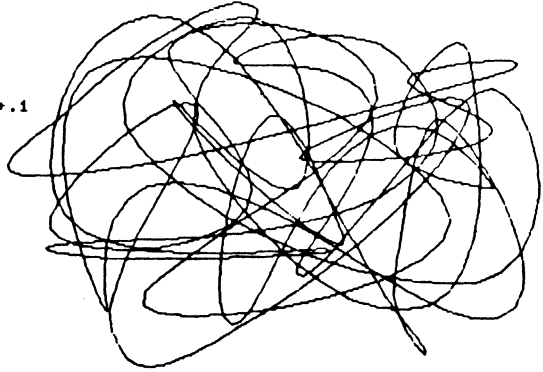
Comments

- ☐ The points are specified in window coordinates.
- ☐ The two arrays must have only one dimension.
- ☐ Points on the curve are plotted starting with array element 0. Note that the last array element of each array plotted is 1 less than the number of points specified (since plotting starts at element 0).
- ☐ If you specify four equidistant points the **PLOT CURVE** statement can draw circles (and ellipses). For example:

```
10 X(0)=.2 \ X(1)=.5 \ X(2)=.8 \ X(3)=.5
20 Y(0)=.5 \ Y(1)=.3 \ Y(2)=.5 \ Y(3)=.7
30 PLOT CURVE (X(),Y(),4%,-1%)
```


Example

```
10 N=50
20 DIM X(N),Y(N)
30 FOR I=0 TO N-1
40   X(I)=RND*.8+.1\Y(I)=RND*.8+.1
50 NEXT I
60 X1=X(N-1)\Y1=Y(N-1)
70 SET POSITION (X1,Y1)
80 PLOT CURVE (X(),Y(),N-1,1)
90 PRINTSCREEN
100 END
run
```



PRINTSCREEN

Syntax

PRINTSCREEN

Purpose

Prints the current screen image on the printer.

Comments

- You must have an appropriate printer connected to your Professional 350 in order to use this statement. The LA100 and LA50 printers are both suitable.
- While the screen image is being printed, no other activity can take place. Your program will appear to freeze, but will continue after the printer is done.
- The printer displays a reversed image, in the sense that it prints black on a white background, whereas your screen usually displays white (and other colors) on a black background. Thus, with the default color settings, the printer interprets the background color (color 0) as white, and all other colors (1 through 7) as black. The printer ignores the red, green and blue settings of the colors.

However, if the background color (color 0) is specified as anything but black, its image will be reversed as well, and the entire screen will be printed as black.

SCROLL

Syntax

SCROLL (dx,dy)

where:

dx,dy are numeric expressions indicating the amount of horizontal shift (dx) and the amount of vertical shift (dy) from the current location.

Example of Syntax

SCROLL (.1,.05)

Purpose

SCROLL moves the graphic image in the specified direction.

Comments

- ☐ The horizontal and vertical shifts are given in terms of the current window coordinates. See SET WINDOW.
- ☐ The graphics which move off the edge of the screen are lost, and solid background comes in at the opposite edges.

Example

```

10 CLEAR
 0 SET POSITION (.5,.35)
30 PLOT ARC (.5,.5,360) \ REM      DRAW THE CIRCLE
40 DX=.002 \DY=0 \ REM          INITIAL X,Y DIRECTION
50 CALL INKEY (A$) \ REM        HAS USER HIT A KEY?
60 REM
70 REM if we get a character with INKEY, see if it means anything
80 REM
90 IF A$='q' OR A$='Q' THEN GOTO 190 \ REM      Q--> 'QUIT'
100 IF A$='a' OR A$='A' THEN GOTO 10 \ REM      A--> 'AGAIN'
110 IF A$='2' THEN DX=0 \DY=.0045 \ REM        2--> move down
120 IF A$='4' THEN DX=.0045 \DY=0 \ REM        4--> left
130 IF A$='5' THEN DX=0 \DY=0 \ REM            5--> stop
140 IF A$='6' THEN DX=-.0045 \DY=0 \ REM       6--> right
150 IF A$='8' THEN DX=0 \DY=-.0045 \ REM       8--> up
160 REM
170 SCROLL (DX*2,DY*2) \ REM MOVE THE CIRCLE BY SCROLLING THE SCREEN
180 GOTO 50 \ REM GET ANOTHER KEY
190 END

```

SET CHARACTER

Syntax

SET CHARACTER character, array

where:

character is a single character to be defined in a font.

array is a 16 element integer array which contains the character's definition.

Example of Syntax

```
SET CHARACTER 'A',NEW_A%( )
```

Purpose

Allows the user to design printing characters.

Comments

- To create a character you must design the character and then translate that design to numeric information understandable to the computer. The numeric information is then stored in an array which has the name of the character defined. When the character is defined, the information about its composition needed to display it is copied from the array.

A grid is used to help design and define a character. The grid is numbered along its horizontal and vertical dimensions to represent the rows and columns of the character cell. A character cell is made up of many pixels, which are identified according to the columns in each row. (A pixel is the smallest controllable unit on the screen.

A pixel can be lighted or can be turned off. Characters are displayed with lighted pixels against a background of unlighted pixels. If you look closely at the screen, you can see that characters are composed of these small spots.)

The numbering along the vertical dimension counts the rows that make up the character. It goes from top to bottom, starting from 0. The numbering along the horizontal dimension goes by column from left to right. It starts at -32768 and works down from 16384 by dividing by two each time. These are the bit values of each pixel.

Below are two grids. They are of different sizes and have different numbering because they are each used to define characters for different fonts. Characters of font 1 are of 16 pixels by 16 pixels. Characters of font 2 are of 8 pixels by 8 pixels. Use the correct grid for the font you are creating characters for.

FONT 1

	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
0																
1																
2																
3																
4																
5																
6																
7																
8																
9																
10																
11																
12																
13																
14																
15																

FONT 2

	32768	16384	8192	4096	2048	1024	512	256
0								
1								
2								
3								
4								
5								
6								
7								

To create a printing character:

1. Design your character in the grid. When the design is done, add the values in each row of each pixel value that has been drawn in to compose the character.
2. Use the SET FONT statement to specify which font the character is to be included in. See the SET FONT statement in this chapter.
3. Enter the sum of the values from each row of the grid to each element of the array.
4. Use the SET CHARACTER statement to associate the character with the array.

Example

The example program below includes a small routine (lines 300 - 400) that you can use to automate the process of defining the values that compose a character, and entering those values to an array.

```

100 REM  0123456789012345
110 DATA M
120 DATA '   X           '
130 DATA '   X    XX      '
140 DATA '   X    XXX      '
150 DATA '   X     XX      '
160 DATA '   X    XXXXXX    '
170 DATA ' XX  XXXXXX  XX   '
180 DATA '   XXX  XXXX  XXX   '
190 DATA '     X   XXX  XXX   '
200 DATA '           XXXX  XXX '
210 DATA '           X  XX  X   '
220 DATA '           XX  XX     '
230 DATA '           XX   XX     '
240 DATA '           X     X     '
250 DATA '           X       X     '
260 DATA '           XX      XX     '
270 DATA '
280 DIM A%(15)
290 CLEAR
300 READ CHAR$
310 FOR I=0 TO 15
320   N=0
330   READ A$
340   FOR J%=15% TO 1% STEP -1%
350     IF MID$(A$,J%+1%,1%)<>' ' THEN N=N+ (15%-J%)
360   NEXT J%
370   IF MID$(A$,1%,1%)<>' ' THEN N=N-32768
380   A%(I)=N
390 NEXT I
400 SET FONT 1 \ SET CHARACTER CHAR$,A%( )
410 SET CHARACTER SIZE .2,.2 \ GRAPHIC PRINT CHAR$

```

SET CHARACTER SIZE

yntax

SET CHARACTER SIZE width, height

where:

width is a numeric expression which specifies the width of a character cell in the current window coordinates.

height is a numeric expression which specifies the height of a character cell in the current window coordinates.

Example of Syntax

SET CHARACTER SIZE .04,.083334

Purpose

Sets the size of the graphics characters displayed in subsequent GRAPHIC PRINT statements.

Comments

- The width and height you specify determines the size (in window coordinates) of the character, though the displayed size is actually the closest multiple of the number of pixels used in the character's definition. The displayed size of the character therefore will be no larger than the size specified, and may be smaller. (A pixel is the smallest controllable unit of the screen. The screen has 240 pixels vertically and 960 horizontally.)

Font 0 characters are 12 pixels wide and 10 pixels high.

Font 1 characters are 16 pixels wide and 16 pixels high.

Font 2 characters are 8 pixels wide and 8 pixels high.

Thus, characters in font 0 can be 12 pixels wide, 24 pixels wide, or 36 pixels wide, and they can be 10 pixels high, 20 pixels high, or 40 pixels high.

- The size also depends on the text angle. Characters written with text angles other than horizontal and vertical will appear 1.6 times larger than they do normally.
- Using font 0, and assuming the default viewport and window, the default character width is .02 (1.6/80) and the default character height

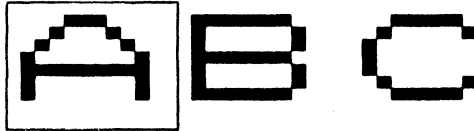
is .041667 (1/24). These dimensions result in the full 24 row by 80 column size of the screen. These values are reset whenever you run a program.

- Setting the character size also sets character spacing. You must reset the character spacing if you do not want characters to display with default spacing.

Example

```
10 SET CHARACTER SIZE .3,.3
20 PLOT (0,0),(0,.3),(.3,.3),(.3,0),(0,0)
30 GRAPHIC PRINT 'ABC'
40 PRINTSCREEN
```

run



Ready

SET CHARACTER SPACING

Syntax

SET CHARACTER SPACING x, y

where:

- x** is a numeric expression representing the horizontal distance between characters.
- y** is a numeric expression representing the vertical distance between characters.

Example of Syntax

SET CHARACTER SPACING .08,.15

Purpose

Sets the spacing between characters.

Comments

- The space between characters is measured (in window coordinates) from the beginning of one character to the beginning of the next. In other words, if characters should appear right next to each other, the horizontal character spacing should be the same as the character width. See SET CHARACTER SIZE.
- The default value for horizontal spacing is the same as the horizontal character size, which is .02 (1.6/80). The default value for vertical spacing is 0, that is, characters are displayed on the same horizontal line.

Example

```
10 SET CHARACTER SIZE .05,.15
20 SET POSITION (0,.3)
30 GRAPHIC PRINT 'digital'
40 SET POSITION (0,0)
50 SET CHARACTER SPACING .15,0
60 GRAPHIC PRINT 'digital'
70 PRINTSCREEN
```

run

digital

d i g i t a l

SET CLIP

Syntax

```
SET CLIP {ON }
         {OFF }
```

Example of Syntax

```
SET CLIP OFF
```

Purpose

Turns clipping on or off.

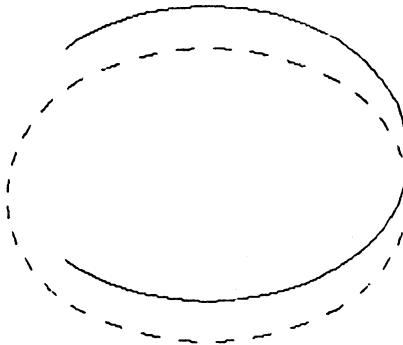
Comments

- ☐ The SET CLIP statement enables or disables the display of images that are not within the viewport boundaries most recently set.
- ☐ This statement affects only the graphics displayed after this statement is executed.
- ☐ Clipping is on by default.

Example

```
set position (.6,.55)
plot arc (.25,.55,360)
```

```
set clip "off"
set line style 2
set position (.6,.45)
plot arc (.25,.45,360)
```



SET COLOR

Syntax

SET COLOR color

where:

color is represented by an integer expression of value 0 to 7 which specifies one of the eight colors.

Example of Syntax

SET COLOR 4

Purpose

Selects a color for use by subsequent graphics statements.

Comments

- ☐ SET COLOR identifies the colors to be used when drawing patterns to the screen.
- ☐ The background is defined as color number 0. Its default is black.
- ☐ If the color number specified falls outside the range 0 - 7 the color is not changed.
- ☐ The color is reset to 7 whenever you run a program. Since color 7 is reset to white whenever you run a program, the default color is white.
- ☐ The default colors for color monitors are listed below:

Color Number	Default Color
0 (background)	BLACK
1	RED
2	GREEN
3	BLUE
4	YELLOW
5	MAGENTA
6	CYAN
7 (default writing color)	WHITE

- Refer to the SET COLORMAP statement in this chapter for more information on color settings.

Example

```

10 CLEAR
15 REM Print the color labels ...
20 PLOT (0,0),(1,0),(1,1),(0,1),(0,0)
40 SET POSITION (.35,.87) \ GRAPHIC PRINT "A COLOR WHEEL"
50 SET POSITION (.4,.15) \ SET COLOR 1 \ GRAPHIC PRINT "red"
60 SET POSITION (.7,.25) \ SET COLOR 4 \ GRAPHIC PRINT "yellow"
70 SET POSITION (.8,.57) \ SET COLOR 2 \ GRAPHIC PRINT "green"
80 SET POSITION (.62,.77) \ SET COLOR 6 \ GRAPHIC PRINT "cyan"
90 SET POSITION (.2,.7) \ SET COLOR 3 \ GRAPHIC PRINT "blue"
100 SET POSITION (.06,.4) \ SET COLOR 5 \ GRAPHIC PRINT "magenta"
105 REM print the color wheel
110 SET FILL (.5,.5) \ SET POSITION (.3,.3)
120 FOR I=1 TO 6
130     READ COLOR
140     SET COLOR COLOR
150     PLOT ARC (.5,.5,360/6)
160 NEXT I
170 PRINT 'hit RETURN' \ LINPUT WAIT$ \ CLEAR \ REM wait til user types return
190 DATA 1,4,2,6,3,5
200 END

```

SET COLORMAP

Syntax

SET COLORMAP color-number, red value, green value, blue value

where:

color-number	is a numeric value between 0 and 7 indicating the color to be defined.
red value	is a numeric value between 0 and 1 indicating the intensity of red in the color defined.
green value	is a numeric value between 0 and 1 indicating the intensity of green in the color defined.
blue value	is a numeric value between 0 and 1 indicating the intensity of blue in the color defined.

Example of Syntax

SET COLORMAP 0,.9,.2,.9

Purpose

SET COLORMAP defines eight colors available to your program.

Comments

- If your system has a color monitor and the extended bit-map option you can draw in eight colors. With a monochrome monitor you can draw in eight shades of grey. If your system doesn't have the extended bit-map option, this statement has no effect.

The default colors for color monitors are listed below:

Color Number	Default Colormap (R G B)	Display Color
0 (background)	0 0 0	BLACK
1	1 0 0	RED
2	0 1 0	GREEN
3	0 0 1	BLUE
4	1 1 0	YELLOW
5	1 0 1	MAGENTA
6	0 1 1	CYAN
7 (default writing color)	1 1 1	WHITE

- ☐ The eight colors are defined by intensities of red, green, and blue, represented by numeric values between 0 and 1, with 0 the lowest intensity and 1 the highest intensity.
- ☐ The color 0 is the background color. The color 7 is the default writing color.

The default shades for monochrome monitors are listed below:

Color Number	Default Value	Shade of Grey
0	0 0 0	DARKER
1	1/7 1/7 1/7	•
2	2/7 2/7 2/7	•
3	3/7 3/7 3/7	•
4	4/7 4/7 4/7	•
5	5/7 5/7 5/7	•
6	6/7 6/7 6/7	•
7	1 1 1	LIGHTER

- Anything previously written on the screen with the defined colors will be immediately changed to the new colors. The example below draws a circle with color set to 2 (green), then changes the color from green to red.

```

10 SET COLOR 2
20 SET POSITION (.3,.3)
30 PLOT ARC (.5,.5,360) \ REM Plot circle
40 INPUT WAIT$
50 SET COLORMAP 2,1,0,0 \ REM Change green to red

```

Example

```

100 SET VIEWPORT 0,1,0,.625 \ CLEAR
110 SET FILLY 1
120 FOR C=0 TO 7
130     SET COLOR C
140     PLOT (C/8,0),((C+.5)/8,0)
150 NEXT C
160 SET FILLX 1
170 FOR C=0 TO 7
180     SET COLOR C
190     PLOT (0,C/8),(0,(C+.5)/8)
200 NEXT C
210 FOR GREEN=0 TO 7 \ G=GREEN  7
220     FOR BLUE=0 TO 3 \ B=BLUE  3
230         RED=0 TO 7 \ R=RED  7
240         SET COLOR MAP RED,G,R,B
250         NEXT RED
260     NEXT BLUE
270 NEXT GREEN
280 SET COLORMAP 7,1,1,1
290 SET COLORMAP 0,0,0,0

```

SET FILL

Syntax

SET FILL (x,y)

where:

x,y are numeric expressions which specify the coordinates of a point to fill to.

Example of Syntax

SET FILL (.9,.5)

Purpose

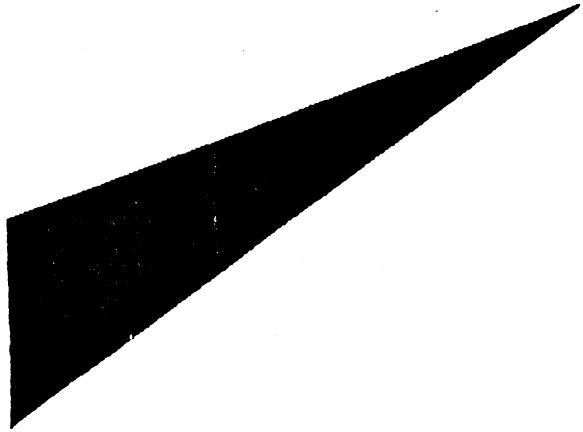
Specifies a point to fill to.

Comments

- ☐ The point to fill to is specified in window coordinates.
- ☐ The fill pattern is affected by the current fill style and by the current writing mode. See SET FILL STYLE, SET LINE STYLE, and SET WRITING MODE.
- ☐ All graphics executed after this statement will be filled to the specified point. Fill is done with the foreground color, the fill pattern is solid by default.
- ☐ SET FILL to a point is much slower than fill to a horizontal line or fill to a vertical line. Use SET FILLX or SET FILLY whenever possible.

Example

```
set fill (0,.5)  
plot (0,0),(1,1)
```



SET FILL OFF

Syntax

SET FILL OFF

Example of Syntax

SET FILL OFF

Purpose

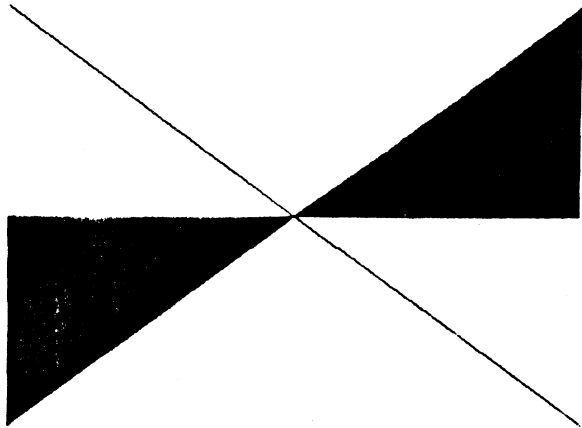
Turn fill off.

Comments

The fill style is reset to solid.

Example

```
set fill .5  
plot (0,0),(1,1)  
set fill off  
plot (0,1),(1,0)
```



Ready

SET FILL STYLE

Syntax

SET FILL STYLE character

where:

character is to be used as the fill pattern.

Example of Syntax

SET FILL STYLE '2'

Purpose

Change the fill pattern to be the specified character in the current font.

Comments

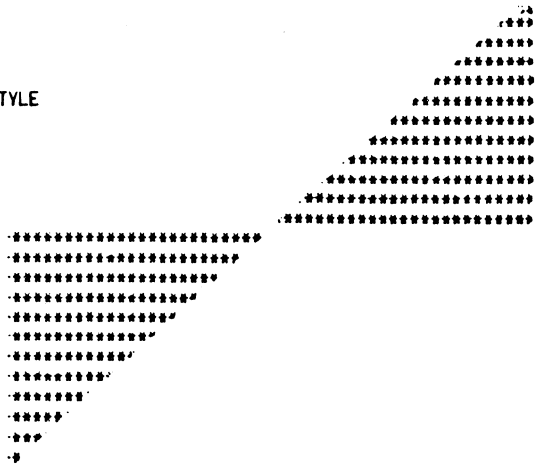
- ☐ If more than one character is provided in the string, the first character is used. If the string is empty (length of 0), the fill style is not changed.
- ☐ Specify control characters in SET FILL STYLE to get a small checkerboard pattern as the fill style. Control characters have ASCII values 1–31. Use the CHR\$ function with ASCII values to specify control characters, for example CHR\$(1).
- ☐ To get solid fill after using some other character as the fill pattern, use SET FILL STYLE CHR\$(0).
- ☐ The fill style is reset to solid whenever you run a program.
- ☐ The effect of SET FILL STYLE depends on SET WRITING MODE. (See SET WRITING MODE in this chapter.)

Example

```

10 REM Demonstrate SET FILL STYLE
20 SET FILL STYLE '*'
30 SET FILLY .5
40 FLOT (0,0),(1,1)
50 END

```



Ready

SET FILLX

Syntax

SET FILLX x

where:

x is the numeric expression that specifies the horizontal coordinate of a vertical line to fill to.

Example of Syntax

SET FILLX .9

Purpose

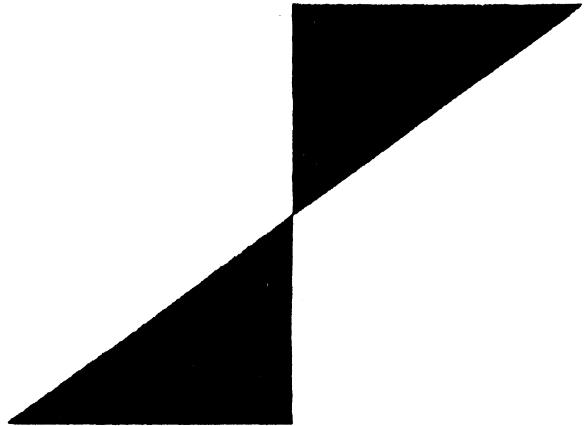
Specifies a vertical line to fill to.

Comments

- ☐ The horizontal position of the line is specified in window coordinates.
- ☐ The fill pattern is affected by the current fill style and by the current writing mode. See SET FILL STYLE, SET LINE STYLE, and SET WRITING MODE.
- ☐ All graphics executed after this statement will be filled to the specified line. Fill is done with the foreground color, and is solid fill by default.

Example

```
set fill* .5  
plot (0,0),(1,1)
```



Ready

SET FILLY

Syntax

SET FILLY y

where:

y is the numeric expression that specifies the vertical coordinate of a horizontal line to fill to.

Example of Syntax

SET FILLY .35

Purpose

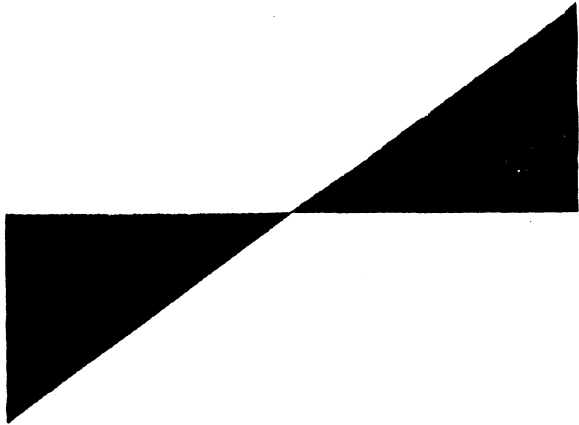
Specifies a horizontal line to fill to.

Comments

- The vertical position of the line is specified in window coordinates.
- The fill pattern is affected by the current fill style and by the current writing mode. See SET FILL STYLE, SET LINE STYLE, and SET WRITINGMODE.
- All graphics executed after this statement will be filled to the specified line. Fill is done with the foreground color, and is solid fill by default.

Example

```
set fillg .5  
plot (0,0),(1,1)
```



Ready

SET FONT

Syntax

SET FONT integer

where:

integer is 0, 1 or 2. 0 indicates the DEC Multinational Character Set, 1 and 2 indicate user-definable fonts.

Example of Syntax

SET FONT 2

Purpose

Selects a font.

Comments

- SET FONT is used to select a character set, and causes GRAPHIC PRINT statements to display characters from the selected font.
- Each font defines the appearance of 95 characters. The characters in these fonts are defined with different dimensions:

Font 0 characters are 12 pixels wide and 10 pixels high.

Font 1 characters are 16 pixels wide and 16 pixels high.

Font 2 characters are 8 pixels wide and 8 pixels high.

The characters of font 0 cannot be changed. The other fonts can be defined by using SET CHARACTER.

- The DEC Multinational Character Set (font 0) is the default.
- The only characters of the DEC Multinational Character Set that can be defined by the user are those from the space character (ASCII 32) to the tilde character (ASCII 127). No other characters can be redefined. See Appendix A for a table of the DEC Multinational Characters and their values.
- See SET CHARACTER in this chapter for more information and examples.

SET ITALICS

Syntax

SET ITALICS expression

where:

expression is a positive or negative numeric expression indicating the angle at which characters are slanted.

Example of Syntax

SET ITALICS 22

Purpose

Determines the forward or backward slant of the text.

Comments

- The SET ITALICS statement will slant text on a horizontal line to the right (clock-wise) if a negative value is provided, and to the left (counter-clockwise) if a positive angle is provided. Normally characters have a slant angle of zero.
- Each character is slanted to the left or right, but the positions of successive characters are not changed. That is, the angle of lines of text is not changed. See SET TEXT ANGLE.
- The default slant angle is 0.
- The text is most readable if the angle is within 45 degrees and -45 degrees.

Example

```

10 SET CHARACTER SIZE .04,.08
20 SET ITALICS -45
30 SET POSITION (.2,.8)
40 GRAPHIC PRINT 'forward slant'
50 SET ITALICS -15
60 SET POSITION (.2,.7)
70 GRAPHIC PRINT 'forward slant'
80 SET ITALICS 0
90 SET POSITION (.2,.6)
100 GRAPHIC PRINT 'no slant'
110 SET ITALICS 15
120 SET POSITION (.2,.5)
130 GRAPHIC PRINT 'backward slant'
140 SET ITALICS 45
150 SET POSITION (.2,.4)
160 GRAPHIC PRINT 'backward slant'
170 PRINTSCREEN

```

```

run
Ready

```

forward slant
forward slant
no slant
backward slant
backward slant

SET LINE STYLE

Syntax

SET LINE STYLE integer

where:

integer is an integer variable or an expression which specifies a line style.

Example of Syntax

SET LINE STYLE 2

Purpose

Select line style.

Comments

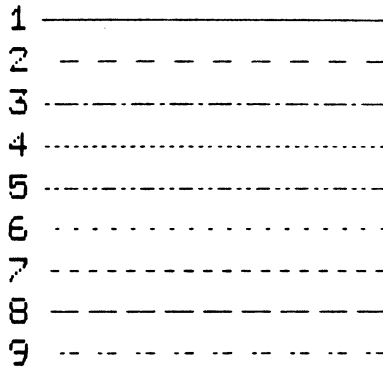
- ☐ Select line style by specifying an integer in the range 1 - 9 from the list below:
 - 1 solid (default)
 - 2 dashed
 - 3 dot-dashed
 - 4 dotted
 - 5 dot-dot-dashed
 - 6 dotted (wide spacing)
 - 7 dashed (short lines)
 - 8 dashed (long lines)
 - 9 dot-dashed (short lines)
- ☐ The line style is reset to solid (style 1) whenever you run a program.
- ☐ Selecting values beyond the valid range (1 - 9) results in selection of line style 1 (solid).

Example

```
10 SET CHARACTER SIZE .04,.08
20 FOR L_STYLE=9 TO 1 STEP -1
30   SET LINE STYLE L_STYLE
40   HEIGHT=1-(L_STYLE/10)
50   SET POSITION (.2,HEIGHT-.04)
60   GRAPHIC PRINT L_STYLE
70   PLOT (.3,HEIGHT),(.9,HEIGHT)
80 NEXT L_STYLE
90 PRINTSCREEN
```

run

Ready



SET POSITION

Syntax

SET POSITION (x,y)

where:

x,y are numeric expressions indicating the new horizontal and vertical coordinate location of the starting point.

Example of Syntax

SET POSITION (.5,.75)

Purpose

SET POSITION sets the specified position as the starting point for graphics.

Comments

- ☐ The new position is given in terms of the current window coordinate system. See SET WINDOW.
- ☐ The position is reset to (0,0) whenever you run a program.

SET TEXT ANGLE

Syntax

SET TEXT ANGLE integer

where:

integer is an integer expression representing the angle at which graphic text prints.

Example of Syntax

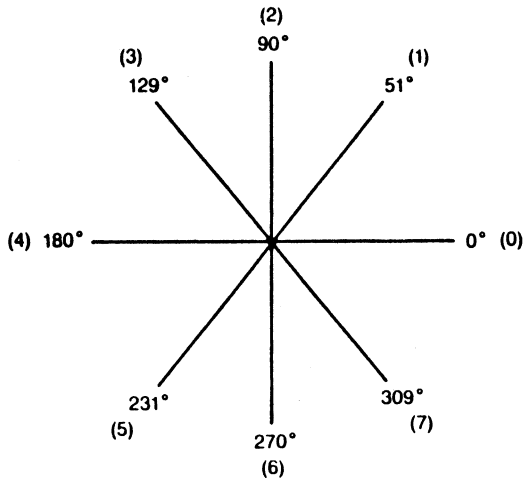
SET TEXT ANGLE 4

Purpose

To specify the angle at which a line of graphic text is printed.

Comments

- ☐ Select from the following eight valid values, each value corresponds to an angle as shown below:



- ☐ The text angle is reset to 0 each time you run a program.
- ☐ If the number specified does not fall in the range of 0 to 7 the text angle is not changed.
- ☐ Contrast SET ITALICS which determines the shift to left or right of each character but does not change the angle of the entire line of text.

Example

```

5 REM Print text at different angles
10 PLOT (.5,.5)
20 SET TEXT ANGLE 4
30 GRAPHIC PRINT 'text at 180 degrees'
40 SET TEXT ANGLE 0
50 GRAPHIC PRINT 'text at 0 degrees'
60 END
run

```

text at 0 degrees

text at 180 degrees

Ready

SET VIEWPORT

Syntax

SET VIEWPORT x min, x max, y min, y max

where:

x min, x max	are the coordinates of the horizontal dimension of the viewport: values are valid in the range of 0 to 1.
y min, y max	are the coordinates of the vertical dimension of the viewport: values are valid in the range 0 to .625.

Example of Syntax

SET VIEWPORT 0,.5,0,.5

Purpose

Specifies the region of the screen which can be used for graphics.

Comments

- SET VIEWPORT sets the clipping region boundaries. When clipping is on, any graphics drawn beyond the viewport boundaries will not appear. When clipping is off, graphics drawn beyond the viewport boundaries are displayed. (Refer to the SET CLIP statement in this chapter.)
- The default viewport is a square filling the right side of the screen. Following are the default values:

SET VIEWPORT .375,1,0,.625

- SET VIEWPORT fails if you attempt to specify a viewport that does not lie completely within the screen, that is, if either x min or y min is less than 0, or if x max is greater than 1, or if y max is greater than .625.

Example

```
10 SET VIEWPORT .5,1,0,.5
20 PLOT (0,0),(1,1)
```

SET WINDOW

Syntax

SET WINDOW x min, x max, y min, y max

where:

x min, x max are the minimum and maximum values of the scale on the horizontal dimension of the viewport.

y min, y max are the minimum and maximum values of the scale on the vertical dimension of the viewport.

Example of Syntax

SET WINDOW 0,10,-1,.5

Purpose

Specifies a range of values applied to the horizontal (x) and vertical (y) dimensions.

Comments

- ☐ The window is set along the current viewport boundaries specified with SET VIEWPORT.
- ☐ The default window coordinates are 0 to 1 in both the horizontal and vertical directions, as shown below:

SET WINDOW 0,1,0,1

- ☐ SET WINDOW sets the clipping region boundaries.

Example

```

1 REM plot y = sin(x) and y=log(x)
20 CLEAR
25 SET VIEWPORT .1,.45,.1,.45 \ REM
30 SET WINDOW -PI,PI,-2,2 \ REM
40 PLOT (-PI,0),(PI,0) \ REM
50 PLOT (0,-2),(0,2)
70 FOR X=-PI TO PI STEP .1 \ REM
80 PLOT (X,SIN(X)),
90 NEXT X
100 REM
110 SET VIEWPORT .55,.9,.1,.45 \ REM
120 SET WINDOW 0,50,-5,5 \ REM
150 PLOT (0,-5),(0,5) \ REM
160 PLOT (0,0),(50,0)
170 FOR X=0 TO 50 STEP .5 \ REM
190 PLOT (X+,1,LOG(X+.1))
200 NEXT X
500 END

```

area for graphics on left
scale for 1 sine wave
plot axes

plot the points $(x, \sin(x))$

now graphics on right
and scaled for $\log(x)$
plot the axes

plot the points $(x, \log(x))$

SET WRITING MODE

Syntax

SET WRITING MODE integer

where:

integer is an integer expression which specifies a writing mode.

Example of Syntax

SET WRITING MODE 4

Purpose

Selects the writing mode for use with PRO/BASIC graphics.

Comments

- There are ten different writing modes, each associated with a number (0 through 9). They are as follows:

0,1	transparent, transparent negate
2,3	complement, complement negate
4,5	overlay, overlay negate
6,7	replace, replace negate
8,9	erase, erase negate

- Some writing modes make a distinction between the foreground of the pattern being drawn and the background of the pattern being drawn. Some patterns (solid patterns, like a solid line) are all foreground.

Other patterns (like a dotted line, or a character) have both a foreground and a background part. In a dotted line, the dots are the foreground and the holes between the dots are the background. In a character, the strokes that make up the character are the foreground, while the rest of the box within which the character is drawn form the background.

- Negated modes switch what they do with the foreground and background of the pattern. For example, complement mode complements the foreground pattern and leaves the background pattern alone. Complement negate mode complements the background pattern and leaves the foreground alone. If a mode makes no distinction between foreground and background, then the negated mode is the same as the unnegated mode. Transparent mode and transparent negate mode, for example, are identical.

- **Transparent (mode 0)**
Draws nothing, but changes the current position as if it were.
- **Complement (mode 2)**
Draws the pattern specified using the complement of the color at the current drawing position on the screen. Complement mode thus does not depend on the color being used. Here is a list of the pairs of complementary colors (and their default appearance):

0 and 4 (black and yellow)
 1 and 6 (red and light blue)
 2 and 5 (green and magenta)
 3 and 7 (blue and white)

If you were plotting a dotted line, the dots would be complemented, and the holes would be left alone.

- **Overlay (mode 4)**
Draws the pattern specified using the current color. Only the foreground of the pattern is written, the background is left the original color.
- **Replace (mode 6)**
Writes the pattern using the current color (as overlay does) and writes the background of the pattern using the background color.
- **Erase (mode 8)**
Writes the pattern using the background color.
- **The writing mode is reset to overlay (4) whenever you run a program.**
- **Complement mode is useful when you wish to draw some image temporarily, and remove the image leaving the original image. This can be accomplished by writing the image twice in complement mode, since the complement of the complement of a color is the original color.**

Example

```

10 SET CHARACTER SIZE .2,.2\ SET POSITION (.2,.3)
15 GRAPHIC PRINT 'AB'
17 SET POSITION (0,.05)
20 GRAPHIC PRINT "A"\ GRAPHIC PRINT "B"\ REM The default mode is OVERLAY
30 SET WRITING MODE 6
40 SET POSITION (.6,.05)
50 GRAPHIC PRINT "A"\ GRAPHIC PRINT "B"\ REM Replace 'A' with 'B'
60 END

```

run



Ready

Appendix A

Appendix A

The DEC Multinational Character Set

This appendix contains a table of the DEC Multinational Character Set. Octal, decimal, and hexadecimal numeric codes appear next to each character.

The DEC Supplemental Graphics Characters appear in columns 10 - 15. These characters can only be used as literals in quoted strings.

APPENDIX A | THE DEC MULTINATIONAL CHARACTER SET

COLUMN				0	1	2	3	4	5	6	7					
BITS				0 0 0 0	0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 1 0 1	0 1 1 0	0 1 1 1					
ROW	14	13	12	11	10	9	8	7	6	5	4	3				
0	0	0	0	0	NUL	0	0	DLE	16	32	48	64	80	96	112	128
1	0	0	0	1	SOH	1	1	DC1 (XON)	21	37	53	69	85	101	117	133
2	0	0	1	0	STX	2	2	DC2	22	38	54	70	86	102	118	134
3	0	0	1	1	ETX	3	3	DC3 (XOFF)	23	39	55	71	87	103	119	135
4	0	1	0	0	EOT	4	4	DC4	24	40	56	72	88	104	120	136
5	0	1	0	1	ENQ	5	5	NAK	25	41	57	73	89	105	121	137
6	0	1	1	0	ACK	6	6	SYN	26	42	58	74	90	106	122	138
7	0	1	1	1	BEL	7	7	ETB	27	43	59	75	91	107	123	139
8	1	0	0	0	BS	10	8	CAN	30	46	62	78	94	110	126	142
9	1	0	0	1	HT	11	9	EM	31	47	63	79	95	111	127	143
10	1	0	1	0	LF	12	10	SUB	32	48	64	80	96	112	128	144
11	1	0	1	1	VT	13	11	ESC	33	49	65	81	97	113	129	145
12	1	1	0	0	FF	14	12	FS	34	50	66	82	98	114	130	146
13	1	1	0	1	CR	15	13	GS	35	51	67	83	99	115	131	147
14	1	1	1	0	SO	16	14	RS	36	52	68	84	100	116	132	148
15	1	1	1	1	SI	17	15	US	37	53	69	85	101	117	133	149

ASCII GRAPHIC CHARACTER SET

KEY

CHARACTER ESC 33 OCTAL
 27 DECIMAL
 1B HEX

8	9	10	11	12	13	14	15	COLUMN	
1 0 0	1 0 0 1	1 0 1 0	1 0 1 1	1 1 0 0	1 1 0 1	1 1 1 0	1 1 1 1	16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	ROW
200 128 80	DCS	220 144 96	240 160 A0	260 176 B0	280 192 C0	300 208 D0	320 224 E0	340 240 F0	0
201 129 81	PU1	221 145 97	241 161 A1	261 177 B1	281 193 C1	301 209 D1	321 225 E1	341 241 F1	1
202 130 82	PU2	222 146 98	242 162 A2	262 178 B2	282 194 C2	302 210 D2	322 226 E2	342 242 F2	2
203 131 83	STS	223 147 99	243 163 A3	263 179 B3	283 195 C3	303 211 D3	323 227 E3	343 243 F3	3
204 132 84	CCH	224 148 9A	244 164 A4	264 180 B4	284 196 C4	304 212 D4	324 228 E4	344 244 F4	4
205 133 85	NEL	225 149 9B	245 165 A5	265 181 B5	285 197 C5	305 213 D5	325 229 E5	345 245 F5	5
206 134 86	SPA	226 150 9C	246 166 A6	266 182 B6	286 198 C6	306 214 D6	326 230 E6	346 246 F6	6
207 135 87	ESA	227 151 9D	247 167 A7	267 183 B7	287 199 C7	307 215 D7	327 231 E7	347 247 F7	7
210 136 88	HTS	230 152 9E	250 168 A8	270 184 B8	290 200 C8	310 216 D8	330 232 E8	350 248 F8	8
211 137 89	HTJ	231 153 99	251 169 A9	271 185 B9	291 201 C9	311 217 D9	331 233 E9	351 249 F9	9
212 138 8A	VTS	232 154 9A	252 170 AA	272 186 BA	292 202 CA	312 218 DA	332 234 EA	352 250 FA	10
213 139 8B	PLD	233 155 9B	253 171 AB	273 187 BB	293 203 CB	313 219 DB	333 235 EB	353 251 FB	11
214 140 8C	PLU	234 156 9C	254 172 AC	274 188 BC	294 204 CC	314 220 DC	334 236 EC	354 252 FC	12
215 141 8D	RI	235 157 9D	255 173 AD	275 189 BD	295 205 CD	315 221 DD	335 237 ED	355 253 FD	13
216 142 8E	SS2	236 158 9E	256 174 AE	276 190 BE	296 206 CE	316 222 DE	336 238 FE	356 254 FE	14
217 143 8F	SS3	237 159 9F	257 175 AF	277 191 BF	297 207 CF	317 223 DF	337 239 EF	357 255 FF	15

←----- DEC Supplemental Graphic Characters -----→

Appendix B

Appendix B

Keywords

ABS	ELSE	LOG	RENUMBER
AND	END	LOG10	RESTORE
AS	ERL	MERGE	RESUME
ASCII	ERR	MID	RETURN
ASK	ERROR	NAME	RND
ATN	ERT	NEW	RUN
CALL	EXIT	NEXT	SAVE
CAT[ALOG]	EXP	NOT	SCROLL
CCPOS	FILE	NUM	SET
CHAIN	FIX	OLD	SHOW
CHR	FOR	ON	SIN
CLEAR	GO	OPEN	SINGLE
CLOSE	GOSUB	OR	STEP
CONT[INUE]	GOTO	OUTPUT	STOP
COS	GRAPHIC	PI	SQR
DATA	IF	PLOT	TAB
DATE	INPUT	POS	THEN
DECLARE	INT	PRINTSCREEN	TIME
DEF	KILL	PRINTUSING	TO
DEL[ETE]	LEN	PROGRAM	VAL
DIM	LET	RANDOMIZE	VIRTUAL
DIMENSION	LINE	READ	WITH
DOUBLE	LINPUT	REM	XOR
EDIT	LIST	RENAME	

Appendix C

Appendix C

Logical Operators

C.1 NUMERIC VALUES IN LOGICAL EXPRESSIONS

Logical expressions can combine relational and/or other logical expressions, and evaluate them to produce a single true/false result.

Logical expressions can also use logical operators to perform logical operations on numeric values.

C.1.1 True and False Are Actually Numeric Values

The discussion of relational and logical operators in Chapter 2 used the terms true and false to indicate the results produced by relational and logical operations. In fact, true and false are derived from numeric values. PRO/BASIC's logical tests consider a value of 0 to be false and a non-zero value to be true.

PRO/BASIC assigns 0 to a false value and -1 to a true value. In the example below the result of the relational expression ($B\% = C\%$) is assigned to $A\%$.

$A\% = (B\% = C\%)$

The variable $A\%$ is assigned true (-1) if the values in $B\%$ and $C\%$ are equal, or false (0) if the values in $B\%$ and $C\%$ are not equal.

Logical expressions are often used to link relational expressions. Let's look at an example from Chapter 2 of using logical operators with relational operands.

IF ($A > B$) AND ($C < D$) THEN GOTO 20

The relational operands ($A > B$), ($C < D$) are first evaluated to true or false, depending on the values in variables A,B,C, and D. Then the logical AND operation is performed on those two values to produce one final result, either true (-1) or false (0).

C.1.2 Integers in Logical Expressions

Logical expressions can use the true or false values output from relational operations. Logical expressions can also use other numeric values as operands.

When numeric values are given to a logical expression, the 16-bit internal integer representation is operated upon with reference to the particular logical operator. (A bit is a digit used in the binary number system; it has a value of 0 or 1.)

The logical operations (NOT, AND, OR, XOR) are performed bit by bit on the 16-bit sequences. Each bit result is determined by the two corresponding bits of the two operands, with reference to the truth tables.

The logical NOT operation complements each bit of the sequence, that is, 0 replaces 1 and 1 replaces 0 in the result of a NOT operation.

For example, assume that $J\% = 5$ and $K\% = 6$ in the following example:

$I\% = J\% \text{ AND } K\%$

The logical expression says perform the logical AND operation on $J\%$ and $K\%$, and store the result in $I\%$. The integers are represented in 16 bits:

$5\% = 0000000000000101$ (binary)

$6\% = 0000000000000110$ (binary)

The logical AND operation is performed on the corresponding bits from both sequences of numbers. The result is:

0000000000000100 (binary)

This result is then stored in $I\%$. If $I\%$ is accessed as a number, its value is 4 (decimal); if accessed as a logical, its value is true because $I\%$ contains a non-zero value.

The results produced by a logical operator can then be tested. When PRO/BASIC evaluates an IF statement containing a logical integer variable, a non-zero value is considered true, and 0 is considered false. For example, this statement:

```
100 IF A% < > 0% THEN RETURN
```

can be replaced with:

```
100 IF A% THEN RETURN
```

The second statement executes faster because the value tested is simpler and requires fewer operations.

C.1.3 The Logical Complement of -1 (true) is 0 (false)

Note that logical expressions with integer operands other than -1 and 0 can produce unpredictable results. For example, in the logical NOT operation that follows, A% is true in both cases because in both cases A% is not equal to zero. The binary and true/false values of A% are shown below.

A%	= 0100100001000001	true
NOT A%	= 1011011110111110	true

A% is true and its logical complement is also true.

Another example:

```
IF X% AND Y% THEN RETURN
```

The decimal, binary, and true/false values of X% and Y% are shown below.

X%	= 2877	= 0000101100111101	true
Y%	= 9216	= 001010001000010	true
		0000000000000000	false

When the logical AND operation is performed on these integer values, the result is false, which is obviously incorrect. Use only -1 and 0 to avoid problems with logical expressions in logical operations. For example:

```
10 FALSE% = 0%
20 TRUE% = -1%
.
.
.
150 DONE% = TRUE%
.
.
.
200 IF NOT DONE% THEN 900
```

Logical operations performed on these values will perform predictably.

C.2 MASKING

Masking is a useful technique that allows only specified bits to be visible to a program, so that only specified bits will be tested for the value they contain. These bits can then be tested, set, or cleared. For example, the function in line 10 below creates a mask that causes only the rightmost, or 3 least significant, bits to be used. Bits 4 through 16 are ignored.

```

10 DEF FNMOD8(NUMBER%) = NUMBER% AND 7%
20 INPUT I%
30 PRINT FNMOD8(I%)
RUN
? 37
5

```

The function FNMOD8 performs a logical AND operation on 7—the mask—and on 37—the value input. The decimal and binary representations of 37 and 7 and the binary and decimal representation of the result, 5, are shown below:

```

37 = 0000000000100101
7  = 0000000000000111
    000000000000101 = 5 (decimal)

```

Appendix D

Appendix D

Advanced Programming Techniques

PROBASIC.BAS

PROBASIC.BAS is a file that is accessed each time PRO/BASIC is run. You can include any PRO/BASIC program statement(s) for execution each time PRO/BASIC is run. Do not include commands or immediate mode statements in PROBASIC.BAS.

You can use PROBASIC.BAS, for example, to print text on the screen when PRO/BASIC is run.

Comment fields

Comment fields are an alternative to the use of REM statements to include comments in a program. They are indicated by the exclamation point symbol (!). PRO/BASIC ignores anything following the comment field.

The difference between comment fields and the REM statement is that when a program with comment fields is listed from PRO/BASIC's memory the comment fields are removed. This is useful when you wish to comment a program fully but do not want to sacrifice too much space in memory.

Because comment fields are removed when the program is listed in PRO/BASIC's memory, use the Professional text editor, PROSE, to list or edit the program, to keep the comment fields in the program.

Comment fields will display when you view the program with PROSE.

Appendix E

Appendix E

PRO/BASIC Program Error Messages

1 Invalid directory for device

The device directory is unreadable or does not exist. Hardware error possible on device.

2 File name not acceptable

File specification is not valid. Check for imbedded blanks, special characters, proper format in the file specification:

device: <directory>filename.type;version

3 Required device is in use

Wait for the device to finish.

4 No room for file on device

There is not enough space for the file. Either the file cannot fit in the available space on the device, or the device directory is full and cannot accept another file name. Delete unnecessary files.

5 File or directory of file not found

No file by that name, or the directory of the file does not exist. Check for spelling errors in the file specification, be certain that the requested file is in the current directory.

6 Not a valid device

No device by that name. Check for correct device name.

7 Channel is already open

A file was already open on this channel when OPEN was executed. Close file on channel.

8 Device not available

Required device is not currently available.

9 Channel is not open

File must be opened on this channel before the file can be accessed.
Open file on channel.

10 File is protected or diskette is not in drive

File is protected against specified action, or there is no diskette in the specified drive. Protection may be changed by using the file services.

11 End of file

All records in the file have been read. No more records can be read.

12 Input or output error

Possible hardware error during attempted operation. Check your system with the System Maintenance diskette.

14 Device is write locked

You can not write to files on this device as long as it is locked.

16 File name already in use

Use a different file name or a different file version, or delete the old file.

28 INTERRUPT-DO keys entered

The INTERRUPT-DO keys were entered at the keyboard.

45 Virtual array not yet open

The disk file containing the virtual array is not open. An OPEN command is needed to access a virtual array file.

46 Invalid channel number

The channel number is outside valid range. Channel numbers must be positive and less than 16; channel 0 is reserved for the terminal.

47 Input line is too long

Keep terminal input lines shorter than 80 characters, and file records shorter than 132 characters.

49 Exponent is too large

The EXP function ("e" raised to a power) overflows with numbers larger than approximately 88. Exponents have low limits in general.

50 Data has invalid format

Strings must have matching quotes or no quotes at all, and numbers may not have quotes. Check that there are no characters between a quoted string and the end of the line or the next comma.

51 Integer overflow

Integers can not be greater than 32767, or less than -32767. If values outside this range are needed, use real numbers.

52 Invalid number

The input is not a number, is too larger or small, or is an incorrectly formed number; for example, "\$1.00" is not a valid number.

53 LOG can not accept this argument

A negative or zero value argument can not be passed to LOG or LOG10. Use positive values.

54 The square root of a negative number is not defined

Taking the square root of a number less than zero is not possible. Use the ABS function to insure that numbers are never negative, or check for negative values.

55 Subscript of array out of bounds

A subscript used in an array reference was less than 0 or larger than the array dimension. Check for unexecuted DIM statements. Dynamic DIM statements can change array sizes, so make sure the array is the expected size.

57 End of data

All data in the program has been read. Check for more READs than data in DATA statements, or wrong line number specified for RESTORE.

58 Expression in ON statement out of bounds

The index value in an ON GOTO or ON GOSUB statement is less than 1 or greater than the number of line numbers in the statement. Check that the index value is in the correct range.

59 INPUT statement required more data than present in record

Input statement has more variables than elements in the record. Change the number of variables to equal the number of elements in the record, or change the record.

61 Division by zero is not defined

The program attempted to divide a quantity by zero. Change the program logic or trap the error with an error handler.

70 Stack overflow

The program's logic requires too many locations to be recorded for program control. Check for missing RETURN statements, jumping out of a FOR/NEXT loop, or FOR/NEXT or GOSUB nested too deeply.

71 The specified line can not be found

The line may have been deleted or renumbered. LIST the program to check the actual line number.

72 No subroutine called; can not RETURN

Program executed a RETURN statement, but was not executing a subroutine. Check program logic. Use GOSUB to enter a subroutine. Use RETURN to exit a subroutine.

74 An undefined function is referenced

Functions must be defined before they are referenced.

88 Arguments do not match

A function or subroutine received an argument of the wrong type. For example, string instead of numeric, or two-dimensional array when one dimension was expected.

89 Expected fewer arguments

Too many arguments passed. For example, an array can not be defined with more than 7 arguments; the MID\$ function should have 3 arguments.

92 Matching NEXT must follow this FOR

The program has a FOR statement without a NEXT statement, or an incomplete FOR/NEXT within a FOR/NEXT.

93 NEXT does not match immediately preceding FOR

The program has a NEXT statement without a FOR statement.

97 Expected more arguments

Too few arguments passed. For example, POS requires 3 arguments.

99 Expected a string

The program required a string where a numeric value was supplied.

100 Expected a number

The program required a number where a string value was supplied.

104 Encountered RESUME when not handling an error

Encountered a RESUME when not handling an error. Isolate error-handling subroutines from normal processing with GOTO statements.

106 Inconsistent subscript usage

The number of subscripts of an array or a function can not change during the execution of a program.

109 Unsaved changes, type EXIT again to exit

Changes to a program are lost if you leave PRO/BASIC without saving the program. Type EXIT again to leave without saving the changes.

110 Expected an array

The DIM statement establishes the dimension of arrays. You can not dimension a function.

111 No program to run

There is no program in memory.

112 Invalid line range

A line range is of the form "number-number", where both numbers are positive and less than 32768, and the first number is less than the second.

113 Strings and numbers can not be compared

Comparison between strings and numbers is not defined.

114 Can not continue

The program can not continue because it is at the end, or an error has occurred. Use RUN or GOTO to start again.

115 Can not pass an entire virtual array

Virtual arrays can not be passed to subroutines or to other programs. Use normal arrays.

116 PRINT USING format error

No field descriptor found in PRINT USING format string.

117 Too many FOR/NEXT statements nested

FOR/NEXT statements can be moved into subroutines to decrease the number of nested FOR/NEXT statements.

118 Implementation error

This is an error in the implementation of PRO/BASIC.

120 PRINT USING buffer overflow

A field or literal string in a PRINT USING format string is too large. Decrease the size of the string to be printed.

123 STOP

The program has executed a STOP statement. Type CONTINUE to continue.

126 No memory available

The program required more memory than was available. To gain memory, deallocate arrays, close files, use fewer files at a time, or split up the program by using CHAIN.

138 File is locked

The file was not properly closed. Use the UNLOCK disk service to unlock the file.

141 Invalid operation on file

The program attempted to PRINT, INPUT, or LINPUT from a virtual array file. Virtual array files can only be accessed as arrays.

149 Not at the end of the file

The program attempted to write to a file when not at the end of the file. Do not write to a file until the end has been reached by executing INPUT or LINPUT statements.

227 String too long

The maximum length of a string is 255. Decrease the length of the string.

228 Wrong type of file

You can not open a virtual array file without stating VIRTUAL in the OPEN, or open a document file while stating VIRTUAL in the OPEN.

234 Renumbered line matches specified line

A reference to a nonexistent line was found. It was not changed, but a renumbered line now matches it.

240 Invalid redimension of array

An array that was declared with constants can not be redeclared. Use variables as subscripts in all declarations of the array if the array will be redeclared.

241 Numeric overflow

Result of an operation is a real number too large to be represented. The largest representable value is approximately $1.7\text{E}+38$.

242 Numeric underflow

Result of an operation is a real number too small to be represented. The smallest representable value is approximately $.3\text{E}-38$.

243 CHAIN to nonexistent line

The line referred to in the CHAIN statement does not exist.

249 Argument out of bounds

One or more of the arguments in a statement is outside the valid bounds, or a range is specified from high to low rather than from low to high.

250 An unimplemented feature

The program has tried to use a feature that is not implemented.

253 A system directive failure

The system failed to execute an action required by PRO/BASIC. Check your system with the System Maintenance diskette.

Index

Index

- ABS, 176
- Accuracy, 17, 21
- Adding a program line, 5
- Algebraic functions
 - ABS, 176
 - EXP, 191
 - FIX, 192
 - INT, 194
 - LOG, 197
 - LOG10, 198
 - SQR, 210
- AND, 25
- Appending to a file, 50, 53
- Arithmetic Operators, 22
- Arrays
 - see Initialization
 - defined, 35
 - subscripted variables, 21
- ASCII (function), 177
- ASK POSITION, 227
- Assigning values
 - see LET statement
 - to arrays, 36
 - to variables, 18
 - to virtual arrays, 36
- ATN, 179
- CALL COLLATE, 98
- CALL INKEY, 129
- CATALOG, 70
- CCPOS, 180
- CHAIN, 100
- Chaining, 61
 - see PROGRAM statement
- Changing a program line, 5
- Choosing PRO/BASIC from a menu, 1
- CHR\$, 182
- CLEAR, 228
- Clipping, 247, 270
- CLOSE, 49, 103
- Commands
 - defined, 69
- Comment Fields, 295
- Comparison, string, 29
- Concatenation, 29
- Conditional transfer, 55
 - IF statement, 126
- Constants
 - numeric, 15
 - string, 16
- CONTINUE, 72
- Control characters, 4, 65
- Control functions, 64
- CONTROL key, 4
 - see Using control functions
- Conversion functions
 - ASCII, 177
 - CHR\$, 182
 - NUM\$, 201
 - VAL, 214
- Conversion when handling numbers, 20
- Coordinates, 220-226
- COS, 184
- Cursor, 2
- Cursor control keys, 3
- DATA, 104
- DATE and TIME functions
 - DATE\$, 185
 - TIME\$, 213
- DATE\$, 185

DEC Multinational Character Set, 15, 262,
279

DECLARE, 106

DEF, 108

DELETE, 73

DELETE key, 3, 45

Deleting a program line, 6

Deleting characters, 3

DIM, 111

DIM #, 114

Displaying the program, 5

EDIT, 74

EDIT\$, 186

END, 118

Entering a line, 3

ERL, 188

ERR, 189

Error Handling

ERL (function), 188

ERR (function), 189

ERT\$ (function), 190

ON ERROR GOTO, 52, 139

ERT\$, 190

Escape sequences, 65

EXIT, 75

EXIT key, 4

EXP, 191

Exponential notation, 16

Expressions, 22

Extended bit-map option (EBO), 219

File

defined, 38

device name, 41

directory name, 41

fields, 38

name, 39

records, 38

sequential, 39, 147

type, 39

versions, 40

virtual array, 39, 148

File handling functions

CCPOS, 180

TAB, 211

FIX, 192

FOR/NEXT, 119

see Loops

Functions

control, 64

defined, 30

table of, 30

User defined (DEF), 108

GOSUB, 123

GOTO, 125

GRAPHIC PRINT, 229

Halting program execution, 60

HELP key, 4

Identifiers, 19

IF, 126

Immediate mode, 2

Implicit arrays, 38

Initialization, 18

arrays, 112, 116

variables, 107

INPUT, 45, 130

INPUT #, 49, 130

INT, 194

Integer constants, 16

Integer variables, 19

INTERRUPT/DO keys, 61

Keywords, 15, 285

KILL, 132

LEN, 196

LET, 133

Line Editor, 2

Line length, 14

Line numbers, 13

LINPUT, 46, 135

LINPUT#, 49, 135

LIST, 76

LOG, 197

LOG10, 198

Logical expressions, 25

Logical names, 41

Logical operators, 25, 289-292

Loops

explained, 57

FOR/NEXT, 58, 119

Making changes to a program, 5

Memory, 9

MERGE, 77

MID\$, 199

Modes of Operation

Immediate mode, 2

Program mode, 2

Monitor, 219

Moving the cursor, 2

Multiple branching, 54

ON GOTO statement, 145

NAME AS, 137
 Naming files, 39
 NEW, 80
 NEXT, 138
 NOT, 25
 NUM\$, 201
 Numeric constants, 16-17
 Numeric relational expressions, 24

OLD, 81
 ON ERROR, 139
 ON GOSUB, 142
 ON GOTO, 145
 OPEN, 48, 147
 Operands, 22
 Operators
 arithmetic, 22-23
 logical, 25-26
 relational, 24
 OR, 25
 Order of Precedence, 22, 26

PI, 203
 Pixel, 240
 PLOT, 231
 PLOT ARC, 233
 PLOT CURVE, 235
 POS, 205
 Precision, 16-17
 PRINT, 43, 149
 and control functions, 64
 and print format, 43
 and TAB function, 211
 Print margins, 44
 PRINT USING, 152
 PRINT#, 49, 149
 Printing to line printer, 151
 PRINTSCREEN, 237
 PRO/BASIC Character Set, 15
 see Appendix A
 PROBASIC.BAS, 295
 PROGRAM, 158
 Program Control, 53-59
 Program Documentation, 14, 295
 Program line length, 14
 Program lines, 14
 Program mode, 2

RANDOMIZE, 160
 READ, 162
 READ and DATA, 47
 Real constants, 16
 Real variables, 19
 Relational operators, 24
 REM, 164
 RENAME, 82
 RENUMBER, 83

RESTORE, 165
 RESUME, 166
 RETURN, 167
 RND, 207
 see RANDOMIZE
 RUN, 86

SAVE, 87
 SCROLL, 238
 SET, 88
 SET CHARACTER, 240
 SET CHARACTER SIZE, 243
 SET CHARACTER SPACING, 245
 SET CLIP, 247
 SET COLOR, 248
 SET COLORMAP, 250
 SET CURRENCY, 168
 see PRINT USING
 SET FILL, 253
 SET FILL OFF, 255
 SET FILL STYLE, 256
 SET FILLX, 258
 SET FILLY, 260
 SET FONT, 262
 SET ITALICS, 263
 SET LINE STYLE, 265
 SET POSITION, 267
 SET RADIX, 169
 see PRINT USING
 SET SEPARATOR, 170
 see PRINT USING
 SET TEXT ANGLE, 268
 SET VIEWPORT, 270
 SET WINDOW, 271
 SET WRITING MODE, 273
 Settable modes, 88
 SHOW, 90
 SIN, 209
 Special characters, 15
 SQR, 210
 Statement separators, 14
 Statements,
 defined, 14
 STEP, 92
 STOP, 60, 171
 String comparison, 29
 String concatenation, 29
 String constants, 15
 String functions
 EDIT\$, 186
 LEN, 196
 MID\$, 199
 POS, 205
 String relational expressions, 29

Prodotto : PRO/BASIC V1.2

Codice : QBA04-A3

Qta	Codice	Descrizione del componente
1	BH-N199D-TH	DCSPD (40.005.03)
1	BL-N606C-BH	PRO/BASIC V1.2 BIN RX50
1	AA-N601C-TH	PRO/BASIC LANGUAGE MANUAL RELEASE NOTES
1	AV-U706A-TH	REFERENCE CARD
1		LISTA DEI COMPONENTI
1		P.E.R.
1		CONTRATTO DI LICENZA SOFTWARE
1		BUSTA PER INVIO CONTRATTO DI LICENZA SOFTWARE

Sommario:

1	Dischetto(i)	1	Manuale(i)	1	Licenza Sw/Garanzia
0	Nastro(i) mag.	1	Scheda riferimento	1	DCSPD
				3	Doc/Hw varia

Note :

(C)Digital Equipment Corporation

HW : PC300 (PROFESSIONAL)

(E) Classificazione SW : Supportato

: P/OS

(F) Durata garanzia : gg 365

Dispositivo di protezione del software : Assente

Ultima modifica | Data
04-FEB-85 |

Software Product Description

DIGITAL CLASSIFIED SOFTWARE

Nome del Prodotto: **PRO/BASIC**, Versione 1.2
per Professional

DCSPD 40.005.03

PRO/BASIC è un prodotto sviluppato e distribuito dalla Digital Equipment Corporation.

Descrizione

Il BASIC è un linguaggio di programmazione di tipo interattivo, sviluppato al Dartmouth College.

Il linguaggio usa per eseguire le operazioni, semplici istruzioni molto simili alla lingua inglese corrente e notazioni matematiche assai familiari.

PRO/BASIC è il nome del BASIC interattivo adottato per il sistema Professional 300. Le sue funzioni rispecchiano gli standard adottati dall'industria dei personal per il linguaggio BASIC, mantenendo tuttavia un alto grado di compatibilità con il BASIC disponibile sui più grossi sistemi Digital PDP-11 e VAX.

Le caratteristiche del PRO/BASIC sono:

- Nomi a lunghezza variabile (fino a 31 caratteri)
- Archivi sequenziali
- Archivio ad accesso diretto (matrici virtuali)
- Concatenamento di un programma ad un altro con passaggio parametri
- Possibilità di aggiornare e modificare le linee di istruzione a video
- Controllo immediato della sintassi durante l'inserzione dell'istruzione
- Strumenti per controllare la correttezza dei programmi con controllo immediato dei passi d'esecuzione del programma stesso
- Singola e doppia precisione che permettono una accuratezza pari a 6 cifre ed a 16 cifre rispettivamente
- Matrice a dimensioni multiple (fino a sette)
- Istruzioni estese IF - THEN - ELSE
- Possibilità di scrivere più istruzioni su una medesima riga
- Possibilità di stampa tipo calcolatrice
- Formato di stampa con l'istruzione PRINT USING
- Gestione programmabile degli errori (ON ERROR GO TO)
- Funzioni definibili dal programmatore

P/O/S è un marchio registrato della Digital Equipment Corporation.

Ottobre 1984
BH-N199D-TH

software

- Funzioni aggiuntive per la grafica (più di 20 istruzioni)
- Supporto per gestione di stringhe e matrici di stringhe
- Messaggi di errore significativi, ad es. "È prevista la virgola" anziché "Errore di sintassi"
- Guida interattiva alle operazioni

PRO/BASIC mette a disposizione dell'utente 18 Kbyte di memoria nell'area di lavoro. L'aggiunta di una scheda per l'espansione della memoria non permette di incrementare questo limite.

Hardware necessario

Qualsiasi configurazione valida di Professional 300 con:

- 512 Kbyte di memoria
- Adattatore per virgola mobile

Hardware addizionale

Modulo di estensione della memoria video.

Software necessario

Sistema Operativo P/OS a Disco Rigido, Versione 1.7

Software addizionale

Nessuno

Installazione

A cura dell'utente.

Dispositivi di protezione del software

Assenti.

Forme e modi d'ordine

Il software in codice binario viene fornito in licenza di uso secondo le condizioni generali espresse nel "Contratto di licenza di prodotti Software Digital della Libreria DCS" che deve essere sottoscritto per ogni copia del software ordinato.

QBA04-A3 Versione inglese, licenza singola, moduli binari
su RX50, documentazione.

GARANZIA

Prodotto software

I prodotti software sono classificati dalla Digital nel Quadro "E" del "Contratto di licenza di Prodotti Software Digital della Libreria DCS" come segue:

- Prodotti Software contrassegnati con "A"
La Digital renderà gratuitamente disponibile su di essi, tramite il proprio Centro di Assistenza Telefonica (CAT-PC) assistenza e/o consulenza concernenti:
 - a) le modalità d'impiego sul Sistema di detti Prodotti Software;
 - b) La redazione da parte del Cliente di correzioni di errori o modifiche, anche temporanee, necessarie per eliminare o limitare gli effetti di eventuali difformità dei Prodotti Software stessi rispetto alle relative specifiche funzionali di cui al DCSPD.

Il servizio verrà reso esclusivamente per telefono o per corrispondenza sulla base dell'analisi di errore che il Cliente fornirà al CAT-PC all'atto della richiesta di intervento, se del caso documentando i dati per iscritto tramite appositi moduli (P.E.R.) in conformità alle relative procedure standard Digital indicate o richiamate nella Documentazione.

Quanto sopra all'espressa condizione che i Prodotti Software Digital siano in versione standard Digital non modificata.

La garanzia avrà durata pari al numero di giorni indicato nel Quadro "F" del "Contratto di licenza di Prodotti Software Digital della Libreria DCS" con decorrenza dalla data di consegna al Cliente dei Prodotti Software.

Il servizio esclude espressamente la soluzione di problemi applicativi.

Il Rivenditore e/o la Digital non garantiscono che, nonostante il diligente intervento di quest'ultima tutti gli errori potranno essere corretti e le difformità superate, ovvero, che lo saranno nei tempi e modi voluti dal Cliente.

— Prodotti Software non contrassegnati con "A"

Sono forniti nello stato in cui si trovano, senza obblighi di garanzia o di altra natura da parte della Digital, del Rivenditore e/o degli eventuali terzi Produttori.

La Digital e/o i Rivenditori Digital locali, forniranno eventualmente dietro pagamento del corrispettivo e previa richiesta del Cliente, versioni aggiornate del prodotto software che la Digital stessa renderà disponibili per l'uso generalizzato da parte dei propri Clienti.

Per ulteriori servizi rivolgersi alla sede locale Digital o al Rivenditore Digital locale.

Supporti magnetici

La Digital garantisce contro difetti di materiale o di lavorazione i supporti magnetici dei Prodotti Software forniti con questi ultimi, per un periodo di 90 (novanta) giorni dalla data di consegna.

In base a tale garanzia l'unica obbligazione della Digital sarà quella di sostituire i supporti magnetici che risultino difettosi.

La sostituzione avverrà per corrispondenza.

Product
Exception
Report

 DIGITAL CLASSIFIED SOFTWARE

Modulo per la segnalazione di anomalie del prodotto Software:

Nome del prodotto _____ Codice del prodotto _____

Sistema Operativo _____ Versione _____

Modello della CPU _____ Serial No. _____

Supporto magnetico _____ Grandezza della memoria _____

Terminale sistema _____ Configurazione del sistema _____

È possibile riprodurre, a comando, l'anomalia riscontrata? Si _____ No _____

Spazio riservato alla descrizione delle anomalie riscontrate:

software



Nome _____

Società _____

Indirizzo _____

Telefono _____

Data Spedizione _____

Nome rivenditore _____

SPEDIRE A:

Digital Equipment S.p.A.

CAT - PC

V.le Fulvio Testi, 11

20092 CINISELLO BALSAMO (MI)