ATARI
Portfolio™

PORTFOLIO POWERBASIC®

USER'S MANUAL

Portfolio PowerBASIC®

Reference Guide

PRINTED IN THE USA.
10 9 8 7 6 5 4 3 2 1

PRINTED IN THE USA.
10 9 8 7 6 5 4 3 2 1

Portfolio PowerBasic

CONTENTS

TABLES

FIGURES

INTRODUCTION

PowerBASIC is a structured, high-level language which you can use to write useful and powerful programs for the Atari Portfolio computer. *Portfolio PowerBASIC* is a command-line compiler which compiles a PowerBASIC source program to machine code that is directly executable on the Portfolio machine.

The Compiler and the Run-Time Library

Portfolio PowerBASIC consists of two files: the *compiler* (called **PB.RUN**) and the *run-time library* (called **PBRUN.RUN**). These files work together to provide an environment in which PowerBASIC programs can be executed. The *compiler* creates a small executable program in machine code from your original source program. The *run-time library* contains "standard" procedures which are used by most programs (routines to read a line of text from a data file or format numeric information, for example).

In order to execute a compiled program, the *run-time library* must be present in memory. This can be accomplished in either of two ways:

- You can load it into memory once, before executing any compiled programs; it will then remain resident until the Portfolio is rebooted. In order to do this, the run-time library file must be a .COM file (**PBRUN.COM** for example).
- You can cause it to be loaded automatically, each time that a compiled program is executed; it will then be automatically unloaded after each program has finished executing. In order to do this, the run-time library must be a .RUN file called **PBRUN.RUN**.

.RUN files vs. .COM files

There are two different types of executable programs in the *Portfolio PowerBASIC* environment: *.RUN* files and *.COM* files. Both have their advantages and disadvantages; *Portfolio PowerBASIC* gives you the flexibility of using either or both types.

.RUN files

A *.RUN* file is an exact sequential byte-for-byte image of an executable program. Since a Portfolio memory

card is treated just like any other portion of the machine's main memory, rather than like a separate disk device, a program image which is stored on a card does not need to be loaded into a separate area of memory before it can begin running. It is always present in main memory while the card is plugged into the Portfolio, in a form which is ready to execute; it can simply "run in place."

The main advantage of a .RUN file is that no extra memory space, beyond that which is necessary to simply store the image (on a RAM card or ROM card), is required to execute the program. .COM files require extra memory space in which to load another copy of the program; this process also takes additional time. .RUN files do have some disadvantages, however:

- Since a .RUN file must contain a complete, sequential image of the code to be executed, it must not be fragmented in any way, *including* by DOS. Modifying or deleting *any* file on a memory card may cause *any or all* of the files on the card to become fragmented as DOS attempts to fill up gaps on the card which were created by the previously-used files. To ensure that no fragmented files exist on a memory card, you must first format the card before copying any files onto it. Simply deleting all of the files which are present on a card (by using **DEL *.***) is not enough; you must actually format the card using **FORMAT**. You may then copy any number of files onto the card and they will not become fragmented. If you ever

delete a file from the card, or modify a file on the card, however, it is most likely that some or all of the files on the card will become fragmented. If a .RUN file becomes fragmented, it will not execute properly, and you must start the whole process over again with a formatted card in order to properly restore the file.

- A .RUN file can only be executed if it is located on a memory (RAM or ROM) card; the Portfolio's internal drive can be used to store such files but not to execute them.
- The memory card upon which an executing .RUN file is located must remain physically plugged into the Portfolio until the program has completed execution. You can't remove the card in order to insert another containing a data file to be accessed by the program, for example; all data must be located on the memory card which contains the .RUN file or on the Portfolio's internal drive.

.COM files

A *.COM* file can contain information which describes different parts of a program, in addition to the program code itself. It is loaded into main memory and may need to be rearranged before it can execute. This requires additional memory space, since two copies of the program must exist at execution time: the original .COM image, and the new "executable" image (which is constructed from the original).

The restrictions which apply to .RUN files (listed above) do not apply to .COM files, however, since .COM files go through a separate loading process into main memory before execution begins. Therefore, the files may be fragmented and may be executed directly from the Portfolio's internal drive. Memory cards may also be swapped during program execution.

Changing executable file types

The *Portfolio PowerBASIC* compiler and run-time library are *distributed* as .RUN files, while the compiler *generates* .COM files. Due to the unique structure implemented in the *Portfolio PowerBASIC* system, you can change the types of these files by simply renaming them to use the appropriate file extension. For example, a program generated by the compiler called **MYPROG.COM** can be made to act like a .RUN file by using the DOS **REN** command to change its name to **MYPROG.RUN**. The compiler itself can be made to act like a .COM file by renaming it to **PB.COM**. Any .COM or .RUN file *which is part of the Portfolio PowerBASIC system* or *which was created by the PB compiler*, may be interchanged in this manner. Do *not* attempt this with the other .COM or .RUN files provided on the Portfolio, or with those created by other programs; only the *Portfolio PowerBASIC* system implements the structure required to interchange both file types.

A .COM file is executed directly at the DOS prompt simply by typing its name; a .RUN file must be exe-

cuted via the DOS **RUN** command. In order for the run-time library to be automatically loaded when a program begins execution, it must be named as a .RUN file: **PBRUN.RUN**. If it is named **PBRUN.COM**, you must load it yourself at the DOS prompt; it will then remain resident in memory until the Portfolio is rebooted.

Installing *Portfolio PowerBASIC*

The *Portfolio PowerBASIC* files are supplied on a ROM (read-only memory) card. Since the files are provided in .RUN format (**PB.RUN** and **PBRUN.RUN**), both the compiler and the run-time library may be executed in place on the card. Keep in mind, however, that you cannot remove the card while the compiler or run-time library is executing. With the ROM card plugged into the Portfolio (thus preventing the use of an additional RAM card), the amount of disk space which is available to store your compiled programs is limited by the capacity of the Portfolio's internal drive. If you have used the DOS **FDISK** command to allocate a significant amount of the internal drive to disk space rather than memory space, your programs may run out of memory quickly or may not be able to execute at all. Therefore we suggest copying **PB.RUN** and **PBRUN.RUN** from the ROM card to a RAM card before using them. By doing so, you will have plenty of disk space on the

RAM card to store your compiled programs, and you can allocate more of your internal drive to memory space for your programs to execute in.

Compiling and running a program

The steps to create and run a program are as follows:

- Use the Portfolio's built-in text editor to create a PowerBASIC source file, or download such a file from your IBM PC. This file may be named anything you wish (as long as it conforms to the DOS file naming conventions; see Chapter 1 of the *Atari Portfolio Owner's Manual*), though we suggest that you use the extension .BAS to easily distinguish PowerBASIC source files from files of other types.
- Start up the compiler. If it is named as a .RUN file (**PB.RUN**), you must execute it in place by typing **RUN PB**. If you have renamed it to **PB.COM**, you must execute it by typing **PB**. In either case, you can follow the name of the compiler with the name of the source file to be compiled, as in: **PB TESTPROG.BAS**. If you do not specify a file extension, as in: **PB TESTPROG**, .BAS will be assumed. To compile a file which really does not have an extension, such as MYPROG1, you must follow the name with a period on the command

line: **PB MYPROG1.**, in order to keep the compiler from appending .BAS.

- If you did not follow the compiler name with the name of the source file to be compiled, *Portfolio PowerBASIC* will assume that you wish to compile the file which was last edited using the Portfolio's text editor application.
- The compiler will now compile your program to a .COM file, displaying the name of the file and the amount of free memory left upon completion. If your program contains an error, an appropriate message is displayed and you are returned to the DOS prompt.
- If you wish your executable program to be a .RUN file rather than a .COM file, simply rename it with the extension .RUN.
- To execute your newly created program, the *Portfolio PowerBASIC* run-time library must be loaded in memory. If the run-time library file is called **PBRUN.RUN**, your program will load it automatically when it begins execution. If you have renamed it to **PBRUN.COM**, you must load it yourself by typing **PBRUN** now; it will remain resident in memory until you re-boot the Portfolio.
- You are now ready to begin execution of your program. If your program is a .COM file, simply type its name (with or without the .COM extension) at the DOS prompt. Otherwise, you must use the **RUN** command to execute it. If you did not previously install the run-time library, your program will detect this and attempt to install

it automatically before continuing. In order for automatic installation to take place, the run-time library must be named **PBRUN.RUN**, and must be present on a card which is plugged into the Portfolio. If the run-time library is not accessible, your program cannot execute.

Creating a program with the Portfolio's editor

The Portfolio contains a built-in Text Editor application which can be used to create *Portfolio PowerBASIC* source programs and data files. Most of the time, you will be performing three simple operations with the editor: load a file, edit the file, and save the file to disk.

To start up the editor, hold down the Atari (⅄) key and press *E*. Alternatively, you can just press *E* if the applications menu is displayed, or you can type **APP /E** at the DOS prompt. The editor now loads the file that you were last editing, or creates a new file called **UNNAMED.TXT**.

If you wish to load a different file, press the Atari key (⅄) or the *F1* key to bring up the main menu, select **Files**, then select **Load...** and type in the name of the file. Once the file has been loaded, you can use the arrow keys and other editor commands to move around in and change the text as you so desire.

When you are ready to save the file, simply press the *Esc* key. If you have made any changes to the file since you loaded it, the editor will now ask you if you wish to save the changes to disk. If you do, select **Yes**. Otherwise the changes will be lost. You are now ready to compile the file with the PB compiler (assuming that you have placed *Portfolio PowerBASIC* source code in the file).

If you make changes to a file and want to save the result under a different file name, press the Atari key (⅄) or the *F1* key (rather than *Esc*) to return to the main menu again. Select **Files**, then **Save as...**. The editor will now display the default file name and allow you to enter a new file name. If the new file already exists, the editor will ask you if you wish to overwrite the old file before doing so.

For more information on using the text editor, see Chapter 7 of the *Atari Portfolio Owner's Manual*.

Using the *Portfolio PowerBASIC* Help File

The Portfolio contains a built-in Address Book application which can be used in conjunction with the file **PB.ADR** to obtain information about *Portfolio PowerBASIC* statements and functions.

To start up the Address Book, hold down the Atari (⅄) key and press *A*. Alternatively, you can just press *A* if the applications menu is displayed, or you can type **APP /A** at the DOS prompt. The Address Book now loads the address file which you were last using, or creates a new file called **UNNAMED.ADR**. If you have never used the Address Book before, or if you were last using it with an address file other than **PB.ADR**, press the Atari key (⅄) or the *F1* key to bring up the main menu, select **Files**, then select **Load...** and type in **PB.ADR**.

Once the file has been loaded, a list of *Portfolio PowerBASIC* statements and functions will appear. Use the arrow keys to scroll the list until you find the item whose help information you wish to view, then press the *RTN* key when the cursor is located on the appropriate line. This will display more information about the selected statement or function. Many items have more than one screen of help information; use the arrow keys to scroll through the text line by line.

When you are ready to return to the main list, simply press the *Esc* key. You can now select another item or press *Esc* again to return to the DOS prompt. The next time that you start up the Address Book, the **PB.ADR** file will be loaded automatically; you will see the list of statements and functions immediately.

For more information on using the Address Book, see Chapter 4 of the *Atari Portfolio Owner's Manual*.

Distributing the Run-Time Library

As described in the *Portfolio PowerBASIC* License Agreement included with this product, the *Portfolio PowerBASIC* run-time library (**PBRUN.COM** and/or **PBRUN.RUN**) may be distributed under the terms and conditions listed in the agreement. These run-time modules are copyrighted and can only be distributed with your software product, in such a manner that they supplement the intrinsic value already existing in your software product.

Customer Support

Atari Corporation welcomes inquiries about your Atari computer products. We also provide technical assistance. Write to *Customer Relations* at the address below.

Atari user groups also provide outstanding assistance. To receive a list of Atari user groups in your area, send a self-addressed, stamped envelope to an address below.

In the United States, write to:

Atari Corporation
Customer Relations
P.O. Box 61657
Sunnyvale, CA 94088

In Canada, write to:

Atari (Canada) Corp.
90 Gough Road
Markham, Ontario
Canada L3R 5V5

In the United Kingdom, write to:

Atari Corp. (UK) Ltd.
P.O. Box 555
Slough
Berkshire SL2 5BZ

Please indicate *User Group List*, *Technical Assistance*, or the subject of your letter on the outside of the envelope.

Typefaces used in this manual

The different typefaces in this manual are used for the following purposes:

Italics indicate areas within commands that you need to fill in with your application-specific information; variable, function, and procedure names, for example, or variable values. For instance,

POKE *address, byte value*

means that you must supply values for *address* and *byte value* when you use the POKE statement in a program.

UPPERCASE indicates that part of a command that you must type in exactly as shown. For example,

RESUME 125

must appear in a program exactly as shown (although capitalization is not required).

SMALLCAPS: All *Portfolio PowerBASIC* reserved words (for example, END, RETURN, and $STACK) appear in this typestyle.

Brackets [ ] indicate that the information they enclose is optional. For example,

OPEN *filespec* AS [#]*filenum*

means that you can include a number sign (#) before the file number in an OPEN statement or leave it out, at your option. Therefore both of the following are legal *Portfolio PowerBASIC* statements:

```
OPEN "cust.dta" AS 1
OPEN "cust.dta" AS #1
```


Braces { } indicate a choice of two or more options, one of which must be used. The options are separated by vertical bars (|). For example,

GOTO {*label* | *line number*}

means that both GOTO followed by a label and GOTO followed by a line number are valid statements, and that GOTO by itself is not.

Ellipses ... indicate that part of a command can be repeated as many times as necessary. For example,

READ *variable* [,*variable*]...

means that multiple variables separated by commas can be processed by a single READ statement:

```
READ a$
READ a$, b$, a, b, c
```

Vertical Ellipses: three vertically spaced periods indicate the omission of one or more lines of program text that are not important for the particular example. For example,

```
FOR n = 1 TO 10
  .
  .
  .
NEXT n
```

These dots are also represented by **{statements}**.

Keycaps indicates a key on your keyboard. It is often used when describing a key you have to press to perform a particular function; for example, "Press *Esc* to exit from a menu."

CHAPTER 1

Programmer's Reference

This chapter contains an alphabetical listing of all of *Portfolio PowerBASIC*'s commands. Each entry goes into exact and specific detail about each command, and is cross-referenced to other relevant commands. *Portfolio PowerBASIC*'s arithmetic, logical, and boolean operators, as well as its data types, are also described.

Command categories

Portfolio PowerBASIC's commands can be grouped into four categories according to their syntactic class:

functions, statements, system variables, and meta-statements.

Functions (meaning *Portfolio PowerBASIC*'s predefined functions, as opposed to user-defined functions) return a numeric or string value. They must be used within expressions. Most functions require that one or more arguments be passed to them by the program; these arguments are either numeric, string, or combinations thereof, depending on the function. For example:

```
t = COS(3.1)            'numeric function
                        '(1 numeric argument)

t$ = LEFT$("Cat",2)     'string function (1 string,
                        '1 numeric argument)
```

Statements are the building blocks that make up programs. They instruct the computer to perform certain actions, such as open a file, display a string, or produce sound. Statements do not return a value, though many of them take one or more arguments. Each statement must appear on a line all by itself, or be separated from other program elements with delimiting colons or comments. For example:

```
'open a file, display some text, beep
'the speaker and assign the value 100
'to the variable Count
OPEN "DATAFILE.TXT" as #1
PRINT "This is a test"
BEEP : Count = 100
```

System variables allow your program to interact with your system ("system" means your computer, its

peripherals, and operating system). They are predefined by *Portfolio PowerBASIC*, and can be used to access and control certain information maintained by your system. For example:

```
a$ = DATE$              'read today's date
TIME$ = "10:15"         'set the current time
```

Metastatements are instructions to the *Portfolio PowerBASIC* compiler. Strictly speaking, metastatements are not part of the PowerBASIC language because they operate at compile time, rather than at run time (during program execution).

A metastatement is preceded by a dollar sign ($) to differentiate it from an ordinary statement. There can be only one metastatement per program line. For example:

```
$STACK 2000             'set the stack size
```

The reference directory format

Every *Portfolio PowerBASIC* command is listed alphabetically as a separate entry in this directory. Each entry contains a brief explanation of what the command does, a description of its syntax, clarifying remarks, and an example program. Where appropriate, related entries are cross-referenced and any restrictions on use are listed.

Explanations of the syntax description conventions used in this manual follow:

A *numeric expression* is a number, a numeric variable, or any *Portfolio PowerBASIC* function that evaluates to a number. Numeric expressions can include arithmetic, logical, and relational operators. Sometimes the type of a numeric expression is specified (*integer expression*, for example). Examples include:

Numeric constant:

16
0.035

Numeric variable:

Count%
Array1(I)

Numeric function:

SIN(3.14159)
LOG(x / (16 * x))

Numeric expression:

16 * x / y
5.6 ^ 1.2

A *string expression* is a string constant, a string variable, or any *Portfolio PowerBASIC* function that evaluates to a string. A string expression may optionally include the concatenation operator (the plus sign, +). Examples include:

String constant:

"Cat"

String variable:

Anthem$
Array1$(20)

Concatenation:

Anthem$ + "Cat"

String function:

LEFT$(Anthem$ + "Cat",4)
CHR$(45)

A *path* is a string expression describing a valid disk drive and/or a valid subdirectory. Examples include:

```
"A:"
"\POWERBAS"
"GAMES"
"C:\DATABASE\SALES"
```

A *filespec* (file specification) is a string expression describing a DOS file name, which consists of one to eight characters optionally followed by a period and a one- to three-character extension (letter case is ignored). A file specification can include a path. Except where noted, file names must be expressed as string variables or string constants. For example:

```
"MYFIRST.BAS"              'file name
"powerbas\myfirst.bas"     'directory + file name
"a:\powerbas\myfirst.bas"  'path + file name
```

A *label* identifies a line or set of lines in a *Portfolio PowerBASIC* program. It takes the form of either an alphanumeric word (which must end with a colon), or a line number. Both kinds of labels are more or less

interchangeable in *Portfolio PowerBASIC*, except that alphanumeric labels must appear on a line by themselves, immediately preceding the program line to which the label refers. For example:

```
10    X = Y + Z              ' line number 10
MYLOOP:
      X = Y + Z              ' label MYLOOP
```

Portfolio PowerBASIC Program elements

This section describes the basic elements of a *Portfolio PowerBASIC* program in greater detail.

Statements

Statements are the simplest building blocks that make up programs. A line of *Portfolio PowerBASIC* code can contain none, one, or several statements, each separated by a colon. The simplest program in *Portfolio PowerBASIC* is a one-line statement, such as:

```
PRINT "Hello world"
```

Usually, you'll want to write longer programs. You could combine several statements on one line, separating the statements by colons (:). It's better to use

a separate line for each statement. You can add comments to the end of a line and use line numbers at the start. Briefly, then, *Portfolio PowerBASIC* programs consist of one or more lines of source text, each of which follows one of these formats:

```
[line number] [statement] [:statement]... [' comment ]
```

or

```
label:
```

or

```
$metastatement
```

If you write lines wider than the Portfolio's screen width, you'll have a hard time visualizing your program. In situations where syntax requirements force you to build a long line (the FIELD statement is notorious for this), put an underscore (_) at the end of the line. This causes *Portfolio PowerBASIC* to regard the next line as an extension of the first. You can use the underscore to continue statements for many lines. For example:

```
PRINT "This is a very long " _
      "line that goes on " _
      "and on and on and " _
      "on..."
```

As far as the compiler is concerned, this is one long line starting with **PRINT** and ending with *"on..."* without any underscore characters.

Comments

Comments can be any text added to the end of a line and separated from the program itself by a single quote (’). The single quote can be used in place of :REM to separate comments from the statements on that line—unless it is at the end of a DATA statement, in which case DATA will think the quote is part of the last item in the statement. To comment a DATA statement, you need to use a colon and a single quote. Commenting DATA statements is definitely recommended. Do you really think you can remember what all those numbers stand for?

```
DATA 130, 140, 150    :' x, y, z coordinates, ship 1
DATA 135, 156, 157    :' x, y, z coordinates, ship 2
```

For other uses, it isn't necessary to separate single quoted comments from adjoining statements with a colon. The following are equal in the eyes of the compiler:

```
area = radius^2 * 3.14159   ' calculate the area
area = radius^2 * 3.14159   : REM calculate the area
```

Line numbers

These are integers in the range 1 to 32767, which serve to identify program lines. *Portfolio PowerBASIC* takes a relaxed stance toward line numbers. They can be freely interspersed with labels and used in some parts of a

program and not in others. In fact, they need not follow in numeric sequence. No two lines can have the same number, and no line can have both a label and a number. Line numbers are essentially labels. If a program causes a run-time error, the line number of the most recently executed line will be displayed as part of the error message. Each line number in your program also consumes an extra six bytes of memory space.

Line numbers are really just a concession to compatibility with older versions of BASIC. Line numbering can lead to bad programming style. Since the numbers themselves can now be in any order, they give a false sense of structure to a program. We recommend that you avoid line numbers and use labels instead, where necessary.

Labels

Using labels instead of line numbers allows you to make your program's flow much more readable. For example

```
GOSUB BuildQuarks
```

tells you much more than

```
GOSUB 1723
```

Each label must appear on a line by itself (though a comment may follow) and serves to identify the statement immediately following it. Labels must begin

with a letter and can contain any number of letters and digits. Case is insignificant—*THISLABEL*, **thislabel**, and *ThisLabel* are all equivalent. A label must be followed by a colon. However, statements that refer to the label (for example, GOSUB) must not include the colon.

Proper use:

```
PRINT "Now Sorting Invoices"
GOSUB SortInvoices          'This calls a label
PRINT "All Done!"
END

SortInvoices:               ' This is a legal label
  { Sorting code goes here. }
RETURN
```

Illegal use:

```
ExitPoint: a = a + 1    ' labels must stand alone
```

Although the following would be acceptable:

```
1010 a = a + 1     ' line numbers are OK on same line
```

Note that unlike a line number, a label name will not be displayed as part of a run-time error message.

Metastatements

Metastatements operate at a different level than standard statements. They are actually commands to the compiler, rather than part of the program. Also

known as *compiler directives*, they always begin with a dollar sign ($). Standard statements control the computer at run time; metastatements control the compiler at compile time.

The following are all valid lines in *Portfolio PowerBASIC*:

```
Start:                          'label only
10                              'line number only
$STACK 2000                     'metastatement
20 a=a+1                        'line # + stmt
a=a+1 : b=b+1                   'two stmts
30 a=a+1 : b=b+1 : c=a+b  'line # + 3 stmts
```

While the last line is perfectly valid, it might be more readable if recast as:

```
addstuff:
  a = a + 1
  b = b + 1
  c = a + b
```

Character set

Portfolio PowerBASIC provides you with a set of fundamental language elements (characters, symbols, and reserved words) that can be put together in infinite ways to build any software machine conceivable. The characters and symbols are called the *character set*. Table 1.1 shows the specific meanings of the *Portfolio PowerBASIC* character set.

Characters and symbols

The letters *A* to *Z* or *a* to *z* and the numbers 0 to 9 can be used in forming the *identifiers*, or names, of labels, variables, procedures, and functions.

The numbers 0 to 9; the symbols ., +, and –; and the letters *E*, *e*, *D*, and *d* can all be used in forming numeric constants.

Naming variables

Strings, numeric variables, and arrays can share the same name. For example, *fog!*, *fog$*, and *fog%(5)* are separate variables. One is a single-precision floating point variable, one is a string, and the other is an element of an integer array. Functions, procedures, subroutines, and other labels cannot share names with each other, nor can they share names with variables.

Rather than trying to remember the rules of what can share names and what can't, you'd be better off using unique names for everything. This also makes your code more understandable when you go back to look at it later.

Table 1.1: Special symbols

Symbol	Description	Function
=	Equal sign	Assignment/relational operator
+	Plus sign	Addition/string concatenation operator
–	Minus sign	Subtraction/negation operator
*	Asterisk	Multiplication operator
/	Slash	Division operator
\	Backslash	Integer division operator
^	Caret	Exponentiation operator
%	Percent sign	Integer type declaration
!	Exclamation point	Single-precision type declaration
#	Number sign	Double-precision type declaration and file number indicator
$	Dollar sign	String type declaration, metastatement prefix
( )	Parentheses	Function/procedure arguments, arrays, expression prioritizing
[ ]	Brackets	Valid only for arrays
	Space	Separator
,	Comma	All-purpose delimiter
.	Period	Decimal point, file-name extension separator
'	Single quote	Comment delimiter
;	Semicolon	All-purpose delimiter
:	Colon	Statement delimiter; label terminator
?	Question mark	PRINT substitute (not good style)
<	Less than	Relational operator
>	Greater than	Relational operator
"	Double quote	String delimiter
_	Underscore	Line continuation character

Reserved words

Portfolio PowerBASIC reserves the use of certain words for predefined syntactic purposes. These reserved words cannot be used as labels, variables, named constants, or procedure or function names, although your identifiers can contain them. Most words which are part of the *Portfolio PowerBASIC* language are reserved words.

For example, *END* is an invalid variable name because it conflicts with the reserved word END. However, *ENDHERE* and *FRIEND* are acceptable; reserved words can be a *part* of a variable name.

There is one exception to this rule: You can't start an identifier with the letters *FN* because it conflicts with the syntax for user-defined DEF FN functions.

Attempting to use a reserved word as an identifier produces a compile-time syntax error.

Portfolio PowerBASIC Data Types

Portfolio PowerBASIC supports three unique numeric types and a string type:

- integer

- single-precision floating point
- double-precision floating point
- string

Tables 1.2 and 1.3 summarize the most important features and distinctions of these data types.

Table 1.2: Summary of data types

Variable Type	Indicator	Element Size (in bytes)	DEF*type**
Integers			
Integer	%	2	DEFINT
Floating point			
Single precision	!	4	DEFSNG
Double precision	#	8	DEFDBL
Strings			
String	$	2	DEFSTR

* DEF*type* refers to all four variable type declaration statements.

Table 1.3: Numeric data types—sizes and ranges

Data Type	Size	Range
Integer	16 bits (2 bytes)	–32,768 to 32,767
Single precision	32 bits (4 bytes)	$\pm 8.43\times10^{-37}$ to $\pm 3.37\times10^{38}$
Double precision	64 bits (8 bytes)	$\pm 4.19\times10^{-307}$ to $\pm 1.67\times10^{308}$

Integers (%)

The simplest and fastest numbers rattling around inside *Portfolio PowerBASIC* programs are *integers*. To *Portfolio PowerBASIC*, an integer is *a number with no decimal point* (what mathematicians call whole numbers) within the range –32,768 to +32,767. These values stem from the underlying 16-bit representation of integers: 32,768 is 2^{15}.

Integers are identified by following the variable name with a percent sign (for example, *var%*), or by using the DEFINT statement. For example, if you use this declaration in your program code:

DEFINT I, J, K

all variables following this declaration that start with the letter I, J, or K will be integers by default.

Although this range limits the usefulness of integers, for many applications they will suffice. In all programs, there will be at least a few variables (such as the counters in FOR/NEXT loops) that can function within these constraints. Using integers produces extremely fast and compact code. Your computer is uniquely comfortable performing integer arithmetic (it does it fast), and each number requires only two bytes of memory.

Single-Precision floating point (!)

Single-precision floating point numbers (or simply *single precision*) may be the most versatile numeric type within *Portfolio PowerBASIC*. Single-precision values can contain decimal points and have a phenomenal range: $\pm 8.43 \times 10^{-37}$ to $\pm 3.37 \times 10^{38}$.

Single-precision variables are identified by following the variable name with an exclamation point (*var!*) or nothing (if you don't use a type indicator, *Portfolio PowerBASIC* assumes you want a single-precision value), or by using the DEFSNG statement as described in the previous discussion of integers.

What happens if your finger slips, and you have the variable *count!* in one place in your program, and *count* with no type indicator in another? Unless you have used a DEF*type* statement to declare that all variables starting with the letter "c" are by default of another type, the compiler will assume that these two variables are one and the same.

You would be hard-pressed to dig up a quantity that wouldn't fit into a single-precision number. Calculation speed isn't bad, although not as quick as that for integers, and four bytes are required for each number.

Such numbers are wonderfully useful. However, while single precision can represent both enormous and microscopic numbers, it cannot remember numbers beyond six-digit accuracy. In other words, single precision does a good job with figures like $451.21 and

$6,411.92, but $671,421.22 can't be represented exactly because it contains too many digits. Neither can 234.56789 or 0.00123456789. A single-precision representation will come as close as it can in six digits: $671,421, or 234.568, or 0.00123457.

Double-Precision floating point (#)

Double-precision floating-point numbers take twice as much space in memory as single-precision numbers (8 bytes versus 4 bytes), and consequently take longer to calculate, but they offer greater range ($\pm 4.19 \times 10^{-307}$ to $\pm 1.67 \times 10^{308}$) and accuracy (16 digits versus the 6). The storage requirement of double-precision numbers becomes especially significant when dealing with arrays. A double-precision, 5,000-element array requires 40,000 bytes. An integer array with the same number of elements occupies only 10,000 bytes.

Double-precision variables are identified by following the variable name with a pound sign (*var#*), or by using the DEFDBL statement as described in the previous discussion of integers.

Strings ($)

A *character* can be any ASCII value from 0 to 255, including the familiar members of the alphabet, both

uppercase and lowercase. When characters are strung together, they are called, sensibly enough, *strings*. Characters and strings are used in many ways in programming.

String variables contain character data of arbitrary length. Each string variable consumes two bytes which are used internally to locate information about the string.

Your programs may use approximately 64K of memory as string space. Each string may be at most 32,750 bytes (characters) long.

Constants

Portfolio PowerBASIC programs process two distinct classes of data: *constants* and *variables*. A variable is allowed to change its value as a program runs. A constant's value is fixed at compile time and cannot change during program execution (hence, it remains constant). *Portfolio PowerBASIC* supports two types of constants: string constants and numeric constants.

String constants

String constants are simply groups of characters surrounded by double quotes. For example,

"This is a string"

"3.14159"
"Kurt Inman, Engineer at Large"

If a string constant is the last thing on a line, the closing quotes are optional:

```
PRINT "This is sloppy but legal.
```

Numeric constants

Numeric constants represent numeric values. They consist primarily of the digits 0 through 9 and a decimal point. Negative constants need a leading minus sign (-); a plus sign (+) is optional for positive constants. The amount of precision you supply determines the internal representation (integer, single precision, or double precision) which *Portfolio PowerBASIC* will use in processing that constant.

If a numeric constant is not followed by a type specifier, the following rules are used to determine which precision to store the value in:

1. If the value contains no decimal point and is in the range –32,768 to 32,767, *Portfolio PowerBASIC* stores the value as an integer.
2. If the value contains a decimal point (or is an integer outside the range for integer constants) and has up to six digits, *Portfolio PowerBASIC* stores it as a single-precision floating point.

3. A numeric constant with a decimal point and more than six digits, or a whole number too large to be an integer or single-precision number but small enough to fall within the range of double-precision floating point, is stored in double-precision floating-point format. For example:

345.1	Single precision
1.10321	Single precision
1.103213	Double precision
3453212.1234	Double precision

Variables

Variables represent numeric or string values. Unlike constants, the value of a variable can change during program execution. Like labels, variable names must begin with a letter and can contain up to 255 letters and digits (although in practical terms you really can't exceed the length of a line). Be generous in naming important variables.

Portfolio PowerBASIC supports four variable types: string, integer, and single- and double-precision floating point. Usually, variable typing is accomplished by appending a type declaration character to the variable name. Table 1.2 on page 31 summarizes the distinguishing characteristics of variable types.

If you don't include a type declaration character with a variable, *Portfolio PowerBASIC* uses its default type,

single precision. To make another type the default, use the DEF*type* statement. For example,

```
cat# = 1.312      ' cat# is a double-precision variable
cat = 16.5        ' cat is single precision by default
cat% = 3          ' cat% is an integer variable
DEFINT c          ' variables beginning with c are now
                  '  integer by default
cats = 16         ' cats is an integer variable by default
```

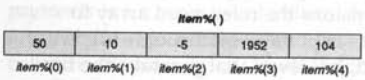
Note that *cat#*, *cat*, and *cat%* are all separate variables.

A common practice to save space and make calculations faster is to declare **DEFINT a-z** at the start of all programs. Then *all* variables will be integer unless declared otherwise. If you do this consistently, you won't need the % after each integer variable, and you won't get confused.

Arrays

It's often useful to treat a set of variables as a group. This lets you perform repetitive operations more easily. An *array* is a group of string or numeric data sharing the same variable name. The individual values that make up an array are called *elements*. An element of an array can be used in a statement or expression wherever you would use a regular string or numeric variable. In other words, each element of an array is itself a variable.

You can think of an array as a row of boxes numbered from 0 to a predetermined number; 4, in the example in Figure 1.1. Each box holds a distinct value, which may or may not differ from the values in the other boxes. The boxes and their numbers are represented by parentheses surrounding a number; for example, *item%(3)* represents box number three of the array *item%*. Thus, if the *value* held within box number 3 is 1952, the code **total% = item%(3)** would place the value 1952 into *total%*.



At program startup time, each element of each numeric array is set to zero; string arrays are set to the null string ("", length zero). This process is called *initializing* (or *clearing*) an array.

Declaring the name and type of an array, as well as the number and organization of its elements, is performed by the DIM statement. For example:

```
DIM payments(55)
```

creates an array variable *payments*, consisting of 56 single-precision elements, numbered 0 through 55. Array *payments* and a single-precision variable also named *payments* are separate variables. If this is confusing, you'll understand why we suggested earlier that you use different variable names.

Subscripts

Individual array elements are selected with subscripts, which are integer expressions within parentheses to the right of an array variable's name. For example, *payments*(3) and *payments*(44) are two of *payment*'s 56 elements. The first element of an array has a subscript value of zero.

Whenever *Portfolio PowerBASIC* finds a reference to an array which has no corresponding DIM statement, it automatically DIMensions the referenced array to eleven elements (subscript values 0 through 10). We strongly recommend, however, that you take the time to explicitly dimension every array your program uses.

String arrays

The elements of string arrays hold strings instead of numbers. Each string can be a different length, from 0 to the maximum string size (32750 characters). The total string space available for string data and string array data is approximately 64K. For example,

```
DIM words$(50)
```

creates a sequence of 51 independent string variables:

```
words$(0) = "Pat likes cats."      '15-char string
words$(1) = ""                     'a null string
```



```
words$(2) = "Eric is a sweet child." '22-char string
.
.
.
words$(50) = SPACE$(200)   '200-char string
```

Multidimensional arrays

Arrays can have one or more dimensions, up to a maximum of eight. A one-dimensional array such as *payments* is a simple list of values. A two-dimensional array represents a table of numbers with rows and columns of information. Multidimensional arrays are equally possible:

```
DIM one(15)          ' one-dimensional list
DIM two(15,20)       ' two-dimensional table
DIM three(7,9,1)     ' 8x10 game board with room
                     ' in the third dimension for 2
                     ' items: piece type and owner
```

The maximum number of elements per dimension is 32767.

Array storage requirements

For technical reasons relating to execution speed and code size efficiency, *Portfolio PowerBASIC* limits the amount of memory for variables and arrays to 64K. This is in addition to the 64K which is available only for

string data. Each element of a string array contains a string descriptor, which takes up two bytes of the 64K variable/array space. In addition, the actual string data stored in each element takes up space in the 64K string data area. The maximum number of elements an array may contain is a function of its type (see Table 1.2 on page 31).

Expressions

An expression consists of operators and operands. *Operators* are symbols or words, such as the plus sign (+), that represent mathematical, relational, or logical (Boolean) operations. *Operands* are the quantities on which operations are performed. Like the data types, there are two fundamental types of expressions in *Portfolio PowerBASIC*: string and numeric.

A *string expression* consists of string constants, string variables, and string functions, optionally combined with the concatenation operator (+). String expressions always produce strings as their result. Examples of string expressions include

```
"Cats and dogs"                 ' string constant
firstname$                      ' string variable
firstname$ + lastname$          ' concatenation
LEFT$(a$ + z$,7)                ' string functions
a$ + MID$("Cats and dogs",5,3)
RIGHT$(MID$(a$ + z$,1,6),3)
```


Numeric expressions, not surprisingly, are formed from numeric constants, variables, and functions, optionally separated by numeric operators. Numeric expressions produce a value in one of the three numeric types (integer, single precision, or double precision). Examples of numeric expressions include:

```
37
37/15
a
37/a
SQR (37/a)
SQR ((c + d)/a) * SIN (37/a)
```

NOTE: If *Portfolio PowerBASIC* encounters an expression in which all operands are integers, it will expect both the intermediate and final results of the expression evaluation to be within the integer range. It will *not*, however, check to see whether the calculation actually exceeded this range. To force an expression (which would otherwise overflow in this manner) to be evaluated using floating-point arithmetic, you must include a floating-point operand:

```
a% = 32768              'causes an Overflow error at
                        ' run-time

a% = 32767 + 1          'no error since 32767 and 1 are
                        ' both integers

a% = 32767 + 1.0        'causes error since 32767 + 1.0
                        ' is 32768.0

a% = 2
b% = 20000
```

```
c% = a% * b%        'no error since a% and b% are
                    ' both integers
```

In forming numeric expressions, you should be aware that certain operations will be performed before others according to a hierarchy. The following is a list of the order of expression evaluation. Exponentiation has the highest priority, meaning it will be performed first; XOR has the lowest, meaning it will be performed last.

- exponentiation (^)
- negation (-)
- multiplication (*), floating-point division (/)
- integer division (\)
- modulo (MOD)
- addition (+), subtraction (-)
- relational operators (<, <=, =, >=, >, <>)
- NOT
- AND
- OR, XOR (exclusive OR)

For example, the expression **3 + 6 / 3** evaluates to 5, not 3. Division has a higher priority than addition, so the division operation **(6 / 3)** is performed first. Even though the compiler won't get confused, people could, so better programming style might be to use **3 + (6 / 3)** or **3 + 6/3**, using either parentheses or spacing to make the intent clear. Otherwise, it's easy to misread the statement as **(3 + 6) / 3**.

To handle operations of the same priority, *Portfolio PowerBASIC* proceeds from left to right. For example, in the expression **4 - 3 + 6**, the subtraction **(4 - 3)** is performed before the addition **(3 + 6)**, producing the intermediate expression 1 + 6.

Operations inside parentheses are of highest priority and are always performed first; within parentheses, standard precedence is used. Use parentheses like garlic—generously, but not to excess.

Operators

The numeric operators are classified into three major groups: arithmetic, relational, and logical.

Arithmetic operators

Arithmetic operators perform normal mathematical operations. Table 1.4 lists the *Portfolio PowerBASIC* arithmetic operators in precedence order.

Table 1.4: Arithmetic operators

Operator	Action	Example
^	Exponentiation	10^4
–	Negation	–16
*, /	Multiplication, floating-point division	45 * 19, 45 / 19
\	Integer division	45 \ 19
MOD	Modulo	45 MOD 19
+, –	Add, subtract	45 + 19, 45 – 19

Several of these operators merit a word of explanation. The backslash (\) represents integer division. Integer division rounds its operands to integers to produce a truncated quotient with no remainder. For example, **5 \ 2** evaluates to 2, and **9 \ 10** evaluates to 0.

The remainder of an integer division can be determined with the MOD (modulo) operator (MOD is valid for all numeric types). MOD is similar to integer division except that it returns the *remainder* of the division rather than the quotient. For example, **5 MOD 2** returns the value 1, and **9 MOD 10** returns the value 9.

Relational operators

Relational operators allow you to compare the values of two strings or two numbers (but not one of each) to obtain a Boolean result of TRUE or FALSE. The result of the comparison is assigned an integer value of –1 if

the relation is TRUE, and 0 if the relation is FALSE. For example:

```
PRINT 5 > 6, 5 < 6, (5 < 6) * 15
```

prints out 0, –1, and –15. Although they can be used in any numeric expression (for example, **a = (b > c) / 13**), the numeric results returned by relational operators are generally used in an IF or other decision statement to make a judgment regarding program flow. Table 1.5 lists the relational operators.

Table 1.5: Relational operators

Operator	Relation	Example
=	Equality	5 = 5
<>	Inequality	5 <> 6
<	Less than	5 < 6
>	Greater than	6 > 5
<=, =<	Less than or equal to	5 <= 6
>=, =>	Greater than or equal to	6 >= 5

When arithmetic and relational operators are combined in an expression, arithmetic operations are always evaluated first. For example, **4 + 5 < 4 * 3** evaluates to –1 (TRUE) because the arithmetic operations (addition and multiplication) are carried out before the relational operation. This then tests the truth of the assertion **9 < 12**.

Logical operators

Logical operators perform logical (Boolean) operations on numbers of any type. Before a Boolean operation takes place, *Portfolio PowerBASIC* automatically converts floating-point numbers to integers. Used with relational operators, logical operators allow you to set up complex tests like

IF day > 29 AND month = 2 THEN PRINT "Huh?"

AND

This statement performs a logical AND on the integer results returned by the two relational operators. The AND operator returns TRUE if both its operands are TRUE. The AND operator has a lower priority than the > and = relational operators, so parentheses aren't needed. For example, if the value of *day* is 30 and *month* is set to 2, both relational operators return TRUE (-1). The AND operator then performs a logical AND on these two TRUE results producing a final TRUE result.

OR

The OR (sometimes called inclusive or) operation returns TRUE if one or both of its arguments are TRUE, and returns FALSE only if both arguments are FALSE. For example,

-1 OR 0 is TRUE
0 OR 0 is FALSE
5 > 6 OR 6 < 4 is FALSE

XOR

The XOR (exclusive or) operation returns TRUE if the values being tested are different, and returns FALSE if they are the same. For example,

-1 XOR 0 is TRUE
-1 XOR -1 is FALSE
5 > 6 XOR 6 < 4 is TRUE

Table 1.6: Truth Tables for Logical Operations

x	*y*	*x AND y*	*x OR y*	*x XOR y*
T	T	T	T	F
T	F	F	T	T
F	T	F	T	T
F	F	F	F	F

Bit manipulations

In addition to creating complex tests, logical operators permit control over the underlying bit patterns of their integer operands. The most common operations are AND, OR, and XOR (exclusive OR) masking.

AND masks clear selected bits of an integer quantity without affecting the other bits of that quantity. For example, to clear the most-significant (leftmost) 2 bits in the integer value 9700 Hex, use AND with a mask of 3FFF Hex; that is, the mask contains all 1s, except for the bit positions you wish to force to 0:

```
          1001 0111 0000 0000  9700 Hex
AND       0011 1111 1111 1111  3FFF Hex  (the mask)
          0001 0111 0000 0000  1700 Hex  (the result)
```

An OR mask sets selected bits of an integer without affecting the other bits. To set the most-significant 2 bits in 9700 Hex, use OR with a mask of C000 Hex; that is, the mask contains all 0s, except for the bit positions you wish to force to 1:

```
          1001 0111 0000 0000  9700 Hex
OR        1100 0000 0000 0000  C000 Hex  (the mask)
          1101 0111 0000 0000  D700 Hex  (the result)
```

An XOR mask complements (reverses) selected bits of an integer quantity without affecting the other bits of that quantity. For example, to complement the most-significant 2 bits in 9700 Hex, use XOR with a mask of C000 Hex; that is, all zeros, except for the positions to be complemented:

```
          1001 0111 0000 0000  9700 Hex
XOR       1100 0000 0000 0000  C000 Hex  (the mask)
          0101 0111 0000 0000  5700 Hex  (the result)
```

Strings and relational operators

Portfolio PowerBASIC lets you compare string data. String expressions can be tested for equality, as well as for "greater than" and "less than" alphabetic ordering.

Two string expressions are equal if and only if they contain exactly the same characters in exactly the same order. For example,

```
a$ = "CAT"
PRINT a$ = "CAT", a$ = "CATS", a$ = "cat"

Output:  -1      0      0
(true, false, false)
```

String ordering is based on two criteria: first, the ASCII values of the characters they contain, and second, the length of the strings.

For example, the letter *A* is less than the letter *B* because the ASCII code for *A*, 65, is less than the code for *B*, 66. Note, however, that *B* is less than *a* because the ASCII code for each lowercase letter is *greater* than the corresponding uppercase character (exactly 32 greater). When comparing mixed uppercase and lowercase information, use the UCASE$ or LCASE$ functions to keep case differences from interfering with the test.

```
city1$ = "Seattle"
city2$ = "Tucson"
IF UCASE$(city1$) > UCASE$(city2$) THEN
  PRINT city1$
ELSE
  PRINT city2$
END IF

city1$ = UCASE$(city1$)
city2$ = UCASE$(city2$)
IF city1$ > city2$ THEN
  PRINT city1$
```

```
ELSE
  PRINT city2$
END IF
```

Note the difference between the two sets of statements. In the first case, the string variables *city1$* and *city2$* are converted to uppercase for the comparison only, so the first IF/THEN prints *Tucson*. In the second case, the conversion is performed on the variables themselves, so the result will be *TUCSON*.

Length is important only if both strings are identical up to the length of the shorter string, in which case the shorter one evaluates as less than the longer one; for example, *CAT* is less than *CATS*.

Signed and unsigned integer representation

Four *Portfolio PowerBASIC* functions, STRPTR, STRSEG, VARPTR, and VARSEG, return floating point values instead of integers as you might expect. The values returned by these functions are always positive integers in the range 0..65535 (0000..FFFF hex), but they are stored as floating point numbers since *Portfolio PowerBASIC*'s integer variables can only contain numbers in the range –32768..+32767 (which also represents 0000..FFFF hex). Normally, you will be using these functions in conjunction with BIN$, DEF SEG, HEX$,

INP, OCT$, OUT, PEEK, POKE, and REG, all of which can handle either representation, accepting floating point values in the range –32768..+65535 and interpreting –32768..-1 as +32768..+65535. This is compatible with *PowerBASIC* for the IBM PC.

There may be times, however, when you wish to store a value returned by one of these functions in an integer variable. Using the following DEF FN, you can create a signed integer value in the range –32768..+32767 from an unsigned value of 0..65535:

```
DEF FNSigned%(Unsigned!)
  IF Unsigned! > 32767 THEN
    FNSigned% = Unsigned! - 65536
  ELSE
    FNSigned% = Unsigned!
  END IF
END DEF
```

The resulting signed integer representation can be stored in an integer variable and manipulated with logical operators like AND and OR.

$COM metastatement

$COM allocates space for the serial communications port receive buffer.

$COM *size*

This metastatement sets the size of the serial communications buffer. *size* is an integer constant defining the buffer capacity in bytes (0 to 32767). *size* must be in increments of 16 bytes. If a $COM metastatement is not present in your program, the buffer is allocated to the default size of 256 bytes. The $COM metastatement should only be used once in any program, before any executable source code.

See also: OPEN COM

$STACK metastatement

$STACK declares the size of the run-time stack.

$STACK *size*

size is a numeric constant from 512 to 32766 (bytes). $STACK determines how much run-time memory will be devoted to the stack. The stack is used for return addresses, parameter passing, local variables during procedure and function calls, and within structured statements (FOR/NEXT, WHILE/WEND, etc.).

The default stack size is 1024 bytes. You may want to allocate more stack space if your program is heavily nested, uses many local variables, or performs recursion. The FRE function allows you to figure out how much stack space your program needs; it returns the smallest amount of stack space which was *ever* available during the execution of your program. The $STACK metastatement should only be used once in any program, before any executable source code.

See also: FRE

ABS function

ABS returns the absolute value of its argument.

***y* = ABS(*numeric expression*)**

The absolute value of a number is its positive value. For example, the absolute value of –3 is 3, and the absolute value of +3 is also 3.

ASC function

ASC returns the ASCII code of the first character in its argument.

***y* = ASC(*string expression*)**

ASC returns the ASCII code (0 to 255) of the first character of *string expression*. The acronym ASCII stands for American Standard Code for Information Interchange, a standardized code in which the numbers between 0 and 255 are used to represent the upper- and lowercase letters, the numerals 0 to 9, punctuation marks, and other symbols used in data communication. CHR$ does the reverse of ASC (it produces the one-character string corresponding to its ASCII code argument).

Restrictions: The string expression passed to ASC may not be a null (empty) string. If it is, run-time error 5 ("Illegal function call") is generated.

See also: CHR$

ATN function

ATN returns the trigonometric arctangent of its argument.

***y* = ATN(*numeric expression*)**

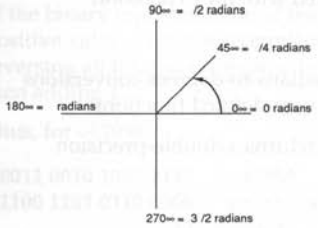
ATN returns the arctangent (inverse tangent) of *numeric expression*; that is, the angle whose tangent is *numeric expression*.

The result, as with all operations involving angles in *Portfolio PowerBASIC*, is in radians rather than degrees. Although most people are accustomed to measuring angles in degrees, the radian is a more convenient measurement for mathematical and trigonometric operations. One radian is defined as the angle at the center of a circle that subtends an arc equal in length to one radius. Since for all circles, using the constant π,

circumference/radius = 2 * π

the length of the circumference of a circle is equal to 2 * π * radius, and the angle of a full circle (360 degrees) is equal to 2 * π radians. If you place 0 radians on the positive x-axis and measure counterclockwise, you have:

Figure 1.2



If you're more comfortable with degrees, radians can be converted to degrees by multiplying the radian value by 57.2958. For example, the arctangent of 0.23456 can be converted this way:

```
t = ATN(.23456)   't = 0.230395... (radians)
t = 57.2958 * ATN(.23456)   't = 13.200... (degrees)
```

To convert degrees to radians, multiply by 0.01745333. For example,

```
14 degrees = (0.01745333 * 14) radians
           = 0.24435 radians
```

Rather than memorizing the radians/degrees conversion factors, calculate them for yourself by remembering this relationship: 2π radians equals a full circle (360 degrees), so 1 radian is $180/\pi$ degrees. Conversely, 1 degree equals $\pi/180$ radians.

π is a transcendental constant, meaning that it has an infinite number of decimal places. To 15-place accuracy, adequate for most applications, $\pi = 3.141592653589793$. This value can be calculated with the expression

```
pi# = 4 * ATN(1)
```

Degrees-to-radians and radians-to-degrees conversions are good applications for user-defined functions.

The ATN function always returns a double-precision result.

See also: COS, SIN, TAN

BEEP statement

BEEP makes the speaker sound.

BEEP

BEEP plays a single quarter-second, 800-Hz tone through the computer's built-in speaker.

See also: TONE

BIN$ function

BIN$ returns a string that is the binary (base 2) representation of its argument.

***s$* = BIN$(*numeric expression*)**

numeric expression must be in the range –32768 to 65535; if it is outside this range, error 6 ("Overflow") is generated. Any fractional part of *numeric expression* is rounded before the string is created. If *numeric expression* is negative, BIN$ returns the two's complement of the binary representation of the corresponding positive value. The two's complement is formed by first reversing all bits (all 0s become 1s and vice versa) and then adding 1.

Thus, for –12959:

```
0011 0010 1001 1111   ' +12959
1100 1101 0110 0000   ' reverse all bits
               +1     ' add 1
1100 1101 0110 0001   ' -12959
```

Because of the use of two's complement binary form to represent negative numbers, some bit patterns (those with bit 16 equal to 1) can represent two different values depending on whether they are interpreted as a two's complement or straight binary. Specifically,

```
BIN$(-32768) = BIN$(32768) = 1000000000000000
BIN$(-32767) = BIN$(32769) = 1000000000000001
.
.
.
BIN$(-2)     = BIN$(65534) = 1111111111111110
BIN$(-1)     = BIN$(65535) = 1111111111111111
```

See page 52 for more about converting between these representations.

See also: HEX$, OCT$

CALL statement

CALL invokes a procedure.

CALL *procname* [(*parameter list*)]

procname is the name of a procedure defined elsewhere in the program with SUB/END SUB. *parameter list* is an optional, comma-delimited list of variables to be passed to *procname*. When the code in a procedure has finished executing, control passes to the statement immediately following the CALL.

The number and type of arguments passed must agree with the parameter list in *procname*'s definition; otherwise, a compile-time "Parameter mismatch" error occurs. Procedure arguments must only consist of scalar (non-array) variables; neither constants, expressions, individual array elements, nor whole

arrays, may be passed as arguments. Each argument is passed by reference, meaning that the procedure is passed the address of the parameter and the parameter can be modified by the procedure.

See also: SUB

CALL INTERRUPT statement

CALL INTERRUPT invokes a system interrupt.

CALL INTERRUPT *n*

n is an integer expression specifying the interrupt to be triggered, from 0 to 255. Just before the interrupt handler receives control, the processor's registers are loaded with the data in the register buffer (set using the REG statement). When the interrupt handler terminates and control returns to the program, the register buffer (which can be read using the REG function) is loaded with the data in the processor's registers. At any time, this buffer contains the data which was in the processor's registers at the completion of the most recent CALL INTERRUPT statement.

The *Atari Portfolio Technical Reference Manual* contains complete information on the many functions available through the CALL INTERRUPT mechanism.

Restrictions: Interrupt 97 (61 hex) is used internally by the Portfolio. Interrupts 0, 4, 36 (24 hex), 98 (62 hex), and 99 (63 hex) are used internally by the *Portfolio*

PowerBASIC system. In addition, if your program performs serial communications, interrupt 12 (0C hex) is used internally by *Portfolio PowerBASIC*. Do *not* attempt to access these interrupts with the CALL INTERRUPT statement.

See also: REG

CHDIR statement

CHDIR changes the current default directory (as does the DOS **CHDIR** command).

CHDIR *path*

path is a string expression conforming to DOS path-naming conventions. If *path* does not indicate a valid directory on the current drive, run-time error 76 occurs, "Path Not Found."

The *current directory* is the location where your program is to perform file operations by default. Thus,

```
CHDIR "\DATA"
OPEN "MYFILE.TXT" FOR INPUT AS #1
```

opens a file called *MYFILE.TXT* in the directory *\DATA*.

A program that changes the current directory from within *Portfolio PowerBASIC* also changes *Portfolio PowerBASIC*'s active directory.

Restrictions: CHDIR cannot be used to change the default disk drive.

See also: MKDIR, RMDIR

CHR$ function

CHR$ converts an ASCII code into the corresponding ASCII character.

***s$* = CHR$(*integer expression*)**

CHR$ returns a string containing a single character which represents the ASCII code (*integer expression*) argument. The argument must be a value between 0 and 255. CHR$ complements the ASC function, which returns the ASCII code of a string's first character. CHR$ is handy for creating characters that are difficult to enter at the keyboard, such as graphics characters for screen output and control sequences for printer output.

See also: ASC

CIRCLE statement

CIRCLE draws or erases a circle on the graphics screen.

CIRCLE (*x,y*), *Radius* [*,color*]

(*x,y*) specifies the screen coordinates of the center of the circle. *x* must be a numeric expression which evaluates

to a value between 0 and 239. *y* must be a numeric expression which evaluates to a value between 0 and 63. *Radius* is a numeric expression which specifies the radius of the circle in screen pixel units. Any points on the circle which fall outside of the Portfolio's 240 x 64 screen are not displayed; no run-time error is generated in this case. *color* is an optional numeric expression which evaluates to either 0 or 1. Using 1 for *color* causes the circle to be drawn, while 0 causes it to be erased.

See also: LINE, POINT, PSET, SCREEN

CLEAR statement

CLEAR zeros all variables.

CLEAR

CLEAR sets all numeric variables to zero and all string variables to the null string.

Warning! A CLEAR statement inside of a loop clears the counter each time it is executed, causing an endless loop.

CLOSE statement

CLOSE closes a file.

CLOSE [[#]*filenum* [, [#]*filenum*]...]

CLOSE ends the relationship between a file handle (or number) and the disk file that was associated with the handle by an OPEN statement. I/O to that file is concluded, any associated buffer is flushed, and a DOS **CLOSE** is performed on it to update the directory entry.

It's good practice to periodically CLOSE files that a program writes. This ensures that all the information is physically written to the disk, and that the file's directory entry is properly updated, minimizing the possibility of data loss in the event of a subsequent power failure or other problem.

A CLOSE without a file number closes all open files (as does END).

See also: END, OPEN

CLS statement

CLS clears the screen.

CLS

CLS clears the screen in text or graphics mode and moves the cursor to the upper left corner (row 1, column 1).

COMMAND$ function

COMMAND$ returns the command line used to start the program from DOS.

*s$* = **COMMAND$**

COMMAND$ returns everything that was typed following the program name when the program was started from the DOS prompt (some DOS manuals refer to this text as the trailer).

Use COMMAND$ to collect run-time arguments, like file names and program options.

For example, consider a program named *FASTSORT* that reads data from one file, sorts it, and puts the result in a new file. Using COMMAND$ lets you specify the input and output file names when the program is invoked:

```
A> FASTSORT cust.dta cust.new
```

When *FASTSORT* begins execution, COMMAND$ will hold the string "**cust.dta cust.new**". *FASTSORT* must include code to parse this string into two file names.

COS function

COS returns the trigonometric cosine of its argument.

y = **COS(*numeric expression*)**

numeric expression is an angle specified in radians. To convert radians to degrees, multiply by 57.2958. To convert degrees to radians, multiply by 0.017453. For more information on radians, see the ATN entry.

COS returns a double-precision value that always ranges between –1 and +1.

See also: ATN, SIN, TAN

CSRLIN function

CSRLIN returns the current vertical position (row number) of the screen cursor.

y = **CSRLIN**

CSRLIN returns an integer representing the current vertical position (row number) of the cursor.

Use the POS function to read the cursor's horizontal position (column number). Use the LOCATE statement to move the cursor to a specific line and column.

See also: LOCATE, LPOS, POS

CVI, CVS, and CVD functions

These functions convert string data read from random-access files to numeric form.

integervar% = CVI(*2-byte string*)
singlevar! = CVS(*4-byte string*)
doublevar# = CVD(*8-byte string*)

These functions are used only for random-access file processing.

Command	Variable	Converts to
CVI	2-byte string	Integer
CVS	4-byte string	Single-precision float
CVD	8-byte string	Double-precision float

Because of the way *Portfolio PowerBASIC* handles random-access files, numeric values must be translated into strings (using MKI$ and its sister functions) before they can be written to disk, and then translated back into numbers when the file is read. Don't confuse these functions with VAL, which takes a string like "3.7" and turns it into a number.

See also: FIELD, GET, MKI$ and associated functions

DATA statement

DATA declares constants in the source code to be read by READ statements.

DATA *constant* [,*constant*]...

constant is a numeric or string constant. Numeric constants can be floating point or integer. String constants don't have to be enclosed in quotes unless they contain delimiters (commas or colons) or significant leading or trailing blanks, and can be freely mixed with numeric constants. For example:

```
DATA Sprouts,.79,Avocado Burger,2.29,"Shrimp Toast"
```

A program can contain as many DATA statements as needed, and they can be located anywhere in the source code; they need not be on sequential lines. Each DATA statement can contain as many constants as can fit on a single line.

At run time, READ statements access the DATA statements in the order in which they appear in the source program, and individual constants from left to right within each DATA statement. The most common error associated with reading DATA statements is improper synchrony between READ statements and DATA constants, which usually results in the program trying to load string data into a numeric variable; this generates a syntax error (run-time error 2). You won't get an error if you load numeric constants into string variables, even though that is probably not what you intended.

The RESTORE statement lets you reset the READ pointer so that constants can be reread from the first statement or any specified DATA statement. If you try to READ more times than your program has constants in DATA statements, run-time error 4 results, "Out of data." No error is generated, however, if some DATA constants go unread.

Restrictions: You cannot use underscore continuation characters in DATA statements. Don't use the single quote (') to comment a DATA statement; *Portfolio PowerBASIC* will think that the last entry and your comment are part of a single, long string constant. For example,

```
DATA cats,dogs,pigs  'list the animals
```

is interpreted as containing three string constants: *"cats"*, *"dogs"*, and *"pigs 'list the animals"*. You can, however, safely use REM or the single quote for this purpose if you precede it with a colon (*Portfolio PowerBASIC*'s statement separator):

```
DATA cats,dogs,pigs  :REM list the animals
DATA cows,hogs,frogs :' more animals
```

See also: READ, RESTORE

DATE$ system variable

DATE$ is used to set and retrieve the system date.

**DATE$ = *s$*** (sets system date according to *s$*)

***s$* = DATE$** (*s$* now contains system date)

Assigning a properly formatted string value to DATE$ sets the system date. You can also assign DATE$ to a string variable which stores 10 characters in the form "mm-dd-yyyy", where *mm* represents the month, *dd* the day, and *yyyy* the year.

To change the date, your date string must be formatted in one of the following ways:

```
mm-dd-yy
mm/dd/yy
mm-dd-yyyy
mm/dd/yyyy
```

For example, **DATE$ = "10-17-89"** sets the system date to October 17, 1989.

See also: TIME$

DEF FN/END DEF statement

DEF FN/END DEF defines a function.

Single-line function:
DEF FN*Identifier* [(*argument list*)] = *expression*

Multi-line function:

```
DEF FNIdentifier [(argument list)]
  .
  . {statements}
  .
  [EXIT DEF]
  .
  .
  .
  [FNIdentifier = expression]
END DEF
```

Identifier is the name of the function. *It must be unique*: no other variable, function, procedure, subroutine, or label can share it.

argument list is an optional, comma-delimited sequence of formal parameters. The parameters used in the argument list serve only to define the function; they have no relationship to other variables outside of the function with the same name.

DEF FN and END DEF bracket and name a subroutine-like group of statements called a DEF FN function. A DEF FN function may be passed one or more arguments (which are passed by value, not by reference). A DEF FN function returns a value; the type of the value is determined by the symbol that terminates the function's name (for example, a function named *FNCalculate%* returns an integer value, *FNInterest#* returns a double-precision floating point value, and so on). A DEF FN function may, therefore, be called from within any statement that can accept a value of the function's type.

Function definitions and program flow

The position of DEF FN function definitions within your source code is immaterial, although clarity is served by grouping them together in one area. You need not direct program flow through a DEF FN function as an initialization step. The compiler sees your definitions wherever they might be.

Also, unlike subroutines, execution can't accidentally "fall into" a DEF FN function. As far as the execution path of a program is concerned, function and procedure definitions are invisible. For example,

```
t = FNPrintStuff
DEF FNPrintStuff
  PRINT "Printed from within FNPrintStuff!"
END DEF
END
```

When this program is executed, the message appears only once.

Function definitions should be treated as isolated islands of code; don't jump into or out of them with GOTO, GOSUB, or RETURN statements. Within definitions, however, such statements are legal.

Function and procedure definitions can't be nested; that is, it is illegal to define a procedure or function within a another procedure or function (although a procedure or function can call other procedures and functions).

Variables which appear within DEF FN function definitions have the SHARED attribute; that is, they are global to the rest of the program.

When a DEF FN function is referenced, the name of the function (*Identifier*) must always be preceded by *FN*:

```
t = FNMyFunc%(a,b)
```

Use the EXIT DEF statement to return from a DEF FN function before reaching its END DEF statement.

See also: END, EXIT, GOSUB

DEFINT, DEFSNG, DEFDBL, and DEFSTR statements

The DEF*type* statements declare the default type for variable identifiers that begin with specified letters.

DEF*type letter range* [,*letter range*]...

type represents one of the four *Portfolio PowerBASIC* variable types: INT (integer), SNG (single-precision floating point), DBL (double-precision floating point), and STR (string).

letter range is either a single alphabetic character (*A* through *Z*, case insignificant), or a range of letters (two letters separated by a hyphen; for example, *A-M*).

DEF*type* tells the compiler that variables and user-defined functions whose names begin with the specified letter or range of letters are of the specified type. This allows you to use variables other than single-precision floating point in your program without including type identifiers (for example, % and #).

Normally, when the compiler finds a variable name without a type identifier, it assumes the variable to be single-precision floating point. For example, in this statement, both *blint* and *lemming* are assumed to be single precision:

```
blint = lemming + 17
```

If, however, this statement was preceded by

```
DEFINT b, l
```

then *blint* and *lemming* would both be integer variables, as would any other variable whose name started with *b* or *l* in uppercase or lowercase.

Restrictions: A DEF*type* statement redefines the type of any corresponding variables that are already being used in the program. The example program demonstrates this rather subtle point.

DEF SEG statement

DEF SEG defines the data segment to be used by the POKE statement and the PEEK function.

DEF SEG [= *numeric expression*]

numeric expression can range from –32768 to +65535, and specifies the base address of the memory segment in which subsequent POKE statements and PEEK functions will operate. Segments are a part of the memory addressing method used by Intel 80x86 family processors. Addresses are normally specified with two 16-bit integers: a segment (as set with DEF SEG) and an offset into the segment (as specified in POKE or PEEK). For example, the following code:

```
DEF SEG = segment
x = PEEK(offset)
```

retrieves the byte stored at absolute memory address **segment * 16 + offset**.

DEF SEG with no argument resets the segment value to its start-up default value, which is *Portfolio PowerBASIC*'s main data segment.

See page 52 for more about positive and negative representations of the segment value.

See also: PEEK, POKE

DIM statement

DIM declares single- or multi-dimensional arrays.

DIM *Var(subscripts)* [,*Var(subscripts)*]...

DIM declares *Var* to be an array whose type is specified by appending a type specifier to the name or changing the default variable type for identifiers beginning with the same letter by using the DEF*type* statement. *subscripts* is a comma-delimited list of one or more integer constants defining the dimensions of the array. Each dimension begins with element zero. For example, **DIM MyArray(20)** defines an array of one dimension that has 21 elements, from **MyArray(0)** to **MyArray(20)**. **DIM Array3D(5,5,5)** defines a 6×6×6 array (with a total of 216 elements), where each dimension consists of elements 0 through 5.

Portfolio PowerBASIC sets each element of a numeric array to zero when a program is first executed, and sets each element of string arrays to the null string (length zero). If a CLEAR statement is executed, numeric arrays are reset to 0 and string arrays to the null string.

If an array variable is used without a preceding DIM statement, the compiler automatically dimensions it with a maximum value of 10 for each of its subscripts. It is good practice, however, to explicitly declare every array.

See also: CLEAR, FRE

DO/LOOP statement

DO/LOOP defines a group of program statements that are executed repetitively as long as a certain condition is met.

DO [{WHILE | UNTIL} *expression*]
.
.
.
***statements* [EXIT LOOP]**
.
.
.
{LOOP | WEND} | [{WHILE | UNTIL} *expression*]

expression is a numeric expression, in which nonzero values represent logical TRUE and zero values represent logical FALSE.

DO/LOOP statements are extremely flexible. They can be used to create loops for almost any imaginable programming situation. They allow you to create loops with the test for the terminating condition at the top of the loop, the bottom of the loop, both places, or neither. A DO statement must always be paired with a matching LOOP statement at the bottom of the loop. Failure to match each DO with a LOOP results in a compile-time error.

The WHILE and UNTIL keywords are used to add tests to a DO/LOOP. Use the WHILE if the loop should be repeated if *expression* is TRUE, and terminated if *expression* is FALSE. UNTIL has the opposite effect; that is, the loop will be terminated if *expression* is TRUE, and repeated if FALSE.

For example,

```
DO WHILE a = 13
  {statements}
LOOP
```

executes the statements between DO and LOOP as long as *a* is 13. If *a* is not 13 initially, the statements in the loop are never executed. In contrast,

```
DO UNTIL a = 13
  {statements}
LOOP
```

executes the statements between DO and LOOP as long as *a* is not 13. If *a* equals 13 initially, the loop is never executed.

At any point in a DO/LOOP you can include an EXIT LOOP statement. This is equivalent to performing a GOTO to the instruction after the terminating LOOP statement. For more information, see EXIT.

The WHILE/WEND statements can be used in many cases to perform the same functions as DO/LOOP. For example, this DO/LOOP:

```
DO WHILE a < b
  {statements}
LOOP
```

has the same effect as this WHILE/WEND loop:

```
WHILE a < b
  {statements}
WEND
```

DO loops can be nested with other DO loops and with WHILE/WEND and FOR/NEXT loops. Nesting must be complete; that is, an outer loop must contain both the start and finish of any inner loops. The following code will not compile because the WHILE/WEND loop is not completely within the DO loop:

```
DO WHILE X < 10
  WHILE Z > 50
  {statements}
LOOP
  WEND
```

When using nested loops, be careful that inner loops do not modify variables that are used by the outer loop's terminating condition test. For example, the following code was intended to print out all 20 elements of a 2×10 array (dimensioned *arry*(9,1)):

```
Count1 = 0
DO WHILE Count1 < 10
  FOR Count2 = 0 TO 1
    PRINT arry(Count1,Count2)
    Count1 = Count1 + 1
  NEXT Count2
LOOP
```

Because *Count1* is incremented within the inner loop, which executes twice for each pass through the outer

loop, this code would not print all the array values, but would only print out the values for *arry*(0,0), *arry*(1,1), *arry*(2,0), *arry*(3,1) and so on. By moving the **Count1 = Count1 + 1** statement to just below the **NEXT Count2** statement, the code functions as intended.

If an EXIT LOOP statement is used within nested loops, it exits only the current loop, not the entire nest.

The *Portfolio PowerBASIC* logical operators can be used to construct multiple test conditions for loop control. For example,

```
DO WHILE x < 10 AND y < 10
  {statements}
LOOP
```

is executed only as long as both *x* and *y* are less than 10. Similarly, the loop

```
DO UNTIL X > 10 OR Y > 10
  {statements}
LOOP
```

is executed until either *x* or *y* (or both) is (are) greater than 10.

Although the compiler doesn't care about such things, it's a good idea when writing your source code to indent the statements between DO and LOOP. The same is true of FOR/NEXT loops, WHILE/WEND loops, and multi-line IF statements. Such indenting makes the *appearance* of your source code reflect the *logical structure* of your program, resulting in greater readability. Indenting is particularly valuable when nesting

multiple loops of the same type, since it makes it easier to see which LOOP goes with which DO.

See also: EXIT, FOR/NEXT, IF, WHILE/WEND

END statement

END terminates execution of a program or defines the end of a structured block.

END [{DEF | IF | SUB}]

END without arguments terminates program execution. An END statement can be placed anywhere in a program, and there can be more than one. Encountering an END causes a program to close all open files and return to DOS.

An END statement isn't strictly required by the compiler, although using it is good practice. If a program simply runs out of statements to execute, an implied END operation is performed.

END followed by one of the DEF, IF, or SUB keywords defines the end of a structured block. END does *not* terminate program execution if it is followed by one of these keywords.

See also: CLOSE, DEF FN, IF, SUB

EOF function

EOF returns the end-of-file status of an opened file.

***y* = EOF(*filenum*)**

Use EOF to determine when the end of a file has been reached while reading its data. *filenum* is the file number specified when the file was OPENed; if it does not refer to a valid, open file, a run-time error occurs. EOF returns logical TRUE (non-zero) if the end of the specified file has been reached; otherwise, it returns logical FALSE (zero).

For a sequential input file, end-of-file has been reached if there is no more data to be read from the file *or* if an EOF character (ASCII code 26) has been read.

For random-access and binary files, end-of-file has been reached if the most recent file read operation was unable to read as many characters as requested (or as many as required by the random-access buffer size).

For a communications file, EOF returns TRUE if the communications buffer for that file is empty; otherwise it returns FALSE.

ERL and ERR functions

ERL and ERR return the program line number and error code of the most recent *Portfolio PowerBASIC* run-time error.

y = ERL
y = ERR

ERR returns the error code of the most recent run-time error. This number can be tested in an error-trapping routine (declared with the ON ERROR statement), so that appropriate error-handling code can be executed. Use the RESUME {*label* | *line number*} statement to continue program execution after error trapping.

ERL returns the line number of the most recent error. If the error occurs in a statement without a line number, ERL returns the number of the most recently executed line which had a line number. If no numbered lines have been executed, ERL returns zero.

See also: ERROR, ON ERROR, RESUME

ERROR statement

ERROR simulates the occurrence of a specific run-time error.

ERROR *errcode*

errcode is an integer expression from 0 to 255. If *errcode* is one of *Portfolio PowerBASIC*'s predefined run-time error codes, then ERROR causes your program to behave as though that error had actually occurred. Use the ERROR statement as an aid in debugging error-trapping routines.

To define your own error codes, use values for *errcode* that aren't used by *Portfolio PowerBASIC*. If you don't define an error-handling procedure for these new custom error codes, your running program will display the message:

Error *n*

where *n* is the error code. If any numbered lines have been executed prior to the ERROR statement, the line number of the most recently executed numbered line will be displayed in the error message generated:

```
10 PRINT
ERROR 17      'displays "Error 17 at line 10"
END
```

See also: ERL, ERR, ON ERROR, RESUME

EXECUTE statement

EXECUTE transfers control to the specified program.

EXECUTE *filespec*

filespec is a string expression which follows standard DOS file-naming conventions and represents the program which *Portfolio PowerBASIC* will transfer control to. EXECUTE can run .EXE, .COM, or .RUN files. It cannot run .BAT files, or any other sort of file which would require a DOS command interpreter (such as the IBM PC's **COMMAND.COM** file) to be invoked. When the program is finished executing, control returns to DOS, rather than to the program which performed the EXECUTE. If the file to be executed is not present, no error is generated; control simply returns to DOS.

EXIT statement

EXIT transfers program execution out of a structure.

EXIT {DEF | FOR | IF | LOOP | SUB}

The EXIT statement lets you leave a structure prematurely. The type of structure being EXITed must be included as part of the EXIT statement, as follows:

EXIT option	Structure exited
DEF	DEF FN/END DEF function definition
FOR	FOR/NEXT loop
IF	IF/END IF block
LOOP	DO/LOOP or WHILE/WEND loop
SUB	SUB/END SUB procedure definition

Using EXIT is preferred over using GOTO. Note the difference between EXIT and END: EXIT operates during program execution to transfer control out of a structure; END operates during program compilation to mark the end of the structure's source code.

Restrictions: When using EXIT to leave a function, you must assign the function a result before EXIT is executed or the value returned by the function will be undefined.

See also: DEF FN, DO/LOOP, END, FOR/NEXT, IF, SUB, WHILE/WEND

EXP function

The EXP function returns *e* (2.718282...) raised to a power.

y = EXP(x)

EXP returns *e* to the *x*th power, where *x* is a numeric expression and *e* is the base for natural logarithms, approximately 2.718282. You would get the same result with the statement *e*^*x*. One thing you can do with EXP is calculate *e* itself:

```
e = EXP(1)
```

EXP returns a double-precision result.

See also: LOG, SQR

FIELD statement

FIELD defines the variables of a random-access file buffer.

FIELD [#]*filenum, width* AS *string-var* [,*width* AS *string-var*]...

filenum is the number used when the file was opened. *width* is the number of bytes allocated to each field variable. *string-var* is the field variable itself, which must be a string.

FIELD defines a mapping between string variables and the I/O buffer of a random-access file. Once used in a FIELD statement, string variables gain a special status as "field variables." They should only be assigned to using LSET and RSET in preparation for writing to the indicated random-access file.

Restrictions: A string identifier used in a FIELD statement should never be used on the left side of an assignment statement. Doing so disassociates the identifier from the random-access file's field definitions, causing it to revert to a normal string.

See also: GET, LSET, PUT, RSET

FOR/NEXT statement

FOR and NEXT define a loop of program statements whose execution is controlled by an automatically incrementing or decrementing counter.

FOR *Counter* = *start* TO *stop* [STEP *increment*]
.
. **{statements}**
.
NEXT [*Counter* [,*Counter*]...]

Counter is a numeric variable serving as the loop counter; *start* is a numeric expression specifying the value initially assigned to *Counter*; *stop* is a numeric expression giving the value that *Counter* must reach for the loop to be terminated; *increment* is an optional numeric expression defining the amount by which *Counter* is incremented with each loop execution. If not specified, *increment* defaults to 1.

When a FOR statement is encountered, *start* is assigned to *Counter* and *Counter* is tested to see if it is greater than (or, for negative *increment*, less than) *stop*. If not, the statements within the FOR/NEXT loop are executed, *increment* is added to *Counter*, and *Counter* again tested against *stop*. The statements in the loop are executed repeatedly until the test fails, at which time control passes to the statement immediately following the NEXT.

When using floating-point values with FOR/NEXT, be sure to allow for round-off errors when mixing numbers of different precisions.

FOR/NEXT loops run fastest when *Counter* is an integer variable and *start*, *stop*, and *increment* are integer constants. The value of *Counter* is available like any other variable within the loop. It is wise to avoid explicitly modifying the value of *Counter* within the loop. If you need to exit the loop prematurely, use an EXIT FOR statement. If you use the maximum value of an integer (32767) as a stopping condition, you'll get an overflow error message when the variable is incremented in the NEXT statement.

The body of the loop is skipped altogether if the initial value of *Counter* is greater than *stop* (or, for a negative *increment*, if *Counter* is less than *stop*).

FOR/NEXT loops can be nested within other FOR/NEXT loops. Be sure to use unique counter variables and to make sure that the inner loop's NEXT statement occurs before the outer loop's NEXT. This code has crossed loops and won't compile:

```
FOR n = 1 TO 10
  FOR m = 1 TO 20
    .
    .
    .
  NEXT n
NEXT m
```

If multiple loops end at the same point, a single NEXT statement containing each counter variable suffices:


```
FOR n = 1 TO 10
FOR m = 1 TO 20
    .
    .
    .
NEXT m, n
```

The counter variable in the NEXT statement can be omitted altogether, but if you include it, it must be the right variable. For example,

```
FOR n = 1 TO 10
    .
    .
    .
NEXT   'NEXT n would work too, but not NEXT m
```

Although the compiler doesn't care about such things, indent the statements between FOR and NEXT by two or three spaces to set off the structure of the loop.

If a NEXT is encountered without a corresponding FOR (or vice versa), a compiler error is generated.

FRE function

FRE returns the amount of free memory available to your program.

freememory = FRE({0 | -1 | -2})

FRE with an argument of zero returns a value representing the number of bytes available to be allocated

as a single string, up to the maximum string size (32750). FRE(-1) returns the number of bytes of available memory. FRE(-2) returns the smallest amount of stack space (in bytes) which was *ever* available during the execution of your program.

FRE(*0*) tells you how large the largest continuous block of unused string memory is, up to the maximum string size. For example, if you have 60K of memory available while your program is running, FRE(*0*) will return 32750 (the maximum string size). If you have only 16000 bytes of memory available however, FRE(*0*) will return 16000. FRE(*0*) does *not* tell you the total amount of memory available in the machine for strings.

See also: $STACK, CLEAR

GET statement

GET reads a record from a random-access file.

GET [#]*filenum* [, *recnum*]

filenum is the number under which the file was opened, and *recnum* is the record to be read. If *recnum* is omitted, then the next record in sequence (following the one specified by the most recent GET or PUT) is read; if the file was just opened, the first record is read.

GET reads the indicated record from the file and puts the data into the variable(s) of the random-access file buffer associated with that file.

If *recnum* is greater than the number of records in the file, no error occurs but unpredictable data may be read in. Use the EOF function to avoid GETting past the end of the file.

See also: EOF, FIELD, LOC, LSET, PUT, RSET

GET$ function

GET$ reads a string from a file opened in binary mode.

GET$ [#] *filenum, Count, string variable*

Count is an integer expression ranging from 0 to the maximum string size (32750). GET$ reads *Count* characters from file number *filenum* and assigns them to *string variable*. File *filenum* must have been opened in binary mode. Characters are read starting at the current file pointer position, which can be set with the SEEK statement. When the file is first opened, the pointer is at the beginning of the file (position 0). After GET$, the file pointer position will have been advanced by *Count* bytes.

GET$, PUT$, and SEEK provide a low-level alternative to sequential and random-access file-processing techniques, allowing you to deal with files on a byte-by-byte basis.

See also: EOF, LOC, LOF, OPEN, PUT$, SEEK

GOSUB statement

GOSUB invokes a subroutine.

GOSUB {*label* | *linenumber*}

GOSUB causes *Portfolio PowerBASIC* to jump to the statement prefaced by *label* or *linenumber*, after first saving its current location on the stack. Executing a RETURN statement returns control to the instruction immediately following the GOSUB.

When using GOSUB, be sure that each subroutine returns to its caller gracefully through a RETURN statement. Run-away GOSUBs that loop upon themselves will eat up large chunks of stack space and eventually cause the program to run out of memory, or at least withhold memory from the program that would otherwise be available.

As useful as GOSUBs can be, *Portfolio PowerBASIC*'s procedures and functions can do the work of subroutines with the added benefits of recursion, parameter passing, and local variables, within bodies of enclosed, protected code.

See also: $STACK, DEF FN, ON/GOSUB, SUB, RETURN

GOTO statement

GOTO transfers program execution to the statement identified by a label or line number.

GOTO {*label* | *linenumber*}

GOTO causes program flow to jump unconditionally to the code identified by *label* or *linenumber*. GOTO differs from GOSUB and other similar control statements in that, after execution of a GOTO, the program retains no memory of where it was before it executed the jump.

Used with care, GOTOs can be a fast, effective programming device. Used carelessly, they can choke a program with tangled spaghetti-like strands of code that can be almost impossible to puzzle out (especially months or years after they are written).

Modern structured programming practice discourages the use of GOTOs. In fact, the GOTO statement is one of the major reasons why many programming "pundits" disparage BASIC in favor of other languages such as C. While these complaints were valid in regard to early versions of BASIC, *Portfolio PowerBASIC*'s functions and procedures provide sufficient structured control of program flow. You'll rarely, if ever, need to use GOTO. The FOR/NEXT, WHILE/WEND, DO/LOOP, and IF block structures, as well as the EXIT statement, assist in GOTO reduction.

See also: CALL, DO/LOOP, EXIT, FOR/NEXT, GOSUB/RETURN, IF block, SUB, WHILE/WEND

HEX$ function

HEX$ returns a string that is the hexadecimal (base 16) representation of its argument.

s$ = HEX$(*numeric expression*)

numeric expression must be in the decimal range –32768 to 65535; if outside this range, run-time error 6 (overflow) is generated. Any fractional part of *numeric expression* is rounded before the string is created.

Hexadecimal is a number system that uses base 16 rather than base 10 (used by the everyday decimal system). In hexadecimal, the digits 0 to 9 represent the same numbers as in decimal, and the letters *A* to *F* represent the decimal values 10 to 15. Hexadecimal notation is commonly used in programming because the binary bit patterns used internally by computers translate directly into hex digits. A single hex digit represents 1 nibble (4 bits), two hex digits represent one byte, and so on.

Binary	Octal	Decimal	Hex
0000 0000	0	0	0
0000 0001	1	1	1
.	.	.	.
.	.	.	.
.	.	.	.
0000 0111	7	7	7
0000 1000	10	8	8
0000 1001	11	9	9
0000 1010	12	10	A
0000 1011	13	11	B
0000 1100	14	12	C
0000 1101	15	13	D
0000 1110	16	14	E
0000 1111	17	15	F
0001 0000	20	16	10
.	.	.	.
.	.	.	.
.	.	.	.
1111 1110	376	254	FE
1111 1111	377	255	FF

HEX$ returns the two's complement form of a negative argument (see the BIN$ function entry and page 52 for a discussion of two's complement arithmetic).

See also: BIN$, OCT$

IF statement

IF tests a condition and executes one or more program statements only if the condition is met.

IF *integer expression* THEN {*statements*} [ELSE {*statements*}]

If *integer expression* is TRUE (evaluates to a nonzero value), the statements following THEN are executed and the statements following the optional ELSE are not executed. If *integer expression* is FALSE (zero), then the statements following THEN are not executed and the statements following the optional ELSE are executed. If the ELSE clause is omitted, execution continues with the next line of the program if *integer expression* is FALSE.

integer expression will often be a result returned by a relational operator as shown here:

```
IF Inc > Exp THEN PRINT A% ELSE PRINT B%
```

It can also be a single flag value. For example, your program could set the variable *PrinterOn* to 1 if the printer is available, and to 0 if it isn't, then use an IF statement to control output:

```
IF PrinterOn THEN LPRINT answer$
```

integer expression can include the logical operators AND and OR, as in:

```
IF (a = b) AND (c = d) THEN PRINT "They are equal"
```

The IF statement and all its associated statements, including those after an ELSE, must appear on the same logical program line. The following is therefore illegal:

```
IF a < b THEN t = 15 : u = 16 : v = 17
ELSE t = 17 : u = 16 : v = 15
```


because the compiler treats the ELSE statement as a brand-new statement unrelated to the one above it. If you have more statements than can fit on one line, you can use the line continuation character, the underscore (_), to spread a single logical line over several physical lines. For example, the following is a legal way of restating the last example:

```
IF a < b THEN t = 15 : u = 16 : v = 17 _
ELSE t = 17 : u = 16 : v = 15
```

A colon must not appear before the ELSE keyword. For example, the following statement won't compile:

```
IF a < b THEN c = d : ELSE e = f
```

A better method of programming long and complex IF/THEN constructs is to use the IF block statement.

See also: IF block

IF block statement

The IF block creates IF/THEN/ELSE constructs with multiple lines and/or conditions.

IF *integer expression* THEN
{statements}
[ELSEIF *integer expression* THEN
{statements}]
[ELSE
{statements}]
END IF

In executing IF block statements, the truth of the *integer expression* in the initial IF statement is checked first. If it evaluates to FALSE (zero), each of the following ELSEIF statements is examined in order (there can be as many ELSEIF statements as desired). As soon as one is found to be TRUE, *Portfolio PowerBASIC* executes the statement(s) following the associated THEN and before the next ELSEIF or ELSE, then jumps to the statement just after the terminating END IF without making any further tests. If none of the test expressions evaluates to TRUE, the statement(s) in the ELSE clause (which is optional) are executed.

Note that there must be nothing following the THEN keyword on the first line of an IF block; that's how the compiler distinguishes an IF block from a conventional IF statement. There must also be nothing on the same line as the ELSE.

IF block statements can be nested; that is, any of the statements after any of the THENs may contain IF blocks. Although the compiler doesn't care, the clarity of source code is improved by indenting the statements controlled by each test a couple of spaces, as shown in the example.

IF block statements must be terminated with END IF. Note that END IF requires a space and ELSEIF does not.

See also: END, IF

INKEY$ function

INKEY$ reads a character from the keyboard without echoing the character to the screen.

***s$* = INKEY$**

INKEY$ returns a string of zero, one, or two characters that reflects the status of the keyboard buffer.

A null string (LEN(*s$*) = 0) means that the buffer is empty. A string of length one (LEN(*s$*) = 1) means that an ASCII key was pressed and the string contains the ASCII character.

A string of length two (LEN(*s$*) = 2) means a non-ASCII key was pressed. In this case, the first character in the string has a value of 0, and the second is an extended keyboard code that represents one of the keyboard's non-ASCII keys (such as a function or an arrow key).

INKEY$ is a flexible mechanism for getting user input into your program without the restrictions of the INPUT statement. Since INKEY$ doesn't wait for a character to be pressed before returning a result, you will usually use it within a loop in a low-level routine, continuously checking it and building an input string to be checked by higher-level routines. Sometimes it is desirable to empty the keyboard buffer before accepting input to be sure that inadvertent keystrokes are not included. For example:

```
'empty keyboard buffer
WHILE INKEY$ <> "" : WEND
' prompt for input, get characters one at a time
PRINT "Enter some characters followed by <RTN>"
'endless loop
DO
  'get one character
  Char$ = INKEY$
  'exit if it's RTN
  IF Char$ = CHR$(13) THEN EXIT LOOP
  ' add it to input string
  InputString$ = InputString$ + Char$
LOOP
PRINT InputString$
END
```

INKEY$ passes all keystrokes, including control key combinations (like *Ctrl-Tab*, *Ctrl-RTN*, and *Ctrl-Backspace*), to your program without displaying or processing them, with the following exception:

- *Ctrl-Alt-Del* causes a system reset.

See also: INPUT, INPUT$, INSTAT, LINE INPUT

INP function

INP reads a byte from a processor I/O port.

y = INP(*portno*)

INP returns the byte read from I/O port *portno*, where *portno* indicates a hardware input port in the range

–32768 to +65535. If no device is attached to the specified port, the return value is undefined.

INP is used for reading status information presented by various hardware subsystems, such as communications ports and other I/O devices. See the Portfolio's technical reference manual for port assignments.

The OUT statement is used to write to an I/O port.

See also: OUT

INPUT statement

INPUT prompts the user for keyboard entry and assigns the input to one or more variables.

INPUT [;] [*prompt string* {; | ,}] *variable list*

prompt string is an optional string constant. *variable list* is a comma-delimited sequence of one or more string or numeric variables. You can read up to 255 characters into each string variable.

INPUT displays *prompt string* on the screen, waits for the user to enter data from the keyboard, and assigns the data to the variable(s) in *variable list*. If you include a semicolon after *prompt string*, *Portfolio PowerBASIC* outputs a question mark after the string. Use a comma instead to suppress the question mark.

Data entered from the keyboard must match the types of the variables in *variable list*; that is, nonnumeric characters are unacceptable for numeric variables.

If a single INPUT statement prompts for more than one variable, then you must enter the proper number of values on a single line, separated by commas or spaces. For example, in response to

```
INPUT A%,B%,C%
```

you could enter

```
100,200,300
```

If a semicolon appears immediately after the INPUT keyword, the cursor will remain on the same line when you press *RTN* to terminate the response. Otherwise, a carriage-return/linefeed pair is sent to the display and the cursor moves to the beginning of the next line.

Use LINE INPUT instead of INPUT when you need to enter string information which contains delimiters (that is, commas) that would otherwise confuse an INPUT statement.

See also: INKEY$, INPUT #, INPUT$, LINE INPUT, LINE INPUT #

INPUT # statement

INPUT # loads variables with data from a sequential file.

INPUT #*filenum, variable list*

filenum is the number given when the file or device was opened, and *variable list* is a comma-delimited sequence of one or more string or numeric variables.

The data in the file must match the type(s) of the variable(s) defined in the INPUT # statement. The file data should appear just as if it were being typed from the keyboard in response to an INPUT statement; that is, it should be separated by commas with a carriage return at the end.

See also: INPUT, INPUT$, LINE INPUT #

INPUT$ function

INPUT$ reads a specific number of characters from a file, a device, or the keyboard.

***s$* = INPUT$(*n* [, [#] *filenum*])**

n is the number of characters to be read, and *filenum* is the file to read from. If you omit *filenum*, the keyboard is read. If the keyboard is used, the typed characters are not echoed on the screen.

INPUT$ accepts all characters, including control characters.

INPUT$ is primarily used for keyboard input, though it can also read characters from sequential and random-access files and the communications port. Binary-mode files (accessed using GET$ and PUT$) offer more flexibility when working with data on a character-by-

character basis. INKEY$ offers more flexibility when reading from the keyboard.

Certain keys and key combinations (for example, the function and cursor control keys) don't return ASCII values. When such keys are pressed, INPUT$ substitutes CHR$(0); INKEY$ doesn't have this limitation.

See also: GET$, INKEY$, INPUT, INSTAT, LINE INPUT, PUT$

INSTAT function

INSTAT returns the keyboard status.

***y* = INSTAT**

INSTAT returns keyboard status information. If a key has been pressed, INSTAT returns logical TRUE (non-zero); otherwise, it returns logical FALSE (zero). Its most common use is in loops that suspend program execution until the user presses a key:

```
PRINT "Press any key to continue"
WHILE NOT INSTAT : WEND
K$ = INKEY$
```

INSTAT doesn't remove a keystroke from the keyboard buffer, so if it ever returns TRUE, it will continue to return TRUE until the keystroke is removed by INKEY$ or another keyboard-reading function. That's the reason for the **K$ = INKEY$** statement in the above code fragment—while you don't need to know the specific key that was pressed, you must remove it from the keyboard buffer so it won't cause problems later in the program.

See also: INKEY$

INSTR function

INSTR searches a string for the first occurrence of a specified character or string.

***y* = INSTR([*n*,] *main string, match string*)**

n is an integer expression ranging from 1 to the maximum string size (32750), and *main string* and *match string* are string variables, expressions, or constants.

INSTR returns the position of *match string* in *main string*. If *match string* is not in *main string*, INSTR returns 0. If the optional *n* parameter is included, the search begins at position *n* in *main string*. If *match string* is null (length 0), INSTR returns 1 (if *n* not specified) or *n* (if *n* is specified).

INSTR is case-sensitive, meaning that upper- and lowercase letters must match exactly in *match* and *main string*.

See also: LEFT$, MID$, RIGHT$

INT function

INT converts a numeric expression to an integer.

***y* = INT(*numeric expression*)**

INT returns the largest integer less than or equal to *numeric expression*.

KILL statement

KILL deletes a disk file.

KILL *filespec*

filespec is a string expression specifying the file or files to be deleted, and can include a path name and/or wildcard characters. For example,

```
KILL "TEST.DOC"
FileName$ = "*.BAS"
KILL FileName$ 'potentially dangerous!
KILL "C:\DATA\INCOME.?87"
```

If *filespec* does not exist, error 53 ("File not found") is generated. If *filespec* is read-only, error 75 ("Path/File access error") occurs.

KILL is analogous to the DOS **DEL (ERASE)** command. KILL cannot delete a directory. Use RMDIR instead, after first deleting all the files in the directory.

LCASE$ function

LCASE$ returns an all-lowercase version of its string argument.

***s$* = LCASE$(*string expression*)**

LCASE$ returns a string equal to *string expression*, except that all the uppercase letters in *string expression* are converted to lowercase.

See also: UCASE$

LEFT$ function

LEFT$ returns the leftmost *n* characters of a string.

***s$* = LEFT$(*string expression, n*)**

n is an integer expression and specifies the number of characters in *string expression* to be returned. *n* must be in the range 0 to 32750 (the maximum string size).

LEFT$ returns a string consisting of the leftmost *n* characters of its string argument. If *n* is greater than or equal to the length of *string expression*, all of *string expression* is returned. If *n* is 0, LEFT$ returns the null string.

See also: INSTR, MID$, RIGHT$

LEN function

LEN returns the length of a string.

***y* = LEN(*string expression*)**

LEN returns a value from 0 to the maximum string size (32750), representing the number of characters in *string expression*.

LET statement

LET assigns a value to a variable.

[LET] *variable* = *expression*

variable is a string or numeric variable, and *expression* is of a suitable type (that is, string for string variables and numeric for numeric variables).

LET is optional in assignment statements, and is included to provide compatibility with BASIC source files originally written for early versions of BASIC interpreters. In practice, it is rarely, if ever, used.

LINE statement

LINE draws or erases a straight line on the graphics screen.

LINE (*x1,y1*)-(*x2,y2*) [,*color*]

(*x1,y1*) and (*x2,y2*) specify the graphics screen coordinates of the ends of the line. *x1* and *x2* must be numeric expressions which evaluate to a value between 0 and 239. *y1* and *y2* must be numeric expressions which evaluate to a value between 0 and 63. Any points on the line which fall outside of the Portfolio's 240 x 64 screen are not displayed; no run-time error is generated in this case. *color* is an optional numeric expression which evaluates to either 0 or 1. Using 1 for *color* causes the line to be drawn, while 0 causes it to be erased.

See also: CIRCLE, POINT, PSET, SCREEN

LINE INPUT statement

LINE INPUT reads an entire line from the keyboard into a string variable, ignoring delimiters.

LINE INPUT [;][*prompt string*;] *string variable*

prompt string is an optional string constant. LINE INPUT displays *prompt string* on the screen and waits for user input. You can read up to 255 characters into each string variable. Keystrokes are accepted until you press RTN, at which time the resulting string is loaded into *string variable*.

Use LINE INPUT instead of INPUT when you need to enter string information which contains delimiters (that is, commas) that would otherwise confuse an INPUT statement. For example,

```
INPUT "Enter patient address: ", a$
PRINT a$
```

fails if the address contains a comma:

```
Enter patient address: 101 Main Street, Apt 2
101 Main Street
```

LINE INPUT accepts commas without a problem. If a semicolon follows the INPUT keyword, then when *RTN* is pressed to end the input sequence, a carriage return won't be sent to the display (that is, the cursor stays on the same line).

See also: INPUT, INPUT$, LINE INPUT #

LINE INPUT # statement

LINE INPUT # reads a line from a sequential file or communications port into a string variable, ignoring delimiters.

LINE INPUT #*filenum, string variable*

filenum is the number of the file or device to read, and *string variable* is the string variable to be loaded with the data read from the file.

LINE INPUT # is like LINE INPUT except that it reads the data from a sequential file or communications port rather than from the keyboard. LINE INPUT # reads the current record in the file and loads it into *string variable*.

As with LINE INPUT, use LINE INPUT # to collect data that has delimiter characters (commas) mixed in with data items.

See also: INPUT #, LINE INPUT

LOC function

LOC returns the current position of a file pointer.

y = LOC(*filenum*)

filenum is the file number under which the file was opened. The behavior of LOC depends on the mode in which the file was opened.

If *filenum* is a random-access file, LOC returns the number of the last record written or read.

If *filenum* is a sequential file, LOC returns the number of 128-byte blocks written or read since opening the file. By convention, LOC returns 0 for files that have been opened but have not yet been written or read.

If *filenum* is a binary file, LOC returns the file pointer position.

For a communications file, LOC returns the number of characters in the input buffer (waiting to be read).

See also: LOF, SEEK

LOCATE statement

LOCATE positions the screen cursor and/or defines its visibility.

LOCATE [*row*][,[*column*][,[*cursor*]]]

row is an integer expression specifying the screen row on which to position the cursor (1 to 8). *column* specifies the column (1 to 40). *cursor* is a numeric value that controls whether the cursor will be visible (0 means invisible; 1 means visible).

LOCATE is often used before a PRINT statement to control where the output will appear on the screen.

See also: CSRLIN, LPOS, POS, PRINT

LOF function

LOF returns the length of a file.

***y* = LOF(*filenum*)**

filenum is the number under which the file was opened. LOF returns the size of the indicated file in bytes. For communications files, LOF returns the amount of available space left in the communications buffer. If *filenum* does not refer to an open file or device, a run-time error is generated.

See also: LOC, SEEK

LOG function

LOG returns the natural (base *e*) logarithm of its argument.

y = LOG(*numeric expression*)

A logarithm of a number is the power to which the base would have to be raised to yield the number. Thus,

logarithm (base *e*) of n = x if *e*^x = n

and

***e*^log(n) = n**

By definition, the logarithm (any base) of 1 is 0. LOG returns the natural logarithm (base *e*, where *e* = 2.718282...) of its argument. If *numeric expression* is less than or equal to zero, run-time error 5, "Illegal Function Call," results. LOG always returns a double-precision result.

LPOS function

LPOS returns the number of characters on the current line in the printer buffer.

***y* = LPOS(*printer*)**

printer is 0 or 1, either of which selects printer 1.

LPOS reports how many characters have been sent to the printer since the last carriage-return character was

output. In effect, it gives the current horizontal position of the printhead.

See also: CSRLIN, LOCATE, LPRINT, POS, TAB

LPRINT and LPRINT USING statements

LPRINT and LPRINT USING send data to the printer.

LPRINT [*expression list* [{, | ;}]]
LPRINT USING *format string; expression list* [{, | ;}]

expression list is a comma-, semicolon-, or space-delimited series of numeric and/or string expressions. *format string* contains formatting information.

LPRINT and LPRINT USING function identically to the PRINT and PRINT USING statements except that the data is sent to the printer rather than to the screen. See the entries for PRINT and PRINT USING for further information.

Portfolio PowerBASIC inserts a carriage-return/linefeed (CR/LF) pair at the end of each line that it prints. The line width (the number of characters output before each CR/LF) is 80 by default.
If the printer is out of paper or off-line when LPRINT is executed, your program will wait approximately 12 seconds for the condition to be corrected. If it is not corrected within this time, the program will generate an appropriate error. You can change the duration by

using POKE to alter the byte at memory segment 0, offset 1144 (decimal) in the BIOS:

```
DEF SEG = 0
POKE 1144, NewDuration%
```

The stored value is the *approximate* number of seconds which the BIOS will wait for the condition to be corrected before reporting an error.

See also: LPOS, PRINT, PRINT USING, TAB

LSET statement

LSET moves string data into a random-access file buffer.

LSET *field variable = string expression*

LSET and its sister function RSET both move string data into "field variables" that have been defined in a previous FIELD statement as belonging to the buffer of a random-access file.

If the length of *string expression* is less than the size of *field variable* as specified in a FIELD statement, LSET left-justifies *string expression* within the field by padding it with spaces. This means that spaces are inserted after the last character of *string expression* so that after the LSET operation, LEN(*field variable*) still equals the width defined in the associated FIELD statement.

If, on the other hand, the length of *string expression* is greater than the size of *field variable* as specified in a

FIELD statement, then *string expression* is truncated to the FIELD length.

RSET works similarly, but performs right justification.

See also: FIELD, GET, PUT, RSET

MID$ function

MID$ returns a portion of a string.

***s$* = MID$(*string expression, start* [*, length*])**

start and *length* are numeric variables or expressions, and can range from 1 to the maximum string size (32750) and 0 to the maximum string size, respectively.

MID$ as a function returns a substring of *string expression* that is *length* characters long and starts at the *start* character of *string expression*. For example,

MID$("Jean-Luc Picard",1,8)

returns *Jean-Luc*.

If *length* is omitted, or there are fewer than *length* characters to the right of the *start* character of *string expression*, all remaining characters of *string expression*, including the *start* character, are returned. If *start* is greater than the length of *string expression*, MID$ returns a null string. Thus,

MID$("Jean-Luc Picard",6) returns *Luc Picard*
MID$("Jean-Luc Picard",6,20) returns *Luc Picard*

MID$("Jean-Luc Picard",20) returns the null string

See also: INSTR, LEFT$, MID$ statement, RIGHT$

MID$ statement

MID$ replaces characters in a string with characters from another string.

MID$(*string variable, start* [*, length*]) = *replacement string*

start and *length* are numeric variables or expressions, and can range from 1 to the maximum string size (32750) and 0 to the maximum string size, respectively.

As a statement, MID$ replaces *length* characters of *string variable*, beginning at character position *start*, with the contents of *replacement string*. If *length* is included, it determines how many characters of *replacement string* are inserted into *string variable*. If *length* is omitted, all of *replacement string* is used. The replacement will never extend past the end of the original *string variable*; that is, MID$ never alters the length of a string. MID$ is case-sensitive.

See also: INSTR, MID$ function

MKDIR statement

MKDIR creates a subdirectory (like the DOS MKDIR command).

MKDIR ***path***

path is a string expression naming the directory to be created.

MKDIR (make directory) creates the subdirectory specified by *path*. If you try to create a directory that already exists, run-time error 5 occurs, "Illegal function call."

See also: CHDIR, RMDIR

MKI$, MKS$, and MKD$ functions

These functions convert numeric data into strings for random-access file output.

***DataTypeString$*** **= MKI$(*integer expression*)**
***DataTypeString$*** **= MKS$(*single-precision expression*)**
***DataTypeString$*** **= MKD$(*double-precision expression*)**

These functions are part of the process of saving numeric values in random-access files. The statements that place information in the buffer of a random-access file (FIELD, LSET, and RSET) operate only on strings, so

numeric data must be translated into string form before it can be PUT into a random-access file.

Command	Converts To	From
MKI$	2-byte string	Integer
MKS$	4-byte string	Single-precision
MKD$	8-byte string	Double-precision

The complementary functions CVI, CVS, and CVD convert the strings back to numeric form when reading random-access files.

Don't confuse MKI$ and related functions with STR$ and VAL, which turn a numeric expression into printable form and vice versa:

```
i = 123.45
a$ = STR$(i) : b$ = MKS$(i)
' a$ contains something worth putting onscreen
' b$ doesn't
PRINT a$, b$
```

See also: CVI and associated functions, FIELD, GET, LSET, PUT, RSET

NAME statement

NAME renames a file (like the DOS **REN** command).

NAME ***filespec1*** **AS** ***filespec2***

filespec1 and *filespec2* are string expressions conforming to DOS path and file-naming conventions. NAME gives

the file represented by *filespec1* the name *filespec2*. Since *filespec2* can contain a path name, it's possible to move the data from one directory to another. You can't move from one disk to another.

If *filespec1* does not exist, run-time error 53 ("File not found") occurs. If *filespec2* already exists, run-time error 75 ("Path/File access error") occurs.

OCT$ function

OCT$ returns a string that is the octal (base 8) representation of its argument.

s$ = OCT$(*numeric expression*)

numeric expression must be in the range –32768 to +65535; if outside this range, an Error 6, "Overflow," is generated. Any fractional part of *numeric expression* is rounded before the string is created.

Octal is a number system that uses base 8 rather than the base 10 (used by the everyday decimal system). OCT$ returns the two's complement form of a negative argument. See the BIN$ function entry and page 52 for information about two's complement arithmetic.

See also: BIN$, HEX$

ON ERROR statement

ON ERROR specifies an error-handling routine and enables or disables error trapping.

ON ERROR GOTO {*label* | *line number*}

label or *line number* identifies the first line of the error-trapping routine. Once error handling has been turned on with this statement, instead of displaying an error message and terminating execution, all run-time errors result in a jump to your error-handling code. Use the RESUME statement to continue execution, or END to terminate the program.

To disable error trapping, use ON ERROR GOTO 0. You can use this technique if an error occurs for which you have not defined a recovery path; you can also choose to display the contents of ERL and ERR at this time.

See also: END, ERL, ERR, ERROR, RESUME

ON/GOSUB statement

ON/GOSUB calls one of several subroutines according to the value of a numeric expression.

ON *n* GOSUB {*label* | *line number*} [, {*label* | *line number*}]...

n is an integer expression with a maximum value of 255, and each *label* or *line number* identifies a statement to branch to. When this statement is encountered, the

*n*th label in the list is branched to; for example, if *n* equals 4, the fourth label in the list receives control. If *n* is less than one or greater than the number of labels, no branch occurs, and *Portfolio PowerBASIC* continues execution with the statement immediately following the ON/GOSUB statement. If *n* is a floating-point value, it is rounded to an integer before a corresponding branch is selected.

Each subroutine should end with RETURN, which causes execution to resume with the statement immediately following the ON/GOSUB statement.

The IF block statement also performs multiple branching and may be more flexible than ON/GOSUB, depending on your application.

See also: GOSUB, IF block, ON/GOTO

ON/GOTO statement

ON/GOTO sends program flow to one of several possible destinations based on the value of a numeric expression.

ON *n* GOTO {*label* | *line number*} [, {*label* | *line number*}]...

n is an integer expression with a maximum value of 255, and *label* or *line number* identifies a statement in the program to branch to. The *n*th label is branched to; for example, if *n* equals 4, the fourth label in the list receives control. If *n* is less than one or greater than the

number of labels in the list, *Portfolio PowerBASIC* continues execution with the statement that immediately follows the ON/GOTO. If *n* is a floating-point value, it is rounded to an integer before a corresponding branch is selected.

ON/GOTO behaves exactly like ON/GOSUB except that it performs a GOTO rather than a GOSUB. This means that the program retains no memory of where the branch originated.

The IF block statement also performs multiple branching and may be more flexible than ON/GOTO, depending on your application. See the GOTO entry for a discussion of ways to avoid using GOTOs in your programs.

See also: GOTO, IF block, ON/GOSUB

OPEN statement

OPEN prepares a file for reading or writing.

OPEN *filespec* [FOR *mode*] AS [#]*filenum*[LEN = *record size*]

OPEN *modestring*, [#]*filenum*, *filespec* [,*record size*]

Each *mode* specifies a particular kind of file (sequential, random-access, or binary) for reading, writing (or both), or appending.

mode	File type	Action
OUTPUT	Sequential	Write to
INPUT	Sequential	Read from
APPEND	Sequential	Append to
RANDOM	Random	Reading or writing
BINARY	Binary	Reading or writing

modestring is a string expression whose first (and usually only) character is one of the following:

modestring	Specific mode
"O"	Sequential output
"I"	Sequential input
"A"	Sequential append
"R"	Random input/output
"B"	Binary input/output

filenum can be any integer value. *filespec* is a string expression specifying the name of the file to be opened and, optionally, a drive and/or path specification. To open a serial communications port as a file, use the special filename "COM1:" (see OPEN COM below). *record size* is an integer expression ranging from 1 to 32767, specifying the length in bytes of each record in a random-access file. The default record size is 128 bytes.

The main function of OPEN is to associate a number (*filenum*) with a file and to prepare that file for reading and/or writing. This number is then used, rather than

its name, in every statement that refers to the file. The OPEN statement contains information on the mode of the file; that is, the methods by which the file will be accessed: sequential (for input/output to a new file, or output to an existing file), random-access, and binary. An OPEN statement is usually balanced by a matching CLOSE.

The two forms of the command differ only in the level of verbosity:

OPEN "myfile.dta" FOR OUTPUT AS #1

has the same effect as

OPEN "O",#1,"myfile.dta"

Attempting to OPEN a file for INPUT that doesn't exist causes run-time error 53, "File not found." If you try to open a nonexistent file for OUTPUT, APPEND, RANDOM, or BINARY operations, a new file is automatically created.

See also: CLOSE, OPEN COM

OPEN COM statement

OPEN COM opens and configures a serial communications port.

OPEN "COM1: [*baud*] [,*parity*] [,*data*] [,*stop*] [*options*]" AS [#]*filenum*

baud is an integer constant specifying the communications rate. Valid rates are 75, 110, 150, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, and 57600 (the default is 300).

parity is a single character specifying parity status:

parity	**Specific mode**
S	Space parity (parity bit always 0)
O	Odd parity
M	Mark parity (parity bit always 1)
E	Even parity
N	No parity (ignored on received characters and omitted on transmitted characters)

The default is even parity (*E*).

data is an integer constant from 5 to 8 specifying the number of data bits. The default is 7.

stop is an integer constant from 1 to 2 specifying the number of stop bits. The default is 1 stop bit (for baud rates 75 and 110, it is 2).

filenum is an integer expression specifying the file number through which you access the communications port.

The OPEN COM statement includes an option block that controls status-line handling, parity, and carriage-return/linefeed processing:

options = [,RS] [,CS[msec]] [,DS[msec]] [,CD[msec]] [,LF] [,PE]

RS suppresses the request to send (RTS) line. CS[*msec*] controls clear to send (CTS). DS[*msec*] controls data set ready (DSR). CD[*msec*] controls carrier detect (CD). LF causes a linefeed character to be appended to every carriage-return character. PE turns on parity checking.

The *msec* argument of the CS, DS, and CD options can range from 0 to 65535 and specifies how many milliseconds *Portfolio PowerBASIC* waits for the required line status before returning a Device Timeout error. If it is 0 or omitted, then no line-status checking is performed. The default for *msec* is 1000 for CS and DS, and 0 for CD.

Note that when a communications file is open, the Portfolio is not allowed to power down, as it may disrupt the serial port's programming. This means that both the Portfolio and the serial interface will consume battery power as long as the communications file is open. Therefore to consume precious battery life, it is best not to keep the communications file open during long periods of inactivity.

The PRINT #, INPUT #, and LINE INPUT # statements, as well as the INPUT$ function, are used to transmit and receive data through the communications file. The EOF, LOC, and LOF functions return the status of the communications buffer, while the $COM metastatement determines the size of the buffer. This simple example echoes all data received by the communications port until a key is pressed:

```
' set up a 1K input buffer
$COM 1024
'300 baud, no parity, 8 data bits, no status checks
OPEN "COM1:300,n,8,1,DS,RS,CS,CD" AS #1
PRINT "Press any key to terminate the program..."
' while a key hasn't been pressed
WHILE NOT INSTAT
  ' if there is any input available
  IF LOC(1) > 0 THEN
    ' read any info available in the COM port buffer
    ComPortInput$ = INPUT$(LOC(1), #1)
    ' display input
    PRINT "COM Port input: "; ComPortInput$
  END IF
WEND
CLOSE 1
END
```

See also: $COM, EOF, INPUT$, INPUT #, LINE INPUT #, LOC, LOF, OPEN, PRINT #

OUT statement

OUT writes a byte to a processor I/O port.

OUT ***portno, byte***

OUT sends *byte* to hardware output port *portno*, where *portno* is in the range –32768 to +65535 and *byte* is an integer expression from 0 to 255. OUT is used for controlling various hardware subsystems, such as communications ports, printer ports, and other external

devices. OUT can send a byte to a port even if no device is attached to that port.

Used improperly, OUT can easily crash your system by writing data to ports that are used internally in the Atari Portfolio.

See also: INP

PEEK function

Returns the byte at a specified memory location.

***y%* = PEEK(*address*)**

The PEEK function and complementary POKE statement are low-level methods of accessing individual bytes in memory, something that cannot be done with *Portfolio PowerBASIC*'s defined variables. The data is retrieved from memory starting at offset *address* within the current segment. Be sure to set the current segment with DEF SEG if you want to retrieve the data from somewhere besides the default data segment.

PEEK retrieves a single byte (8 bits) and returns it as an integer with a value from 0 to 255. *address* is a numeric expression from –32768 to +65535, indicating the offset in the current segment where the data retrieval should begin. See page 52 for more about positive and negative representations of the offset value.

See also: DEF SEG, POKE

POINT function

POINT returns information about a pixel on the graphics screen.

c% = POINT(*x,y*)

(*x,y*) specifies the coordinates of a pixel on the graphics screen. *x* must be a numeric expression which evaluates to a value between 0 and 239. *y* must be a numeric expression which evaluates to a value between 0 and 63. POINT returns a value of 1 if the pixel is turned on (with a previous CIRCLE, LINE, or PSET), or a value of 0 if the pixel is turned off.

See also: CIRCLE, LINE, PSET, SCREEN

POKE statement

Stores a byte to a specified memory location.

POKE *address, byte value*

The POKE statement and complementary PEEK function are low-level methods of accessing individual bytes in memory, something that cannot be done with *Portfolio PowerBASIC*'s defined variables. The data is stored to memory starting at offset *address* within the current segment. Be sure to set the current segment with DEF SEG if you want to store the data to somewhere besides the default data segment.

POKE stores a single *byte* (8 bits) whose value is from 0 to 255. *address* is a numeric expression from –32768 to +65535, indicating the offset in the current segment where the data storage should begin. See page 52 for more about positive and negative representations of the offset value.

See also: DEF SEG, PEEK

POS function

POS returns the horizontal position (column number) of the screen cursor.

y = POS(*x*)

x is a dummy numeric argument. The value returned by POS ranges from 1 to 40; it represents the horizontal position (column number) of the cursor.

Use CSRLIN to get the cursor's vertical position (row number). Use LOCATE to move and hide the cursor.

See also: CSRLIN, LOCATE, LPOS, TAB

PRINT statement

PRINT displays information on the screen.

PRINT [*expression* [{, | |;} [*expression*]]...]

expression is a series of numeric and/or string expressions separated by semicolons, blanks, or commas. If *expression* doesn't end with a semicolon, *Portfolio PowerBASIC* outputs a carriage return after the information in expression list. If you omit *expression*, PRINT outputs only the carriage return.

The punctuation separating each *expression* determines the spacing between the individual items. For quick and tidy output, *Portfolio PowerBASIC* divides the screen into print zones of 14 columns each. Using a comma between each *expression* causes each to be printed at the beginning of the next zone. For example,

```
PRINT "Peter", "Paul", "Mary"
```

results in

```
Peter          Paul           Mary
```

A semicolon or space between elements causes each item to be printed immediately after the last, without regard for print zones. For example,

```
PRINT "Peter";"Paul";"Mary"
```

results in

```
PeterPaulMary
```


Normally, the last thing a PRINT statement sends to the screen is a carriage return, which moves the cursor to the beginning of the next line (where subsequent output appears). However, if PRINT is terminated by a comma, semicolon, or the TAB function, the carriage return is not sent and the cursor remains on the same line, one space to the right of the output. Thus,

```
PRINT "Peter, Paul ";
PRINT "and Mary"
```

outputs

```
Peter, Paul, and Mary
```

while

```
PRINT "Peter, Paul "
PRINT "and Mary"
```

outputs

```
Peter, Paul
and Mary
```

Numeric values are always PRINTed followed by a single space. Positive numbers are preceded by a space, negative numbers by a minus sign.

By default, PRINT displays numeric values with up to 6 or 7 significant digits. Therefore, all digits of integers and single-precision floating point numbers are printed, while double-precision floating point numbers are rounded off. This is done to conserve screen space, given the 40-column × 8-line nature of the Portfolio's display. If you wish to display all digits of a floating

point number with more than 7 significant digits, use the STR$ function with PRINT:

```
PRINT SIN(2)          'prints .9092974
PRINT STR$(SIN(2),12) 'prints .909297426826
PRINT STR$(SIN(2),16) 'prints .9092974268256816
```

PRINT can be abbreviated as a question mark (a typing aid of dubious merit dating from interpretive days of yore):

```
? "Hello"
```

is the same as

```
PRINT "Hello"
```

See also: LPRINT, LPRINT USING, PRINT #, PRINT USING, STR$, TAB

PRINT USING statement

PRINT USING sends formatted information to the screen.

PRINT USING ***format string; expression*** **[{, | ;} [*expression*]]...**

format string is a string constant or variable that describes how to format the information in *expression*. *expression* is the string or numeric information to be printed. Each *expression* is separated by commas, spaces, or semicolons (PRINT USING ignores the punctuation in *expression*).

Formatting String Output

To print the first *n* characters of a string, use a format string consisting of two backslashes (\\) enclosing *n*-2 spaces. For example, if the format string is "\\" (no spaces, length equal to 2), then two characters are printed; for "\ \" (two spaces, length equal to 4), four characters are printed:

```
a$ = "dogs and cats"
PRINT USING "\\"; a$
PRINT USING "\  \"; a$

do
dogs
```

To print only the first character of a string, use a format string of "!":

```
a$ = "dogs and cats"
PRINT USING "!"; a$
```

This code's output:

```
d
```

To print all of a string, use an ampersand (&) as the format string:

```
a$ = "dogs and cats"
PRINT USING "&"; a$
```

This code's output:

```
dogs and cats
```

Formatting Numeric Output

With numeric output, the format string controls the total width of the output field, the number of digits output, and the placement of the decimal point. You can also specify placement of special characters, such as dollar signs, commas, and plus or minus signs.

Unsigned Format

Pound signs (#) in the format string, up to a maximum of 16, represent the digits of a number, with the decimal point placed as specified. Extra spaces to the right of the decimal point are filled with zeros, while extra spaces to the left are padded with spaces. The one exception to this rule is for $1 > n > -1$, in which case 0 is placed before the decimal point. Numbers are rounded off, if necessary, to fit in the allotted space. Negative numbers are displayed with a leading minus sign, which occupies one of the spaces specified by a "#" in the format string.

Value	Format string	Output
0.468	##.##	0.47 (1 leading space)
0.468	#.####	0.4680 (no leading spaces)
12.5	##.##	12.50 (no leading spaces)
12.5	####.#	12.5 (2 leading spaces)

Signed Format

A plus sign at the beginning of the format string causes the number's sign (+ or –) to be printed before the

number. A minus sign at the beginning of the format string does not have this effect; it is treated as a literal character and will always be displayed, regardless of the sign of the number. If you place a plus sign at the end of the format string, the number's sign is printed immediately following the number. A minus sign at the end of the format string causes a trailing space to be printed if the number is positive, or a trailing minus sign if the number is negative.

Value	Format string	Output
24	+##.##	+24.00 (no leading spaces)
24	##.##+	24.00+ (no leading spaces)
99	##.##-	99.00 (1 trailing space)
–99	##.##-	99.00- (no trailing spaces)

A dollar sign at the beginning of the format string causes a literal dollar sign to be printed before the number, with a minus sign inserted between the dollar sign and first digit for negative numbers. A double dollar sign reserves two extra digit positions and causes a single dollar sign (which counts as one of the digits) to be printed to the immediate left of the number; the minus sign would be printed to the left of the dollar sign if the number were negative. A double asterisk results in leading blanks in the field being filled with asterisks. Note that the double asterisk and dollar sign can be combined. A comma to the left of the decimal point in the format string causes commas to be

printed as thousands separators (for example, 1,000,000 as opposed to 1000000).

Value	Format string	Output
5.69	$##.##	$ 5.69 (no leading spaces)
5.69	$$##.##	$5.69 (2 leading spaces)
25.69	**####.##	****25.69 (no leading spaces)
25.69	**$####.##	****$25.69 (no leading spaces)
15000	#####,.##	15,000.00 (no leading spaces)

Scientific Notation

Numbers can be output in scientific notation by including three to six carets (^) in the formatting string. Each caret corresponds to a place in the exponent: one for the *E*, one for the sign, and one to four for the exponent's digits:

```
PRINT USING "#.###^^^^"; 4567      'prints .457E+04
PRINT USING "#.##^^^^^"; 1000000 'prints .10E+007
```

Literal Format Characters

To output a formatting character as itself, precede it with an underscore:

```
PRINT USING "_##"; 1   'prints #1
PRINT USING "#_#"; 1   'prints 1#
```

If the number cannot fit in the space designated by the format string, *Portfolio PowerBASIC* ignores the format

string and prints the entire number with a leading percent sign (%):

```
PRINT USING "##.##"; 125000   'prints %125000.00
```

Variables can be used as format strings. Thus,

```
FS$ = "###.###"
PRINT USING FS$; amount
```

is equivalent to

```
PRINT USING "###.###"; amount
```

See also: LPRINT, LPRINT USING, PRINT, PRINT #, STR$, TAB

PRINT # and PRINT # USING statements

PRINT # and PRINT # USING write formatted information to a file or device.

PRINT #*filenum*, [[USING *format string;*] expression [{, | ;} [*expression*]]...]

filenum is the value under which the file or device was opened. *format string* is an optional sequence of formatting characters (described in the PRINT USING entry above). *expression* is a numeric or string expression to be output to the file.

PRINT # sends data to a file exactly like PRINT sends it to the screen. If you are not careful, you can waste a lot of disk space with unnecessary spaces, or worse, put fields so close together that you can't tell them apart when they are later input with INPUT #. For example,

```
PRINT #1,1,2,3
```

sends

```
1             2             3
```

to file #1. Because of the 14-column print zones between characters, superfluous spaces are sent to the file. On the other hand,

```
PRINT #1,1;2;3
```

sends

```
1 2 3
```

to the file, and you can't read the separate numeric values from this record because INPUT # requires commas as delimiters. The surest way to delimit fields is to put a comma between each field, like so:

```
PRINT #1, 1 "," 2 "," 3
```

which writes

```
1,2,3
```

to the file, a packet that wastes the least possible space and is easy to read with an INPUT # statement.

PRINT # is advantageous when writing a single number or string on each line in a file. Use PRINT # followed by a comma but no arguments to write a blank line (carriage return/linefeed) to a file:

```
PRINT #1,      'writes a blank line to file #1
```

See also: INPUT #, LINE INPUT #, PRINT

PSET statement

PSET plots or erases a pixel on the graphics screen.

PSET (*x,y*) [,*color*]

(*x,y*) specifies the coordinates of the pixel on the graphics screen to set or reset. *x* must be a numeric expression which evaluates to a value between 0 and 239. *y* must be a numeric expression which evaluates to a value between 0 and 63. *color* must be a numeric expression which evaluates to either 0 or 1. Using 1 for *color* causes the pixel to be set, while 0 causes it to be reset (erased).

See also: CIRCLE, LINE, POINT, SCREEN

PUT statement

PUT writes a record to a random-access file.

PUT [#]*filenum* [, *recnum*]

filenum is the file number specified when the file was opened. *recnum* is a numeric expression specifying the record to be written. PUT is complementary to GET; it writes one record to a random-access file. *recnum* is optional. If it is omitted, *Portfolio PowerBASIC* uses the value used in the last PUT or GET statement plus 1.

It is possible to PUT to records out of contiguous order, as in

```
PUT #1, 1
PUT #1, 100
```

which creates a random-access file 100 records long. The data in records 2 through 99, however, are undefined until you explicitly PUT something there.

See also: EOF, FIELD, GET, LOC, LOF, LSET, RSET

PUT$ statement

PUT$ writes a string to a binary mode file.

PUT$ [#]*filenum, string expression*

PUT$ writes the contents of *string expression* to file *filenum* at the file's current pointer position. File *filenum* must have been opened in binary mode.

If the file pointer position is at the end of the file, PUT$ appends *string expression* to the file, increasing its length by LEN(*string expression*) bytes. If the file pointer is before the end of the file, PUT$ overwrites existing

data with *string expression*. Use LOC, LOF, and SEEK to manipulate the file pointer. In either case, the file pointer position following a PUT$ is at the end of the just-written string.

See also: EOF, GET$, LOC, LOF, OPEN, SEEK

RANDOMIZE statement

RANDOMIZE seeds the random number generator.

RANDOMIZE *numeric expression*

Values returned by the random number generator (RND) depend on the initial seed value. For a given seed value, RND always returns the same sequence of values. Thus, any program that depends on RND will run exactly the same way each time unless a different seed is given.

See also: RND

READ statement

READ loads DATA statement constants into program variables.

READ *variable* [, *variable*]...

variable is a numeric or string variable. READ loads the indicated variable(s) with values found in DATA state-

ments. At run time, READ accesses DATA statements in the order in which they appear in the source program and individual constants from left to right within each DATA statement.

The most common error associated with reading DATA statements is improper synchrony between READ statements and DATA constants that causes the program to try to load string data into a numeric variable; this generates a syntax error (run-time error 2). Loading numeric constants into string variables does not cause an error.

The RESTORE statement lets you reset the READ pointer so that constants can be reread from the first statement or any specified DATA statement. If you try to READ more times than your program has constants in DATA statements, you'll get run-time error 4, "Out of data." No error is generated, however, if some DATA constants go unread.

See also: DATA, RESTORE

REG function and statement

REG sets or returns a value in the register buffer.

Function: **y = REG(*register*)**

Statement: **REG *register, value***

register indicates a processor register:

Number	Register
0	Flags
1	AX
2	BX
3	CX
4	DX
5	SI
6	DI
7	BP
8	DS
9	ES

If *register* is not in the range 0 through 9, run-time error 5 is generated, "Illegal function call." *value* is a numeric variable or expression in the range –32768 to +65535. See page 52 for more about positive and negative representations of the 16-bit value.

REG as a function returns the value of the selected element in the register buffer. REG as a statement causes the selected element in the register buffer to be loaded with the indicated integer value; register 0, the Flags register, may not be loaded, however. The register buffer is *Portfolio PowerBASIC*'s interface with the hardware registers in your computer's central processing unit.

Use REG to pass information to and from assembly language interrupt routines. The register buffer is loaded into the processor's registers just before a CALL

INTERRUPT is performed; the contents of the processor's registers are put in the buffer upon returning from the interrupt. At any time, the buffer contains the state of the processor's registers as they existed at the completion of the most recent CALL INTERRUPT routine.

Restrictions: CALL INTERRUPT is the only operation which transfers data between the register buffer and the actual processor registers. REG does not reflect the state of the actual processor registers at any other time.

See also: CALL INTERRUPT

REM statement

REM indicates that the rest of a line in a source code file is to be interpreted as a remark or comment. It won't be processed by the compiler.

REM ***comment***

comment is any sequence of characters. A comment can appear on a line with other statements, but it must be the last thing on that line, and it must be preceded by a colon. For example, the assignment below won't be compiled or executed:

```
REM now add the numbers : a = b + c
```

because the compiler can't tell where the comment ends and the statement begins. This works:

```
a = b + c : REM now add the numbers
```


The single quotation mark (') is an alternate form of REM. When you use a single quote, you don't need a colon to separate the remark from the other statements on the same line. Don't use a single quote to delimit comments after a DATA statement; use **:REM** or **:'** instead (see the example).

RESTORE statement

RESTORE resets the READ pointer.

RESTORE [*label* **|** *line number***]**

RESTORE *label* or *line number* causes the next READ statement to access the DATA statement identified by *label*. If *label* is omitted, the next READ statement accesses the program's first DATA statement. RESTORE enables your program to read DATA constants more than once.

See also: DATA, READ

RESUME statement

RESUME restarts program execution after error handling with ON ERROR GOTO.

RESUME {*label* **|** *line number***}**

The RESUME statement is used to continue execution of a program after a run-time error has been trapped and processed with an ON ERROR handler. The handler must terminate with either RESUME or END, or else a "Missing Resume" error occurs.

RESUME causes execution to resume at the statement identified by *label* or *line number*.

If a RESUME statement is encountered when the program isn't in an error-trapping routine, run-time error 20 results ("Resume without error").

See also: ERL, ERR

RETURN statement

RETURNs from a subroutine to its caller.

RETURN

RETURN terminates the execution of a subroutine and passes control to the statement directly following the calling GOSUB. Performing a RETURN without a corresponding GOSUB causes unexpected and difficult-to-track errors.

See also: $STACK, CALL, GOSUB, SUB

RIGHT$ function

RIGHT$ returns the rightmost *n* characters of a string.

***s$* = RIGHT$(*string expression, n*)**

n is an integer expression specifying the number of characters in *string expression* to be returned; it must be in the range 0 to the maximum string size (32750).

RIGHT$ returns the indicated number of characters from its string argument, starting from the right and working left. If *n* is greater than the length of *string expression*, all of *string expression* is returned. If *n* is 0, RIGHT$ returns the null string.

See also: INSTR, LEFT$, MID$

RMDIR statement

RMDIR deletes a disk directory (like the DOS **RMDIR** command).

RMDIR *path*

path is a standard path description string. RMDIR deletes the directory indicated by *path*. This statement is the equivalent of the DOS **RMDIR** command. The same restrictions apply, namely that *path* must specify a valid, empty directory. If the directory isn't empty, then run-time error 75 occurs, "Path/File access error."

See also: CHDIR, MKDIR

RND function

RND returns a random number.

y = RND

RND returns a random double-precision value between 0 and 1.

Numbers generated by RND aren't really random, but are the result of applying a pseudorandom transformation algorithm to a starting, or seed, value. Given the same seed, *Portfolio PowerBASIC*'s RND algorithm always produces the same chain of "random" numbers.

To produce random integers between 1 and *n*, inclusive, use this technique:

```
RandomNo% = INT(RND * n) + 1
```

Better yet, create it as a single-line DEF FN function:

```
DEF FNRndInt%(x%) = INT(RND * x%) + 1
```

See the discussion under RANDOMIZE for information on seeding the random number generator.

See also: RANDOMIZE

RSET statement

RSET moves string data into a random-access file buffer.

RSET *field variable = string expression*

RSET and its sister function LSET both move string data into "field variables" which have been defined in a previous FIELD statement as belonging to the buffer of a random-access file.

If the length of *string expression* is less than the size of *field variable* as specified in a FIELD statement, RSET right-justifies *string expression* within the field by padding it with spaces. This means that spaces are inserted before the first character of *string expression* so that after the RSET operation, LEN(*field variable*) still equals the width defined in the associated FIELD statement.

If, on the other hand, the length of *string expression* is greater than the size of *field variable* as specified in a FIELD statement, then *string expression* is truncated to the FIELD length.

LSET works similarly, but performs left-justification.

See also: FIELD, GET, LSET, PUT

SCREEN function

SCREEN returns the ASCII code of the character at the specified screen row and column.

y = SCREEN(*row, column*)

row and *column* are integer expressions that specify the screen row and column that SCREEN returns information about; *row* and *column* can range from 1 to 8 and 1

to 40, respectively. SCREEN may only be used in text mode.

SCREEN statement

SCREEN sets the screen display mode.

SCREEN ***mode***

mode specifies either text (default) or graphics screen mode; it must be a numeric expression which evaluates to a value of either 0 (text mode) or 1 (graphics mode). When a program begins executing, and when it terminates, the display is automatically set to text mode. Whenever the screen switches from text to graphics mode or vice versa, the screen is also cleared. You can display text while in graphics mode by using the PRINT statement just as you would in text mode.

SEEK statement

SEEK sets the file pointer position in a binary file.

SEEK [#] ***filenum, position***

SEEK sets the file pointer position of file *filenum* to *position*. This means that the next GET$ or PUT$ performed on the file will occur *position* bytes deep into the file. File *filenum* must have been opened in binary mode.

Use LOC to determine a binary file's current pointer position, and LOF to determine its length. SEEKing past the end of a file does not produce an error, but no data can be read from there.

See also: EOF, GET$, LOC, LOF, OPEN, PUT$

SGN function

SGN returns the sign of a numeric expression.

***y* = SGN(*numeric expression*)**

If *numeric expression* is positive, SGN returns +1. If *numeric expression* is zero, SGN returns 0. If *numeric expression* is negative, SGN returns –1.

In conjunction with the ON/GOTO and ON/GOSUB statements, SGN can produce a FORTRAN-like three-way branch:

```
ON SGN(balance) + 2 GOTO InTheRed, Even,_
InTheMoney
```

See also: IF, ON/GOSUB, ON/GOTO

SIN function

SIN returns the trigonometric sine of its argument.

***y* = SIN(*numeric expression*)**

numeric expression is an angle specified in radians. To convert radians to degrees, multiply by 57.2958. To convert degrees to radians, multiply by 0.017453. For more information on radians, see the ATN function entry.

SIN returns a double-precision value between –1 and +1.

See also: ATN, COS, TAN

SQR function

SQR returns the square root of its argument.

***y* = SQR(*numeric expression*)**

numeric expression must be greater than or equal to zero. SQR calculates square roots using a faster algorithm than the power-of-0.5 method; that is, y = SQR(x) takes less time to execute than $y = x\text{^}.5$).

Attempting to take the square root of a negative number results in run-time error 5, "Illegal function call." SQR returns a double-precision result.

See also: EXP, LOG

STR$ function

STR$ returns the string representation of a number.

***s$* = STR$(*numeric expression* [, *digits*])**

STR$ returns the string form of a numeric variable or expression; that is, it returns a string comprised of the ASCII characters that you would see on the screen were you to PRINT *numeric expression*. *digits* is an optional integer expression specifying the number of digits to appear in the result. If *numeric expression* is greater than zero, STR$ adds a leading space. For example, STR$(14) returns a three-character string, of which the first character is a space, and the second and third are the ASCII characters "1" and "4".

digits permits control over the format of the result. *numeric expression* is rounded, if necessary, to fit in *digits* places. If *numeric expression* cannot be rounded to fit in *digits* places, an error occurs. Allowable values for *digits* are 1 through 16.

The complementary function is VAL, which takes a string argument and returns the numeric equivalent. Thus, **number = VAL(STR$(number))**.

See also: PRINT, PRINT USING, VAL

STRING$ function

STRING$ returns a string consisting of multiple copies of the specified character.

***s$* = STRING$(*Count*, {*code* | *string expression*})**

Count and *code* are integer expressions. *Count* can range from 1 to the maximum string size (32750); *code*, from 0 to 255.

STRING$ with a numeric argument returns a string of *Count* copies of the character with ASCII code *code*.

STRING$ with a string argument returns a string of *Count* copies of *string expression*'s first character. For example,

```
STRING$(8,32)
STRING$(8," ")
SPACE$(8)
REPEAT$(8," ")
```

all do the same thing—produce a string of eight spaces.

STRPTR function

STRPTR returns the offset portion of the address of a string variable.

***x!* = STRPTR(*string variable*)**

string variable is the name of a string variable. STRPTR returns the offset portion of the address in memory where the contents of *string variable* are stored. Such address information is sometimes called a *pointer;* for example, STRPTR(*a$*) is said to return a pointer to *a$*.

A segment value is also required to fully define an address. The function STRSEG returns the segment portion of the address of a string variable.

Note that STRPTR differs from VARPTR (and STRSEG from VARSEG). When used with a string variable, VARPTR returns the offset of the string's *handle*, while STRPTR returns the offset of the actual string *data*.

STRPTR returns a floating point value in the range 0..65535. See page 52 for an example showing how to convert this value to a signed integer in the range –32768..+32767.

See also: STRSEG, VARPTR, VARSEG

STRSEG function

STRSEG returns the segment portion of the address of a string variable.

***x!* = STRSEG(*string variable*)**

string variable is the name of a string variable. STRSEG returns the segment portion of the address in memory where the contents of *string variable* are stored. An

offset value is also required to fully define an address; the STRPTR function returns the offset portion of the string's address.

Note that STRSEG differs from VARSEG (and STRPTR from VARPTR). When used with a string variable, VARSEG returns the segment of the string's *handle*, while STRSEG returns the segment of the actual string *data*.

STRSEG returns a floating point value in the range 0..65535. See page 52 for an example showing how to convert this value to a signed integer in the range –32768..+32767.

See also: STRPTR, VARPTR, VARSEG

SUB/END SUB statements

SUB/END SUB defines a *Portfolio PowerBASIC* procedure.

SUB *procname* [(*parameter list*)] SHARED
.
. {statements}
.
[EXIT SUB]
.
. {statements}
.
END SUB

procname is the name of the procedure. *It must be unique*: no other variable, function, procedure, subroutine, or label can share it.

parameter list is an optional, comma-delimited sequence of formal parameters. The parameters used in the argument list serve only to define the procedure; they have no relationship to other variables in the program with the same name. Note that all parameters must be scalar (non-array) variables; you cannot pass constants, expressions, individual array elements, or whole arrays to a procedure directly.

SUB and END SUB define a subroutine-like block of statements called a procedure (or subprogram), which is invoked with the CALL statement and can be passed parameters by reference.

The default variable type within the SUB body is SHARED, denoted by the presence of the SHARED keyword in the procedure declaration. Shared variables are global to your entire program; if a SUB modifies a variable called *MyVar*, later references to *MyVar* in another SUB or in the main body of your program will access the new value of *MyVar*. For example:

```
MyVar = 1
CALL BumpIt
CALL BumpIt
PRINT MyVar          'prints 3
END

SUB BumpIt SHARED
  MyVar = MyVar + 1
```

```
END SUB
```

Procedure definitions and program flow

The position of procedure definitions is immaterial. They can be located anywhere in your source code, although clarity is improved by grouping them together in one region. You need not direct program flow through a procedure as an initialization step—the compiler sees your definitions wherever they might be.

Also, unlike subroutines, execution can't accidentally "fall into" a procedure. As far as the execution path of a program is concerned, function and procedure definitions are invisible. For example,

```
CALL PrintStuff
SUB PrintStuff SHARED
  PRINT "Printed from within PrintStuff"
END SUB
```

When this program is executed, the message is only printed once.

Procedure definitions should be treated as isolated islands of code; don't jump into or out of them with GOTO, GOSUB, or RETURN statements. Within definitions, however, such statements are legal.

Note that you can't nest procedure definitions; that is, you cannot define a procedure within another procedure (although a procedure definition can contain calls to other procedures and functions).

You must terminate a procedure definition with END SUB, which returns control to the statement directly

after the invoking CALL. Use the EXIT SUB statement to return from a procedure definition before reaching the END SUB statement.

See also: CALL, DEF FN, END SUB, EXIT SUB, GOSUB, RETURN

TAB function

TAB moves the printing position to the specified column.

TAB(*n*)

n is an integer expression in the range 1 to 255. TAB can only be used in the expression list of an LPRINT, PRINT, or PRINT # statement. TAB(*n*) moves the print position to the *n*th position on the current line. If the current print position is already past *n* (for example, PRINT TAB(20) with the print position at column 30), then *Portfolio PowerBASIC* skips down to the *n*th position on the next line.

If TAB appears at the end of a PRINT statement's expression list with or without a trailing semicolon, *Portfolio PowerBASIC* does not output a carriage return; that is, there is an implied semicolon after TAB.

See also: LPOS, LPRINT, PRINT, PRINT #, STRING$

TAN function

TAN returns the trigonometric tangent of its argument.

y = TAN(*numeric expression*)

numeric expression is an angle specified in radians. To convert radians to degrees, multiply by 57.2958. To convert degrees to radians, multiply by 0.017453. For more information on radians, see ATN.

TAN returns a double-precision result.

See also: ATN, COS, SIN

TIME$ system variable

TIME$ is used to read or set the system time.

To read the time: ***s$* = TIME$**

To set the time: **TIME$ = *string expression***

The system variable TIME$ contains an eight-character string that represents the time of the system clock in the form "hh:mm:ss," where *hh* is hours (in 24-hour military form), *mm* is minutes, and *ss* is seconds. TIME$ won't be accurate unless the DOS clock was set correctly when the computer was last reset.

Assigning a string expression to TIME$ resets the system clock. The string expression must contain time information in military (24-hour) format. Minute and second information can be omitted. For example,

TIME$ = "12"	**'12 noon**
TIME$ = "13:01"	**'1:01 PM**
TIME$ = "13:01:30"	**'30 sec past 1:01 PM**
TIME$ = "0:01"	**'1 min past midnight**

If the hour, minutes, or seconds parameter is out of range (for example, a minutes value of 61), run-time error 5 occurs ("Illegal function call").

See also: DATE$

TONE statement

TONE generates a musical note for a specified duration or a series of tones to dial a number on a touch-tone telephone.

TONE *note, duration*
TONE *touchtone$*

note is an integer expression specifying the musical note to be played. A value of 0 (zero) represents silence for the duration specified. The values 1..25 represent the notes from D sharp (D#) in octave 5 through D sharp in octave 7, respectively:

1 – D# in octave 5	14 – E in octave 6
2 – E	15 – F
3 – F	16 – F#
4 – F#	17 – G
5 – G	18 – G#
6 – G#	19 – A
7 – A	20 – A#
8 – A#	21 – B in octave 6
9 – B in octave 5	22 – C in octave 7
10 – C in octave 6	23 – C#
11 – C#	24 – D
12 – D	25 – D# in octave 7
13 – D# in octave 6	

duration is an integer expression which specifies the length of the note as measured in 10 millisecond (mS) intervals.

touchtone$ is a string expression specifying the touch-tone phone number to be generated. It may contain one or more of the following characters: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', '*', '#'.

See also: BEEP

TROFF and TRON statements

TROFF and TRON turn program execution tracing off and on.

TRON
TROFF

TRON puts your program into a debugging mode in which source code line numbers are displayed as each statement is executed; TROFF turns this debugging mode off. The trace output is displayed along with the program output.

UCASE$ function

UCASE$ returns an all-uppercase version of its string argument.

***s$* = UCASE$(*string expression*)**

UCASE$ returns a string equal to *string expression* except that all the lowercase alphabetic characters in *string expression* are converted to uppercase.

See also: LCASE$

VAL function

VAL returns the numeric equivalent of a string.

***y* = VAL(*string expression*)**

VAL turns its string argument into a number. If *string expression* begins with numeric characters (0 to 9, +, –, or .), then VAL returns the number up to the point of the

first nonnumeric character. If *string expression* doesn't begin with a numeric character, VAL returns 0. Leading white-space characters (spaces or tabs) are ignored.

VAL is often used in data entry routines. With it, a program can prompt the user for numeric data in string form, then convert the legal portions of the string to numbers.

See also: INSTR, LEFT$, MID$, RIGHT$, STR$

VARPTR function

VARPTR returns the offset portion of the address of a variable.

y! = **VARPTR(*variable*)**

variable is any numeric or string variable or an element of an array. VARPTR returns the offset portion of the address in memory where the variable is stored. Such address information is sometimes called a *pointer*; for example, VARPTR(x) is said to return a pointer to x.

A segment value is also required to fully define an address. VARSEG returns the segment portion of the address.

When you use VARPTR to get the offset of a string, keep in mind that the value being returned is the offset of the string *handle*, not the actual data in the string. Use

STRPTR and STRSEG to find the address of the string's data.

VARPTR returns a floating point value in the range 0..65535. See page 52 for an example showing how to convert this value to a signed integer in the range –32768..+32767.

See also: DEF SEG, PEEK, POKE, STRPTR, STRSEG, VARSEG

VARSEG function

VARSEG returns the segment portion of the address of a variable.

y! = **VARSEG(*variable*)**

variable is any numeric or string variable or element of an array.

Both a segment and an offset value are required to fully define the address of a variable. VARSEG returns the segment portion of the address, and VARPTR returns the offset portion. Use VARSEG in conjunction with VARPTR to locate a variable in memory.

When you use VARSEG to get the segment of a string, keep in mind that the value being returned is the segment where the string *handle* is located, not the segment where the actual contents of the string are located; use STRPTR and STRSEG to find this address.

VARSEG returns a floating point value in the range 0..65535. See page 52 for an example showing how to convert this value to a signed integer in the range –32768..+32767.

See also: DEF SEG, PEEK, POKE, STRPTR, STRSEG, VARPTR

WHILE/WEND statements

WHILE and WEND define a loop of program statements that is executed repeatedly as long as a certain condition is met.

WHILE ***integer expression***
.
. **{statements}**
.
WEND

If *integer expression* is TRUE (it evaluates to a nonzero value), all of the statements between the WHILE and the terminating WEND are executed. *Portfolio PowerBASIC* then jumps back to the WHILE statement and repeats the test. If it is still TRUE, *Portfolio PowerBASIC* executes the enclosed statements again. This process is repeated until the test expression evaluates to zero, at which time execution passes to the statement following WEND.

If *integer expression* evaluates to FALSE (zero) on the first pass, then none of the statements in the loop are executed.

One common use of the WHILE/WEND construct is to pause until a key is pressed:

```
PRINT "Press any key to continue..."
WHILE NOT INSTAT : WEND
```

Another use is to input data from a file until the end is reached:

```
WHILE NOT EOF(1)
  INPUT #1,X$
WEND
```

Loops built with WHILE/WEND statements can be nested (enclosed within each other). Each WEND matches the most recent unmatched WHILE. If *Portfolio PowerBASIC* encounters a WEND statement without a pending WHILE, run-time error 30 occurs, "WEND without WHILE." A WHILE without a matching WEND generates run-time error 29, "WHILE without WEND."

Although the compiler doesn't care, you should indent the statements between WHILE and WEND by a couple of spaces to clarify the structure of the loop you've constructed.

Note that

WHILE -1
.
.
.
WEND

creates an infinite loop. To exit a WHILE/WEND loop prematurely, use the EXIT LOOP statement.

Portfolio PowerBASIC's DO/LOOP construct offers a more flexible way to build conditional loops.

See also: DO/LOOP, EXIT, FOR/NEXT

APPENDIX A

Error messages

There are two fundamental types of errors in *Portfolio PowerBASIC*: *compile time* and *run time*. Compile-time errors are errors in syntax discovered by the compiler. Run-time errors are anomalies caught at run time by error-detection mechanisms which the compiler places in your executable programs.

Run-time errors

Run-time errors occur when a compiled program is executed. Examples include file-system errors (disk full or write-protected), improper function call arguments, memory errors (usually, not enough), and a host of other problems.

Trapping run-time errors

Run-time errors can be trapped; that is, you can cause a designated error-handling subroutine to get control should an error occur. Use the ON ERROR statement to accomplish this. This routine can "judge" what to do next based on the type of error that occurs. File-system errors (for example, disk full) in particular are well-suited to handling such routines; they are the only errors that a thoroughly debugged program should have to deal with.

The ERROR statement (which simulates run-time errors) can be used to debug your error-handling routines.

If run-time errors are not explicitly trapped by your code, your program will abort upon encountering an error condition. It will then display the error number which occurred. For example:

Error *nnn*

where *nnn* is a three-digit error code. If your program includes line numbers, the number of the most recently executed numbered line will also be displayed as part of the message:

Error *nnn* at line *lllll*

where *nnn* is a three-digit error code and *lllll* is the line number.

Compiler errors

Most compile-time (compiler) errors are errors of syntax, caused by missing symbols, misspelled commands, unbalanced parentheses, and so on. If the compiler finds something in a source program that it cannot understand or permit, compilation is terminated and an error message is displayed with the line number where the error occurred. You can then edit the offending statement and recompile your program.

Run-time errors—listing

2 Syntax error
A run-time syntax error has been created by a READ statement trying to load string data into a numeric variable. Other syntax errors are caught by the compiler at compile-time.

4 Out of data
A READ statement ran out of DATA statement values.

5 Illegal function call
This is a catch-all error related to passing an inappropriate argument to some statement or function. A few of the 101 things that can cause it:

- Trying to perform invalid mathematical operations, such as taking the square root of a negative number.

- A record number is too large (or negative) in a GET or PUT.

6 Overflow
An overflow is the result of a calculation producing a value too large to be represented in the indicated numeric type. For example, ***x%* = 32767 + 1** causes overflow because 32768 can't be represented by an integer.

7 Out of memory
Many different situations can cause this message, including dimensioning too large an array or using up all of string space.

9 Subscript out of range
You attempted to use a subscript larger than the maximum value established when the array was DIMensioned.

11 Division by zero
You attempted to divide by zero or to raise zero to a negative power.

13 Type mismatch
You used a string value where a numeric value was expected or vice versa. This can occur in PRINT USING statements.

14 Out of string space
String storage space is exhausted.

15 String too long
The string produced by a string expression is longer than the maximum string size.

19 No RESUME
Program execution ran to the physical end of the program while in an error-trapping routine. There may be a missing RESUME statement in an error handler.

20 RESUME without error
You executed a RESUME statement without an error occurring; that is, there is no error-handling subroutine to RESUME from.

24 Device time-out
The specified time-out value for a communications status line has expired. Time-out values can be specified for the ClearToSend, CarrierDetect, and DataSetReady status lines. The program should either abort execution or retry the communications operation.

27 Out of paper
The printer interface indicates that the printer is out of paper. The printer can also be turned off or have some other problem.

50 Field overflow
Given the file's record length, you attempted to define too long a set of field variables in a FIELD statement.

51 Internal error
A malfunction occurred within the *Portfolio PowerBASIC* run-time system. Call Atari's Technical Support group with information about your program.

52 Bad file number
The file number you gave in a file statement doesn't match one given in an OPEN statement, or the file number may be out of the range of valid file numbers.

53 File not found
The file name specified could not be found on the indicated drive.

54 Bad file mode
You attempted a PUT or a GET (or PUT$ or GET$) on a sequential file.

55 File already open
You attempted to open a file that was already open, or you tried to delete an open file.

57 Device I/O error
A serious hardware problem occurred when trying to carry out some command.

58 File already exists
The new name argument specified in your NAME statement already exists.

61 Disk full
There isn't enough free space on the indicated or default disk to carry out a file operation. Create some more free disk space and retry your program.

62 Input past end
You tried to read more data from a file than it had to read. Use the EOF (end of file) function to avoid this problem. This error can also be caused by trying to read from a sequential file opened for output or append.

63 Bad record number
A negative number or a number which is too large was specified as the record argument to a random file PUT or GET statement.

64 Bad file name
The file name specified in a KILL or NAME statement contains invalid characters.

67 Too many files
This error can be caused either by trying to create too many files in a drive's root directory, or by an invalid file name that affects the performance of the DOS Create File system call.

68 Device unavailable
You tried to OPEN a device file on a machine without that device; for example, COM1 on a system without a serial communications port.

69 Communications buffer overflow
You executed a statement to INPUT characters into an already full communications buffer. Your program should either check and empty the buffer more often or provide a larger buffer size.

70 Permission denied
You tried to write to a write-protected disk.

71 Disk not ready
The door of a floppy disk drive is open, or there is no disk in the indicated drive.

72 Disk media error
The disk controller indicates a hard media error in one or more sectors.

74 Rename across disks
You can't rename a file across disk drives.

75 Path/File access error
During a command capable of specifying a path name (OPEN, NAME, or MKDIR, for example), you used a path inappropriately; trying to OPEN a subdirectory or to delete a directory in-use, for example.

76 Path not found
The path you specified during a CHDIR, MKDIR, OPEN, etc., can't be found.

201 Out of stack space
You have run out of stack space. You can increase the stack space available with the $STACK metastatement. You can also reduce the amount of recursion being done by recursive procedures and functions in your code.

242 String/array memory corrupt
The string memory area has been improperly over-written. This could be caused by the improper action of an interrupt routine, by string array access outside of the dimensioned limits, by an error within the *Portfolio PowerBASIC* run-time system, or by a stack which is too small. If the stack is too small, increase the stack size with the $STACK metastatement.

Compiler errors—listing

401 Expression too complex
The expression contained too many operators/operands; break it down into two or more simplified expressions.

402 Statement too complex
The statement complexity caused an overflow of the internal compiler buffers; break the statement down into two or more simplified statements.

405 Block nesting overflow
Your program has too many statement block structures nested within each other. *Portfolio PowerBASIC* block structures may be nested 64 levels deep.

406 Compiler out of memory
Available compiler memory for symbol space, buffers, and so on, has been exhausted. Try the following steps:

1. Remove unnecessary line numbers and labels.
2. Shorten your variable and procedure names.
3. Reset your Portfolio in order to unload any memory-resident programs, such as PBRUN (the *Portfolio PowerBASIC* run-time library).

408 Segment exceeds 64K
Your program code exceeds the 64K limitation. Try to reduce the number of lines in your program which actually generate code (removing REMarks won't help, for example).

409 Variables exceed 64K
All variables (except for string data) are limited to 64K total space. String descriptors, as well as all integer, floating-point, and array variables, are included in this space. Try to remove any unused variables or reduce the size of any arrays in your program.

410 "," expected
The statement's syntax requires a comma (,).

411 ";" expected
The statement's syntax requires a semicolon (;).

412 "(" expected
The statement's syntax requires a left parenthesis (().

413 ")" expected
The statement's syntax requires a right parenthesis ()).

414 "=" expected
The statement's syntax requires an equal sign (=).

415 "-" expected
The statement's syntax requires a hyphen (-).

416 Statement expected
A *Portfolio PowerBASIC* statement was expected. Some character could not be identified as a statement, metastatement, or variable.

417 Label/line number expected
A valid label or line-number reference was expected in an IF, GOTO, GOSUB, or ON statement.

418 Numeric expression requires relational operator
The compiler has found a string operand in a position where a numeric operand should be.

419 String expression requires string operand
The compiler expected a string expression and found something else; for example, **X$ = A$ + 3**.

420 Scalar variable expected
The compiler expected a scalar variable in a SUB or DEF FN definition, or as a parameter in a CALL statement. Scalar variables include non-array string variables, integer variables, single-precision floating-point variables, and double-precision floating-point variables.

421 Array variable expected
An array variable was expected in a DIM statement.

422 Numeric variable expected
A numeric variable was expected in a statement or function.

423 String variable expected
A string variable was expected in a FIELD, GET$, PUT$, or LINE INPUT statement.

424 Variable expected
A variable was expected in a VARPTR or VARSEG function.

426 Positive integer constant expected
A positive integer constant was expected in the array bounds for a DIM statement or in the $COM or $STACK metastatements.

428 Numeric scalar variable expected
Either an integer, single-precision floating-point, or double-precision floating-point variable is expected; for example, in a FOR/NEXT loop.

431 End of line expected
No characters are allowed on a line (except for a comment) following a metastatement, END SUB, or a statement label.

432 AS expected
The AS reserved word is missing in either a FIELD or OPEN statement.

433 DEF FN expected
The compiler found an END DEF or EXIT DEF statement without a DEF FN function defined. When defining a DEF FN function, it must begin with a DEF FN statement.

434 IF expected
The compiler found an END IF or an EXIT IF statement without a beginning IF statement defined.

435 DO loop expected
The compiler found a LOOP or EXIT LOOP statement without a beginning DO statement defined.

438 FOR loop expected
The compiler found an EXIT FOR statement without a beginning FOR statement defined.

439 SUB expected
The compiler found an END SUB or EXIT SUB statement without a procedure defined. You must define a procedure by beginning it with a SUB statement.

440 END DEF expected
A DEF FN function wasn't terminated with a corresponding END DEF statement.

441 END IF expected
An IF block wasn't terminated with a corresponding END IF statement.

442 LOOP/WEND expected
A DO or WHILE loop was not terminated with a corresponding LOOP or WEND statement.

444 END SUB expected
A procedure was not properly terminated with an END SUB statement.

445 NEXT expected
A FOR loop was not properly terminated with a NEXT statement.

446 THEN expected
An IF statement is missing its accompanying THEN part.

447 TO expected
A FOR statement is missing its accompanying TO part.

448 GOSUB expected
An ON statement is missing its accompanying GOSUB part.

449 GOTO expected
An ON statement is missing its accompanying GOTO part.

454 Undefined function reference
You used a function name in an expression without defining the DEF FN function. Check the name of the function for mistakes or provide a definition for the function.

455 Undefined SUB procedure reference
You used CALL to a procedure, but you did not define the procedure. Check the name of the procedure for mistakes or provide the procedure.

456 Undefined label/line reference
You used a line number or label in an IF, GOTO, GOSUB, or ON statement, but you did not define the label or line number. Check the label or line number for mistakes or provide a label.

458 Duplicate label/line number
The same label or line number was used twice. Check your program for duplicate labels or line numbers and change them so that they are unique.

460 Duplicate function definition
A DEF FN name was defined more than once in your code. Check your program for duplicate names and change them so that they are unique.

461 Duplicate SUB procedure definition
A SUB name was defined more than once in your code. Check your program for duplicate names and change them so that they are unique.

463 Duplicate variable declaration
Two variables with the same name have been declared. Check your program for duplicate names and change them so that they are unique.

464 Duplicate $COM definition
More than one $COM metastatement was encountered in your program.

466 Duplicate $STACK definition
More than one $STACK metastatement was encountered in your program.

467 Invalid line number
Line numbers must be in the range 0 through 32767.

468 Invalid label
A label in your code contains invalid characters.

469 Metastatements not allowed here
A metastatement must be the first statement on a line.

470 Block/Scanned statements not allowed here
Block statements (like WHILE/WEND and DO/LOOP) are not allowed in single line IF statements. Also, you cannot have a procedure or function definition nested within the body of another definition.

471 Syntax error
Something is incorrect on the line—the compiler could not determine a proper error message.

475 Parameter mismatch
The type or number of parameters does not correspond with the declaration of the function or procedure.

476 CLEAR parameter
The additional parameters available to the CLEAR statement in Interpretive BASIC are not available in *Portfolio PowerBASIC*.

486 Array exceeds 64K
The size of an array cannot exceed 64K (one data segment).

487 Arrays limited to eight dimensions
The maximum number of dimensions that can be specified for an array is eight. This is an internal limit for the compiler.

488 Invalid numeric format
Your program declared a number with more than 16 digits or a floating-point number with a *D* or *E* component without the exponent value.

489 Invalid function/procedure name
A function or procedure has an invalid name. In the case of a DEF FN function, FN must be followed by a letter and then other letters, digits, and periods, optionally ended with a type identifier (%, !, #, or $). In the case of a SUB procedure, the name must begin with a letter and can be followed by other letters, digits, and periods, but may not include a type identifier.

500 CLEAR not allowed here
CLEAR is illegal within a procedure or function.

501 Line too long
A line of program code can contain at most 388 characters, including underscores. Break the line down into two or more shorter lines.

502 Duplicate definition
A program element which should only appear once was duplicated in your code.

503 SHARED expected
A SUB definition was encountered which did not include the keyword SHARED.

601-606 Internal error
If this error occurs, report it immediately to Atari's Technical Support group.

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

ATARI
Portfolio™

PORTFOLIO POWERBASIC®

USER'S MANUAL

ATARI
COMPUTER

4-3-91

C398792-001 Rev. A

ATARI Portfolio™

Portfolio PowerBASIC™

 HPC-705 C302205-001 Rev. A