Apple III

Apple Business BASIC

Reference Manual - Volume 1

Notice

Apple Computer reserves the right to make improvements in the product described in this manual at any time and without notice.

Disclaimer of All Warranties And Liabilities

Apple Computer makes no warranties, either express or implied, with respect to this manual or with respect to the software described in this manual, its quality, performance, merchantability, or fitness for any particular purpose. Apple Computer software is sold or licensed "as is." The entire risk as to its quality and performance is with the buyer. Should the programs prove defective following their purchase, the buyer (and not Apple Computer, its distributor, or its retailer) assumes the entire cost of all necessary servicing, repair, or correction and any incidental or consequential damages. In no event will Apple Computer be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software, even if Apple Computer has been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

10260 Bandley Drive
Cupertino, California 95014
(408) 996-1010

Reorder Apple Product #A3L0002

Apple III

Business BASIC

Reference Manual Volume 1

Contents

Appendices

Preface

This book is intended to provide you with the information you need to learn and use the Apple Business BASIC programming language. If you are new to the world of computers, or are not familiar with one or more other dialects of BASIC, you should first study a good textbook on elementary programming. You should also read the *Apple III Owner's Guide* that is packed with your computer before reading this manual.

The material in this manual is presented primarily for programmers with some experience, but enough examples, simplified procedures, and descriptions are included to enable you to use it effectively even if you are a beginner.

To make the use of this manual easier for you, we have divided it into two volumes. After you have become familiar with the material in the first part, you can quickly refer to details of Business BASIC as found in the reference portion of the manual, which is contained in the second volume. Paging is continuous through the manual, and the index is placed in the reference volume.

When important terms are first introduced in this manual, they are italicized. Three special symbols are used in this book to point out places where you should pay attention:

The pointing finger indicates an especially useful or noteworthy piece of information.

The eye means "watch out!" It indicates a description of a special feature to which you should be alert.

The stop sign means that in the situation described, you should be especially cautious, because if you do something incorrect, BASIC may not be able to recover. You may have to restart BASIC, and this means that you will probably lose your program.

1

Introduction to Apple Business BASIC

1

1

Introduction to Apple Business BASIC

Introduction

Apple Business BASIC is an extended version of the BASIC programming language for the Apple III. Some of its more important features are listed here for readers familiar with other implementations of BASIC:

The facilities for using disk files are built in, so that programs can easily and conveniently read and write files.

The *long integer* variable type provides 19-digit precision.

The PRINT USING and IMAGE statements are powerful tools for controlling the exact format of data displayed or printed by a BASIC program.

Variable names may be up to 64 characters long. Reserved words are recognized only when they are set off either by spaces or characters other than letters or digits. This allows you to use reserved words embedded in variable names without causing problems.

An optional ELSE clause has been added to the IF...THEN statement.

The input/output (I/O) system allows direct control of a wide variety of plug-in peripheral devices, as well as the use of assembly language routines and graphics facilities.

Starting Up

We recommend that you have Apple Business BASIC running while you read this manual, so that you can try out on your computer anything that the manual describes or suggests.

Follow these steps to start up your system:

1. Begin with the system plugged in and the power switch off.

2. Insert the Apple Business BASIC diskette into the disk drive built into the front of your Apple. Hold the diskette in your right hand with the label facing up and your thumb on it. The label should be nearest the edge of the diskette closest to you. Now insert the diskette into the horizontal slot in the drive without rotating your hand. Push it very gently into the slot until it stops.

3. Turn the power switch on. The system will start up, the drive's in use light will come on, and the disk drive will whir and buzz. Carefully close the disk drive door.

4. Soon you will see a prompt line at the top of the screen, giving the version of Business BASIC that you are running, and below it a right parenthesis character [)] at the left margin of the video display screen, with a cursor to its right. Apple Business BASIC is up and running.

5. See your *Apple III Owner's Guide* if you have difficulty starting up your Apple.

   If there is a BASIC program file named "Hello" on the diskette that you use when starting up your Apple (as there is on the Apple Business BASIC diskette), that program will automatically be executed when the system starts. This feature lets you develop "turnkey" systems easily by just naming your working program "Hello."

The right parenthesis is BASIC's *prompt* character. When it appears on the screen, it means that BASIC is waiting for you to type something. The cursor shows you where the next character that you type will appear on the screen.

Entering Statements

Now that you have the prompt character and the cursor on the screen, you are ready to begin using the Apple Business BASIC language. Type

```
)PRINT "Hi there"
```

and press the key marked RETURN. Your Apple will display the words

```
Hi there
)
```

followed by the prompt and the cursor again. If you misspell the reserved word PRINT you will get this error message:

```
?SYNTAX ERROR
```

The word *syntax* refers to the rules for constructing statements in the BASIC language. Sometimes you will enter a statement that is syntactically correct, but does not say quite what you intended it to say. BASIC will execute the statement exactly as you typed it. For example, if you forget either the first quotation mark or both quotation marks, your Apple will display a zero (the value of the variable Hi):

```
0
)
```

The statement

```
)PRINT "Hi there"
```

is an instruction to your Apple telling it to display all of the characters between the quotation marks, in this case a salutation. You can use the PRINT statement to tell your Apple to display any message you wish.

Don't press the RETURN key if you are typing a long statement and getting close to the right margin of the display screen. When you reach the end of the screen line, BASIC automatically breaks the

statement for you and you find yourself typing on the next screen line. Pressing the RETURN key ends the statement, and BASIC will try to understand what you typed, and carry out your instructions.

Immediate and Deferred Execution

The PRINT statement example we tried earlier was *executed* immediately after you pressed the RETURN key. This is called *immediate execution*. Now type

```
)10 PRINT "Hi again"
```

and press RETURN. Nothing happens on the screen except that BASIC again displays its prompt. Now type

```
)RUN
```

After you press the RETURN key, you should see this on the screen:

```
Hi again
)
```

BASIC has executed the PRINT statement. When you typed the PRINT statement with a number in front of it, BASIC stored the statement in memory as a one-statement *program*. When you typed the RUN statement, it was executed immediately, since it didn't have a number. RUN is a statement that tells BASIC to execute whatever program is currently stored in memory. Such execution of statements previously stored in memory is called *deferred execution*.

Statements without line numbers are executed immediately, and forgotten as soon as they are executed. Statements with line numbers are not executed immediately, but remain in your Apple's memory and can be executed again and again with the RUN statement. A deferred statement and the line number preceding it are often referred to as a program *line*. For example, consider

```
)30 PRINT "Bye now!"
```

Line 30 comprises both the PRINT statement and the line number 30.

Editing Your Program

Let's add another line to the program. Type

```
)20 PRINT "Hi yet again"
```

and press the RETURN key. Then type another RUN statement, and press RETURN once again:

```
)RUN
Hi again
Hi yet again
)
```

You can see that the statement at line number 10 is executed before the statement at line number 20. When BASIC executes statements in a program, it always executes them in the order of their line numbers, unless the program statements tell it to alter that sequence. Note that the order in which statements are executed is not necessarily the order in which you typed them.

By the way, you must surely have gotten the message by now that every program line you enter must be ended by pressing the RETURN key. From now on we'll stop repeating that reminder every time we enter a statement.

If you make a syntax error in typing a statement with a line number, BASIC won't detect the error until you type

```
)RUN
```

and it tries to execute the statement. For example, if you enter the statement

```
)15 PRITN "HELLO"
```

into your program, BASIC won't find the spelling error until it

attempts to execute line 15. At that point, execution stops and BASIC displays the message

```
?SYNTAX ERROR IN 15
)
```

where 15 is the line number of the incorrect statement. You should correct the erroneous statement before you run the program again. There are three ways that BASIC allows you to correct errors that you catch while entering program lines before you press RETURN:

1. Backspace over the error with the left-arrow key, and retype.

2. Use the cursor-move keys.

3. Cancel the entire program line you are typing. Press the X key while holding down the key marked CONTROL. This procedure is called "CONTROL-X." BASIC displays a backslash (\), and returns the cursor to the beginning of the next screen line. You can now retype the entire program line. The line you were typing before you typed the CONTROL-X will not be stored in memory.

To modify a line you have already typed into a program, simply type a new line with the same line number. For example, if you type

```
)20 PRINT "Hi another time"
```

this new version of line 20 replaces the old one. If you want to delete a stored statement altogether, type just its line number, then press RETURN.

In the example given above (where PRINT was spelled PRITN), you can use the cursor-move keys to correct the spelling. First press the ESCAPE key. This leaves the cursor with a "+" in its center, telling you that you are in cursor-move mode. Use the up-arrow key to move the cursor up to the line that you want to correct, and then either the left- or right-arrow keys to place the cursor on the first character in the line.

Press the ESCAPE key to exit cursor-move mode and use the right-arrow key to read in the characters in the line. Stop when you get to the first wrong character (the T) and type in NT, correcting the spelling of PRINT, and use the right-arrow key again until you reach the end of the line. When you press the RETURN key, the entire line will be accepted with your correction.

Using the cursor-move keys becomes more useful as your lines become longer or involve more complicated information, since you will tend to make more errors as you retype longer lines.

It's a good idea to leave room for additional line numbers between the numbers you use when you write a program. That way, if you want to insert a new program line between two lines already in memory, you can use one of the unused numbers. For example, if you have entered two lines numbered 15 and 20, you can put a new line between them by giving it the number 18. Line numbers must be within the range 0 to 63999, or a

```
?SYNTAX ERROR
```

message will be displayed on the screen.

If you want to see your program as currently stored in memory, type

```
)LIST
```

All statements in memory will be displayed on your screen in order of their line numbers.

If you want to get rid of all stored statements to start a new program, type

```
)NEW
```

Now if you type LIST, you will see that the program previously stored in memory is gone.

You can easily edit large Business BASIC programs with the Apple III Pascal Editor. Refer to the *Apple III Pascal Introduction, Filer, and Editor* manual for details about this.

Converting programs into text files to be edited by the Apple III Pascal Editor is described at the end of this chapter under CAPTURING PROGRAMS AS TEXT FILES.

Interrupting a Running Program

Now try typing this program:

```
10 PRINT "This is the first line"
20 PRINT "This is the second line"
30 GOTO 10
```

The purpose of this example is to show you how to stop a program that won't stop itself. The GOTO statement on line 30 tells BASIC to go to line 10 and proceed from there, but then BASIC finds the GOTO statement again in line 30 and goes back to 10. This happens over and over again. Try it by typing RUN.

When you get tired of seeing

```
This is the first line
This is the second line
```

displayed on the screen at high speed, you can stop the parade by holding down the key marked CONTROL and pressing the C key. This is called CONTROL-C, and is the normal way to interrupt a program while it is being executed. Your Apple III will beep and halt execution, with a message on the screen saying, for instance

```
?BREAK IN 20
```

where 20 is the line number of the statement that was being executed when you typed the CONTROL-C.

The program given above could have been written on a single line, if you had wanted it that way. For example,

```
10 PRINT "This is the first line" : PRINT "This is the
second line" : GOTO 10
```

This has the advantage of using somewhat less space in memory, but also is harder to read and understand. For that reason alone, the use of multiple statements per line should be considered an emergency measure for use only in extreme circumstances.

You are now acquainted with the simplest forms of several BASIC statements: PRINT, RUN, NEW, LIST, and GOTO, as well as CONTROL-C and CONTROL-X. You know what immediate and deferred execution are, and you know some simple ways to modify stored programs and correct typographical errors. (Take a cookie break, you've earned it.)

Controlling Text on the Screen

This section explains the statements that allow you to control the way Apple Business BASIC displays things on the screen.

LIST

LIST displays programs stored in memory by line number. You can halt a listing temporarily by typing CONTROL-7 (hold down the CONTROL key and press 7 *on the numeric keypad*). The listing resumes when you type a second CONTROL-7. Typing CONTROL-C terminates the listing altogether.

To list an entire program, type

```
)LIST
```

You can also specify a particular range of lines to be listed. The statement

```
)LIST 20
```

lists only line 20, if it exists. The statement

```
)LIST 33 TO
```

lists from line 33 to the end of the program.

```
)LIST TO 47
```

lists from the beginning of the program to line 47.

```
)LIST 23 TO 341
```

lists from line 23 to line 341, inclusive. In all the examples given, if the first line number doesn't exist, the next greater existing line number is used; if the second line number doesn't exist, the next smaller existing line number is used. If the two line numbers are the same, only one line is listed (assuming that it exists). If the first line number is greater than the second, nothing is listed at all.

In place of the reserved word TO, you can substitute a comma or a hyphen:

```
)LIST 52, 89
)LIST 38-725
```

The Reserved Variables INDENT and OUTREC

You can set the number of spaces to be used to indent FOR..NEXT loops (described in the chapter Controlling Program Execution) in your program listings. This is normally set to two spaces, but can be changed by giving a new value to the modifiable reserved variable INDENT. For example,

```
)LIST
10 REM Count, and print, 50 numbers.
20 FOR X=1 TO 50
30     PRINT X
40     NEXT X
50 END
```

Would look somewhat different if you changed the value of INDENT to 5, like this:

```
)INDENT=5
)LIST
10 REM Count, and print, 50 numbers.
20 FOR X=1 TO 50
30       PRINT X
40       NEXT X
50 END
```

The maximum length of lines output by the LIST command is set by the value of OUTREC. If you have a printer that lacks automatic wrap-around, such as a Qume, you can set OUTREC to avoid having to watch the printer cheerfully running off the right side of the paper onto the platen.

As soon as the printer column position equals the value of OUTREC, BASIC sends a RETURN and LINEFEED to the printer to continue printing at the next line.

If the value of OUTREC is less than the value of INDENT, a ?RANGE ERROR message is given.

DEL

DEL deletes lines from the program stored in memory. You can specify either a single line or a range of lines to be deleted. The statement

```
)DEL 48 TO 52
```

deletes the range of lines from line 48 to line 52, inclusive. It is equivalent to typing

```
)48
)49
)50
)51
)52
```

since typing a program line number followed with a return deletes that line from the program in memory.

If the first line number (line 48 in the example) doesn't exist, the next greater existing line number is used. If the second line number (line 52 in the example) doesn't exist, the next smaller existing line number is used. The statement

```
)DEL 163
```

deletes only line 163, if it exists.

In place of the reserved word TO, you can substitute a comma or a hyphen:

```
)DEL 73, 193
)DEL 996-1010
```

Attempting to delete the line currently being executed in a running program or any smaller numbered lines causes a

```
?RANGE ERROR
```

message to be displayed. For instance, either

```
)300 DEL 245
```

or

```
)500 DEL 500
```

will cause an error. Unlike LIST, DEL requires that both the starting and ending line numbers be given if you want to delete more than one line of a program.

Window

WINDOW allows you to set the position and size of the *text window* where text is displayed on the screen. Think of the text window as a rectangle within the total screen area limiting the area on the screen

where BASIC may display text. Here is an example of a WINDOW statement:

```
)WINDOW 37,9 TO 44,16
```

In the example above, the first number of each pair is the column on the screen, and the second number of each pair is the row number.

The first pair of numbers specify the horizontal and vertical coordinates of the upper left corner of the text window, and the second pair specify the coordinates of the lower right corner. The example above will create a text window 8 columns wide and 8 screen lines high, in the center of your screen. When a WINDOW statement is executed, the cursor moves to the lower left corner of the specified window.

WINDOW statement coordinates may be specified by any arithmetic expression. Each of the four expressions must have a value within the range 0 to 255, or an

```
?ILLEGAL QUANTITY ERROR
```

message will be displayed. Expressions with a value of 0 are treated as though they had a value of 1. If the values of the expressions specified in a WINDOW statement would cause the size of the window to exceed that of the screen, the window is truncated to fit. For example, if the text window is set wider than the 80-column screen width, it will be truncated to 80 columns. Likewise, 24 screen lines of text will fill the screen, so setting the window higher than 24 lines causes truncation of any text that would have been displayed below the 24th screen line.

The parameter values used by WINDOW are assumed to be relative to the limits of the default screen of 80 by 24 lines.

TEXT

The TEXT statement sets the display screen to the usual full-screen text mode, canceling any other text or graphics mode in use. All graphics being displayed (except text characters) is cleared from the

screen, the cursor is moved to the left hand column of the next line of the screen, and a prompt character is displayed.

TEXT also resets the text window to full screen size (24 screen lines by 80 columns), equivalent to

```
)WINDOW 1,1 TO 80,24
```

The Reserved Variables VPOS and HPOS

You can control the position of the cursor within the text window by assigning numerical values to the reserved variables VPOS (Vertical POSition) and HPOS (Horizontal POSition).

Assigning a value to VPOS moves the cursor vertically to the corresponding screen line within the current text window. Assigning a value to HPOS moves the cursor to the corresponding horizontal character position on the current screen line, relative to the text window.

The top screen line within the text window is line 1 and the number of the bottom screen line is equivalent to the height of the window. When VPOS is used to move the cursor vertically, the cursor's horizontal position on the screen is not affected.

The leftmost character position within the window is column 1 and the rightmost is equivalent in value to the width of the window. When HPOS is used to move the cursor horizontally, the cursor's vertical position is not affected.

Values assigned to VPOS and HPOS can be specified by any integer or real arithmetic expression. For example, the statements

```
)VPOS=8
)HPOS=6
```

will move the cursor to the sixth character position on the eighth screen line from the top of the text window. You can find the current

position of the cursor by referring to the values of VPOS and HPOS. For instance,

```
)PRINT VPOS
```

displays the current vertical position of the cursor on the screen, relative to the upper margin of the text window.

```
)PRINT HPOS
```

causes the current horizontal position of the cursor, relative to the left margin of the window, to be displayed.

VPOS and HPOS perform absolute moves within the text window, and cannot move the cursor to a position outside of the text window. Values greater than the height or width of the text window are treated as equal to the largest possible values within the window. Assigning values greater than the height of the text window to VPOS, causes the cursor to move to the bottom screen line within the window. Assigning values greater than the width of the text window to HPOS causes the cursor to move to the right margin of the window. The value 0 is converted to the value 1. Assigning values outside the range 0 to 255 to either VPOS or HPOS causes the message:

```
?ILLEGAL QUANTITY ERROR
```

HOME

HOME clears all text within the current text window and moves the cursor to the upper left corner of the window.

INVERSE and NORMAL

INVERSE sets all subsequent display to black letters on a white background. NORMAL sets the display to white letters on a black background. Characters on the screen before the execution of the INVERSE or NORMAL statement are not affected.

Type this program, and then run it:

```
)10 INVERSE
)20 PRINT "Inverse"
)30 NORMAL
)40 PRINT "Normal"
)50 GOTO 10
```

Use CONTROL-C to stop execution of the program.

INVERSE and NORMAL have no effect on characters read from or written to files.

System and Utility Statements

The statements described in this section allow you to transfer programs between memory and disks, erase programs from memory, save the contents of memory as a program, and start and stop programs. Other statements help you debug programs, determine the size of a program in memory, and perform other tasks.

Memory Management

One of the most important resources at your command when programming is the Apple's memory. Several tools to help you to more efficiently use memory are described below.

NEW

NEW erases the current program and all its associated variables from memory, and closes all open files except a text file being executed (see the EXEC statement in the Section AUTOMATIC EXECUTION). There are no options associated with NEW.

CLEAR

CLEAR sets all numeric variables to zero, all strings variables to null strings, clears all BASIC pointers and stacks, and closes all open disk files except a file being executed. There are no options associated with CLEAR.

If a large program (or one with many variables) has just been run and halted, trying to add additional statements may give an

```
?OUT OF MEMORY ERROR
```

caused by variables left over from the previous time the program ran. CLEAR will allow more statements to be added. (It will also, of course delete the old variables.)

The Reserved Variable FRE

FRE is a reserved variable storing the amount of remaining available memory, measured in bytes. Try typing

```
)NEW
)PRINT FRE
66556
)
```

In this case, you have about 66K bytes of memory.

FRE allows you to check on the amount of space remaining in memory when you are doing things like adding to an existing program, checking the actual size of a program, or trying to increase the size of an array. See the Chapter VARIABLES AND CONSTANTS, and the Appendices VARIABLE MAPS and SPACE SAVERS, for further information on more efficiently using memory space.

Loading and Saving Programs

Programs can be stored in the form of files on disk. The statements described in this section allow you to move programs between memory and diskette files. Before you read the rest of this chapter, or the chapter FILE I/O, you should be familiar with files, pathnames, local filenames, directories, and prefixes as described in the *Apple III Owner's Guide*.

LOAD

LOAD reads a specified BASIC program from a disk file, and stores it in memory. The pathname of the program to be loaded must follow the reserved word LOAD (see the chapter FILE I/O for an explanation of pathnames). For example, to load a program named Countdown on a disk named Shuttle, use

```
)LOAD /Shuttle/Countdown
```

When the program is loaded, its numeric variables are all set to zero, string variables are set to null strings, and all files are closed, with the exception of any EXEC file being executed. Any existing program is cleared from memory.

If the program specified does not exist on the disk, the

```
?FILE NOT FOUND ERROR
```

message is displayed.

Attempting to load a file other than a BASIC program causes the message

```
?TYPE MISMATCH ERROR
```

to be displayed.

SAVE

The SAVE statement writes a copy of the program currently in memory to a diskette file. You must specify the file to be saved by following the reserved word SAVE with a pathname, for example:

```
)SAVE .D2/Inventory
```

If there is already a BASIC program with the same pathname on the

disk, it will be overwritten and lost. If a locked BASIC program with the same name is on the disk, you will get a

```
?FILE LOCKED ERROR
```

message. If a file having the specified name that is *not* a BASIC program is on the diskette, a

```
?TYPE MISMATCH ERROR
```

message will be displayed.

Starting and Stopping Programs

If things get out of hand while your program is running (or the bomb squad happens to make a personal visit), you might want to stop a program while it is running and later restart the stopped program. The statements that allow you to do these things are described below.

RUN

RUN is used to start running a program. When a RUN statement is entered, BASIC clears all variables, closes all open files except executing text files, and begins to execute the program in memory beginning with its smallest line number.

If there is no program in memory, the cursor drops to the next screen line, and redisplays the prompt. For example, to execute whatever program is currently in memory, type

```
)RUN
```

(as you well know by now). If you do not want execution to begin at the smallest line number, but at some other line (for instance line 205), use

```
)RUN 205
```

and execution begins at line 205. If you specify a non-existent line number, the

?UNDEF'D STATEMENT ERROR

message appears.

If you want to run a program that is not stored in memory, but is in a disk file, you can specify the program to be run by giving its pathname. For instance, to RUN the program stored in a file named ASSETS, use

)RUN ASSETS

And if you wanted to begin execution at line 7254, use

)RUN ASSETS, 7254

If the file you specify is not found after searching the disk, the

?FILE NOT FOUND ERROR

message is given. If the file is found, the current program and all of its variables are erased from memory, and all open files (except an executing text file) are closed. The program specified in the RUN statement is then stored in memory and BASIC begins executing it at either the lowest numbered line or at the specified line.

The RUN statement is the last statement executed in any line. For instance, in this line:

)IF X= 1 THEN RUN 1000 : I=1

the variable I will never be assigned the value 1 if X equals 1.

Trying to run a program that is not written in BASIC generates a

?TYPE MISMATCH ERROR

message.

STOP

STOP halts execution of a program, closes any executing text file, returns BASIC to immediate execution, resets the output file to the console, and redisplays the prompt. STOP displays the message, for example

```
?BREAK IN 8712
```

where 8712 is the line number of the program line containing the STOP statement. The program in memory is not altered in any way. STOP has no options associated with it.

END

END is identical to STOP except that no message is displayed when it is executed. END has no options.

CONT

CONT causes execution of a program that has been halted by STOP, END or CONTROL-C to be continued. The CONT statement resumes execution at the statement immediately following the one at which execution was suspended, not with the first statement in the next program line. For example,

```
)0 PRINT "Program Begins"
)10 END: PRINT "Back again"
)20 PRINT "Blitz"
)RUN
Program begins
)CONT
Back again
Blitz
)
```

CONT does not clear the program, or reset the variables in memory, and there are no options associated with it.

CONT has no effect if there is no program in memory.

A program halted by an error may be continued. BASIC will attempt to continue execution starting with the statement in which the error occurred. An error made in immediate execution will not prevent a program from being continued.

A program that has had any of its statements altered, or any new statements added, cannot be continued. If you try, the

```
?CAN'T CONTINUE ERROR
```

message will be displayed. Variables in a program can be changed using assignment statements in immediate execution. For example:

```
)10 X=4 : PRINT X
)20 STOP
)30 PRINT X
)RUN
4

BREAK IN 20
)X=X+2
)CONT
6
)
```

CONT is ignored when used within a program.

Handling Large Programs

Even with the large amount of memory available for Business BASIC programs, you'll soon find ways to fill it up. This has been allowed for with the CHAIN statement described below.

CHAIN

If a program requires more memory than is available on your Apple, you can split the program up into smaller pieces, and execute them individually with the CHAIN statement.

CHAIN automatically loads and runs a specified program, without clearing the values of the variables left over from the previous program, or closing any files the previous program had left open. This allows variables used in one program to be used in another. The pathname of the program to chain must follow the reserved word CHAIN. For instance, to CHAIN to a program named Tires, use

```
)CHAIN Tires
```

If the program specified in the CHAIN statement does not exist on the diskette, then a

```
?FILE NOT FOUND ERROR
```

message will be displayed.

Execution of the specified program begins at the smallest line number, unless you specify otherwise. Therefore,

```
)CHAIN /Link/Fence, 800
```

causes execution to begin at line number 800 of the program named Fence on the volume /Link.

If the chained program uses a variable that was not used in the program executed before it, a new variable will be created; otherwise the old variable will be used.

If the chained program dimensions an array that was dimensioned in the previous program, a

```
?REDIM ERROR
```

occurs.

Here is an example of how you might use the CHAIN statement:

```
)10 PRINT "Program One"
)20 DIM A(44)
)30 CHAIN Program Two : REM Now Program Two will be
 loaded and run
)RUN
Program One
Program Two speaking
0
10 PRINT "Program Two speaking"
20 PRINT A(32)
30 LIST
)
```

will be displayed if you have a program with the file name "Program Two" like the one given below.

```
10 PRINT "Program Two speaking"
20 PRINT A(32)
30 LIST
```

Debugging Programs

In spite of what many people think, programming is not just for an intellectual elite. Even though it is not difficult in principle, it is somewhat unforgiving of various sorts of mistakes.

While Business BASIC can catch things like spelling mistakes and faulty syntax, it is more difficult to catch more subtle sorts of errors. The statements described in the section below are supplied to aid you in catching these types of errors.

TRACE

TRACE functions while a program is executing and prints a "#" followed by the number of each line of the program as it executes. There are no options associated with TRACE.

TRACE is cancelled by NOTRACE, RUN followed by a pathname, and by LOAD followed by a pathname. TRACE is not cancelled by CHAIN or RUN alone.

If the program being traced displays characters on the screen, the output of TRACE combines with the display in unpredictable ways. The line numbers displayed may appear around and within, or even be overlayed with what the program displays.

Using TRACE with a program that uses ON KBD can be risky because TRACE slows execution to the point that many keys can be pressed while the ON KBD statement is being executed. This can cause a

```
?STACK OVERFLOW ERROR
```

giving a false error message for the program.

If you TRACE a program that uses the OUTPUT statement, the line numbers listed by TRACE will be included in the file written to by OUTPUT. (Of course, if that's what you want.)

NOTRACE

There are no options associated with NOTRACE, it simply cancels TRACE; the line numbers of executing program statements are not displayed. To use it, type

```
)NOTRACE
```

Special Keys

Some keys have special functions for users of Business BASIC, as described below.

CONTROL-C

Pressing CONTROL-C while a statement in a program is being executed is equivalent to inserting a STOP statement immediately after the statement. CONTROL-C can be used to stop the execution of any statement; for instance, to terminate the display generated by a LIST statement. It can also be used to abort a program waiting at an INPUT statement if it is the first character typed as an input, and only after the RETURN key is pressed.

CONTROL-C will not stop execution of a program if an ON KBD statement has been executed, since CONTROL-C is treated just as any another keystroke. However, the statement(s) that ON KBD causes to be executed can issue an END or STOP if a CONTROL-C is pressed.

CONTROL-C will also not stop a program if an ON ERR statement has been executed by the program, since CONTROL-C is interpreted as error number 255.

Business BASIC will not recognize a CONTROL-C while a program is waiting for an I/O operation to be completed. If a printer or disk drive, for example, is not properly connected, the program may only be terminated by pressing CONTROL-RESET.

CONTROL-5, -6, -7, -8, -9

Control-5, -6, -7, -8, or -9 entered from the numeric keypad all perform special functions. Please refer to the *Apple III Owner's Guide* for a description of these functions.

RESET

Pressing the RESET button is equivalent to pressing CONTROL-C, except that RESET clears some program stacks and pointers, and you cannot use an ON KBD or ON ERROR statement to handle RESET.

CONTROL-RESET

Pressing the RESET button while holding down the CONTROL key halts I/O operations as soon as any critical steps are finished (for example a WRITE operation), and re-boots your Apple. CONTROL-RESET causes an electrical reset of the entire system. Anything stored in memory is lost after pressing CONTROL-RESET (including your program and BASIC).

Automatic Execution

In addition to immediate execution, directed from the keyboard, and deferred execution directed by programs stored in memory and started by a RUN command, Apple Business BASIC allows operation to be directed from a text file. This increases the power of your Apple by allowing an application to be run without direct keyboard entry. This is made possible by the EXEC statement.

EXEC

EXEC simulates keyboard input by reading the contents of a text file and acting on this as though you were typing commands from the keyboard. You use EXEC by entering the reserved word EXEC followed by the pathname of the text file containing the commands to be executed. For example,

```
)EXEC /Workdisk/Business/Gamestarter
```

If the file you have specified is not a textfile, the

```
?TYPE MISMATCH ERROR
```

message is displayed.

When you use EXEC, all input is expected by BASIC to come from the text file being EXECed. This remains in force until either CONTROL-C, a STOP statement, an error, or the end of the file is reached. When any of these occur, control is returned to the keyboard.

EXEC automatically opens the file that it uses, but this file is not considered by BASIC to be one of the total of ten files that may be open at one time.

Deferred Immediate Statements

You may find that you are running some application on your Apple made up of a series of programs running one after the other. If the individual programs require little or no interaction from you, you might find yourself spending time waiting for one program to finish just to begin operation of the next one. For example,

```
)RUN Program1

wait...wait...

)RUN Program2

wait...wait...

)RUN Program3
```

and so on.

EXEC can let you begin the entire sequence and then go to lunch or play tennis or build up your business while the computer works. (Computers are supposed to work for us, right?) You can do this by typing

```
)EXEC Runner
```

if Runner is a text file containing the lines

```
RUN Program1
RUN Program2
RUN Program3
```

A program like the one given below produces text files that EXEC will run:

```
10 REM Program "TextfileMaker"
20 OPEN# 1 AS OUTPUT,"Runner" : REM Open the text file
30 INPUT "-"; S$ : REM Get a line of text
40 IF LEN(S$)=0 THEN 70 : REM Was only return pressed?
50 PRINT# 1; S$ : REM Write the line into the file
60 GOTO 30 : REM Go back and get another line
70 CLOSE
80 END
```

TextfileMaker prints a hyphen to prompt you to enter a lines of text, one at a time, until you type nothing but a return. The lines will be written into a text file named Runner. If you run TextfileMaker, and enter the following responses:

```
)RUN
-RUN ProgA
-RUN ProgB
-RUN ProgC
-
)
```

you will make the desired EXEC file. When you type

```
)EXEC Runner
```

the contents of Runner will be output, one line at a time, *exactly* as though *you* were entering them at the keyboard. When the line containing RUN ProgA is output, BASIC executes that line as a RUN statement, and runs ProgA. When ProgA is finished running, the next line of text is output, causing ProgB to run. After ProgB finishes, ProgC is RUN. After ProgC finishes, control returns to the keyboard.

Note that you would *not* get the same result by writing a program to run the other three programs. a program containing the lines;

```
10 RUN "ProgA"
20 RUN "ProgB"
30 RUN "ProgC"
```

runs only the first program since executing the RUN statement clears the current program in memory, thus wiping out lines 20 and 30.

EXEC accepts any legal statement in BASIC, including conditional statements. This allows you to make the order of program execution dependent on the result of a program's execution. For example, you might want to run ProgB after ProgA only if the value of X as determined by ProgA is negative. If X is positive, you might want to run ProgC instead. To do this, your EXEC file Runner should contain the following:

```
RUN ProgA
IF X<0 THEN RUN ProgB : ELSE RUN ProgC
```

Remember that an executing textfile replaces the keyboard. If an INPUT or GET statement occurrs in a program, it takes its input from the next line of the text file, *not* the keyboard.

If you call an EXEC file from a program, the EXEC statement must be followed by END, as shown below.

```
10 A$="WoodFile"
20 EXEC A$ : END
25 INPUT A$
30 GOTO 10
```

You can reenter the calling program after the EXEC file is finished with either RUN linenumber or GOTO linenumber as the last line of the EXEC file.

You can force the Apple to take input from the keyboard while a text file is executing by opening the file .CONSOLE in your program or in the text file and doing file input. (See the chapter on File I/O.)

Creating Deferred Statements

When a deferred statement (one with a line number) occurs in an executing text file, the effect is just as if you typed it on the keyboard. The line is stored in memory, and can be run or saved.

Suppose that you have a program in memory (either one that you have just typed, or one previously saved) and you execute a text file containing deferred statements. If these statements have line numbers different from those already in the program, the effect is to add the new statements to the program. If any new line has the *same* number as a line in the program, the new line *replaces* the old one.

Suppose you write a set of programs using modulo functions defined for real variables. A modulo function takes the first expression and returns the remainder after dividing first expression by the second expression, called the "modulus." (The value returned by 7 modulo 5 is 2. 8 modulo 2 returns 0.) This is not the same as the MOD function for long integers in Business BASIC!

If we call the real variable A and the modulus B, we can define a function named ModB as follows:

```
)30 DEF FN ModB(A)=(A/B - INT(A/B))*B
```

Of course, "B" has to be replaced by an actual number. The result is meaningful only if A has a positive value.

If you wanted to have Mod12, Mod15, and Mod255 functions, you could begin each program with the lines

```
1 DEF FN Mod12(A)=(A/12 - INT(A/12))*12
2 DEF FN Mod15(A)=(A/15 - INT(A/15))*15
3 DEF FN Mod255(A)=(A/255 - INT(A/255))*255
```

Instead of typing these lines into each and every program needing these functions, you can create a text file called MODS that contains them, and then enter

```
)EXEC MODS
```

A program like TextfileMaker (shown earlier) could be used to create a text file, called Mods to do this. Before running TextfileMaker, alter line 20 to produce a file named Mods, instead of the one given (producing Runner). When prompted, enter

```
)RUN
-1 DEF FN Mod12(A)=(A/12 - INT(A/12))*12
-2 DEF FN Mod15(A)=(A/15 - INT(A/15))*15
-3 DEF FN Mod255(A)=(A/255 - INT(A/255))*255
-
)
```

Now whenever you want to write a program using these MOD functions, start by typing EXEC MODS.

Any time you want to run a program needing these functions, enter EXEC Mods *before* running the program. (After *loading* it!) The same principle can be used whenever you have some lines to insert in several programs. Make sure that the line numbers in the text file and the line numbers in your program don't overlap.

Capturing Programs as Text Files

Some of your programs may contain lines that you would like to insert in other programs. To do this, you must convert the program into a text file to be inserted into other programs by using EXEC as described above. Use the OUTPUT# statement to send console output to a text file. If you then use LIST, all the output is written to the text file.

The resulting text file may be edited with the Apple III Pascal Editor (as an ascii file). After editing, you can use EXEC to read the edited program back into memory and (after stopping it) save the program from memory into a program file.

Suppose you want to save lines 20 through 150 of a program starting at line 10 in a text file named "Goodlines." First load the program, then type the following:

```
)1 CREATE "Goodlines," TEXT
)2 OPEN# 1 AS OUTPUT, "Goodlines" : OUTPUT# 1
)3 LIST 20 TO 150
)4 CLOSE: END
```

Now RUN the program. This creates a text file named "Goodlines" that you can insert into other programs with EXEC.

This little four-line program is itself a candidate for being captured as a text file to be put into other programs, to capture lines from those programs and turn them into text files to be inserted into still other programs. We'll modify it a little by changing the filename in Line 1 to XXX and the line numbers in the LIST statement to YYY,ZZZ. Each time you execute a text file to insert these lines into a program, you can change XXX to whatever filename you want, and YYY,ZZZ to the line numbers you want to capture. We'll call this text file "CAPTURE." Here's a program that will create CAPTURE:

```
1 CREATE "XXX," TEXT : OPEN# 1 AS OUTPUT, "XXX"
2 OUTPUT# 1
3 LIST YYY TO ZZZ: PRINT
4 CLOSE: END
100 CREATE "CAPTURE," TEXT : OPEN#1 AS OUTPUT,
   "CAPTURE"
101 OUTPUT# 1
102 LIST 1 TO 4: PRINT
103 CLOSE: END
RUN 100
```

The immediate execution statement RUN 100 causes the last four lines of the program to be run, and they capture the first four lines as a text file named CAPTURE. Whenever you want to capture lines from a program in memory, type

```
)EXEC CAPTURE
```

change XXX, YYY, and ZZZ to whatever you want, and run it (in memory).

2

Tools of Your Trade

2

2

Tools of Your Trade

Introduction

The main function of your computer is information handling, and Business BASIC provides you with the tools you need to do the job. These tools are described in this chapter, and some definitions should be presented before you jump in.

A *variable* is a place to store a value. It can be thought of as a "box" that can hold a single value. Variables have limits on the types of things they can contain: the box can contain only things of a certain size. Also, there are certain tasks for which some variables are better suited than others; for instance, a filing cabinet is not a suitable place to store fruit, but a wooden crate may be.

Variables are referred to by their *names*. The box in the example above might be labeled "junk"; the fruit crate could be labeled "apples3". Both "junk" and "apples3" are legal variable names. A variable name is a sequence of characters beginning with a letter and followed by from 0 to 63 additional letters, digits, or periods. Lowercase letters in variable names are considered equivalent to their uppercase counterparts. For example, the names "junk" and "JUNK" refer to the same variable.

Not all possible name can be used. Some words are used by Business BASIC to refer to statements and functions of the language, these are *reserved words* that may not be used as variable names.

A *constant* is an unchanging or fixed value. There are two kinds of constants in BASIC, numeric constants and string constants. A numeric constant is a value written as a number; a string constant is any sequence of characters enclosed in quotation marks. For example, 3.14159265 is a numeric constant; "dangle" and "463" are string constants. Notice that the numeric constant 463 and the string constant "463" do not represent the same value.

The rest of this chapter describes the tools that Business BASIC provides for your use in solving your problems.

Variable Types

There are four elementary *variable types* in Apple Business BASIC: integers, reals, long integers, and strings. The first three types represent numbers of various kinds, the last type represents sequences of characters.

The *type* of a variable is determined by the last character of its name: % for integer, $ for string, and & for long integer. In the absence of any of these special trailing characters, the variable type is considered to be real by default.

Here are examples of names of the four variable types:

Variable Name	Variable Type
Length	real
Marbles7%	integer
Light.Years&	long integer
Myname$	string

Variables are created when they are first used in a program. When BASIC sees a variable name in a statement, it first checks to see if it already has a variable with that name, a task not unlike looking someone up in the phone book. If it finds the name, then it knows where in memory to find the value stored in the variable (just as someone's name allows you to locate that person's number in the phone book).

If BASIC doesn't find a variable matching the name already in memory, it immediately creates a new variable. It places the new variable name in the directory of variable names (the "phone book") so it can be found later, then finds free space in memory to store the value that the variable will contain. It also notes the type of the new variable.

For numeric variables, BASIC stores the value 0 in the variable; for string variables, it stores an empty or null string in the variable.

Integers

An *integer* is any positive or negative whole number without a decimal point. The numbers 3, −3, and 20000 are examples of integer constants. Integer variable names must end with a percent sign (%). For instance, I% is the name of an integer variable. Integer variables can store integer values up to five digits long, from −32768 to 32767. Attempting to assign a value beyond this range to an integer variable generates the message

```
?ILLEGAL QUANTITY ERROR
```

Integers are displayed without leading zeros (with a leading minus sign if negative) followed by up to five digits without a decimal point.

Integers are useful when fractional parts of numbers are not needed. They can also be used to speed up some types of calculations where real numbers are not required.

Long Integers

Long integers are similar to ordinary integers, except that they may be up to 19 digits long. They may not be mixed in arithmetic expressions with regular integers or reals (described below). Long integer variable names must end with an ampersand (&). A long integer value can range from −9223372036854775808 to 9223372036854775807. Exceeding this range causes the message

```
?OVERFLOW ERROR
```

to be displayed. Entering the number −9223372036854775808 from the keyboard will also cause the same error.

Long integers are displayed without leading zeros (with a leading minus sign if negative) followed by up to 19 digits without a decimal point.

Many types of financial programming could profit by using long integers and doing all calculations in pennies. The decimal point could be inserted later when reporting results by using the SCALE function, described later in this manual.

Reals

A *real* is any positive or negative number. Unlike integers, reals may have a fractional part. The numbers 3, 33, 3.3, −3.3, 3., −3.0, .3, and −.3 are examples of real constants. A numeric constant with a decimal point is always of type real, even if it has only zeros to the right of the decimal point: for example, 3. is a real constant. However, not all reals must have decimal points.

Reals whose absolute values are greater than or equal to .01 and less than 999999.2 are expressed in conventional notation. For example, 1, +1, −1., 3.14, 999.999, and −0.2 are all real numbers expressed in conventional notation.

A real may also be expressed in scientific, or "E" (for exponent) notation, such as 3.3E2, −3.3E4, 3.3E-4, or −3.3E-3. The real number 5.3E12, for example, is equal to 5.3 times 10 raised to the 12th power. Here are examples of conventional notation vs. scientific notation.

Conventional Notation	Scientific Notation
300	3E+2 = 3*(10^2)
320	3.2E2 = 3.2*(10^2)
.44	4.4E-1 = 4.4*(10^−1)
−.033	−3.3E-2 = 3.3*(10^−2)
1000000000000	1E+12 = 1*(10^12)

Reals are automatically expressed in scientific notation when the absolute value of the real is less than .01, or greater than or equal to 999999.2 . A real whose absolute value is less than 2.9388E-39 is considered equal to 0 by Business BASIC.

You can enter from the keyboard any number in E notation that is within the legal range of reals in Business BASIC.

When BASIC displays a real, it normally displays six digits, excluding any exponent. Any significant digits beyond the sixth are rounded off. Leading zeros to the left of the decimal point and trailing zeros to the right of the decimal point are not displayed. The decimal point is not displayed if there is no non-zero fraction.

Reals must be within the range −1.7E38 to 1.7E38 or the message

```
?OVERFLOW ERROR
```

will be displayed.

Strings

A *string* is a sequence of characters enclosed within quotation marks. String variable names must end with a dollar sign ($). Strings may contain from 0 characters (the null string) to 255 characters. The number of characters in a string is referred to as its *length*. Strings are not fixed in length, but may grow or shrink as necessary.

Strings must be *bounded* by quotation marks, but may not *contain* quotation marks. For example, the statement

```
)PRINT "What does "FOO" mean?"
```

causes

```
)What does 0 mean?
```

to be printed. The quotes cause BASIC to assume that FOO is a real variable with a value of 0, since it has not been assigned a value.

Strings may contain single quotes; the statement

```
)PRINT "What does 'FOO' mean?"
```

will not cause an error.

You might be wondering how to print quotes, since they can't be included in strings. Try this:

```
)PRINT CHR$(34);"One small step for a man...";CHR$(34)
```

You can use the CHR$ function (described later) to output any ASCII character. Be careful, since some of them have unexpected effects on the console driver. A bit of experimentation would be to your advantage, after you have read the *Standard Device Drivers* manual.

When a program is run, all string variables initially contain the null string.

Reserved Words and Variables

Some variable names are not allowed because they are words having special meanings for BASIC. One of these words is PRINT, another is INVERSE, another is WINDOW, and so on. These words are *reserved words* in BASIC; they cannot be used as variable names. For example,

```
)PRINT INVERSE
```

causes the

```
?SYNTAX ERROR
```

message to be displayed. But

```
)PRINT XINVERSE
```

is perfectly legal.

One group of BASIC reserved words are known as *reserved variables*. The reserved variables are EOF, ERR, ERRLIN, FRE, and KBD. Your program can refer to the values of these variables, but you cannot assign values to them. For example,

```
)PRINT ERRLIN
```

is a legitimate statement, but

```
)ERRLIN=50
```

is not.

HPOS, VPOS, INDENT, OUTREC, and PREFIX$ are *modifiable reserved variables* to which you can assign values with assignment statements (see the Chapter CONTROLLING PROGRAM EXECUTION).

An alphabetical list of all BASIC reserved variables appear in the appendix RESERVED WORDS.

Arrays

An *array* is an ordered collection of single variables, all of the same type. The name of the whole collection, called the *array name*, can be any legal variable name. The last character of the name determines the type of all the variables in the array.

The individual variables (or *elements*) within an array are numbered, starting with number 0. To refer to any element within an array, you specify the name of the array, followed by the number of the element enclosed in parentheses. For example:

```
)PRINT AR(3)
```

displays the contents of element number 3 in the array named AR, and

```
)PRINT PRICES(147)
```

displays the contents of element number 147 in the array named PRICES. The numbers in parentheses following the array name are called the array's *subscript*. The subscript specifies one unique element within the array.

An array can have more than one *dimension*. A one-dimensional array is just a single list of variables, one after the other in a line. The

variable name for such an array needs only one number to specify each variable. For example, a one-dimensional array, named Do, with six elements includes the elements

Do(0), Do(1), Do(2), Do(3), Do(4), Do(5)

Note that the largest subscript value is one less than the total number of elements in that dimension of the array.

In a two-dimensional array two subscripts are needed to specify an individual element. For example, a two-dimensional integer array named D% might include the elements

D%(0,0), D%(1,0), D%(2,0),
D%(0,1), D%(1,1), D%(2,1),
D%(0,2), D%(1,2), D%(2,2)

D%(2,1) would specify the element in the third column and second row.

An array may have three or even more dimensions. The number of dimensions is the number of subscripts needed to specify an individual element within the array. Here are some examples:

Array	Type	Dimensions	Element Accessed
J(2)	Real	1	Element 3
Fudge%(3,2)	Integer	2	Column 4, row 3
Mumble$(7,3,1)	String	3	Column 8, row 4, elevation 2
Frotz&(77,0)	Long Integer	2	column 78, row 1

DIM

To create an array, you must first tell BASIC the maximum number of elements and dimensions you want the array to accommodate. To do this you use a DIM statement (for dimension). For example, the statement

)DIM Mind%(7,2,3)

creates an array named Mind% with three dimensions, with the first subscript ranging from 0 to 7, the second from 0 to 2, and the third from 0 to 3. The character "%" specifies that this will be an integer array.

Remember that the list of dimensions for all arrays begin with subscript number 0, so the number of elements in each dimension is always one greater than the greatest subscript value. Thus, the number of elements in the array Mind% is equal to (7+1)*(2+1)*(3+1) or 96.

More than one array can be defined with a single DIM statement by separating the arrays with commas:

```
)DIM LightS%(78,9), Bulbs$(2,45), Lanterns&(9,0,8), LY(16)
```

The statement above creates an integer array named Light%, a string array Bulbs$, a long integer array Lanterns&, and a real array LY.

Subscripts can range from 0, which is always the first element of each dimension of the array, to a maximum value of 32767. Subscripts may be by any integer or real expression; but the resulting value is converted to an integer before the particular element is actually accessed. Long integer arithmetic expressions may not be used in subscripts, unless first converted to reals with the CONV function described later in this chapter.

If you refer to an array before defining it with a DIM statement, BASIC automatically creates an array having 11 elements per dimension, with subscripts numbered from 0 to 10. For example, when the statement

```
)PRINT TM$(0,0,0)
```

is executed, BASIC defines an array TM$, just as if the statement

```
)DIM TM$(10,10,10)
```

had preceded the PRINT statement. If the statement

```
)PRINT D&(18,LOOP5)
```

is executed before the array D& is defined, a

?RANGE ERROR

is displayed, because the subscript 18 is greater than the default maximum of 10 for each dimension.

If the value of a subscript refers to either a nonexistent dimension or a nonexistent element (one that is greater than the highest numbered element in a given dimension), the

?BAD SUBSCRIPT ERROR

message is given. In the example below, both of the statements after the DIM statement will cause this error.

```
)DIM RealArray(1,2)
)PRINT RealArray(1,3)     (DIM statement did not include 3)
)PRINT RealArray(1,1,0)   (DIM statement did not create a
                           third dimension)
```

Statements

Statements can be used in either immediate or deferred execution. Immediate statements have no line number, and are executed immediately when entered. Deferred statements have line numbers, and are stored in memory (as part of a program) for later execution.

A list of statements may share one line number. In this case, adjacent statements must be separated by a colon (:). The last statement in the list must end with a return. This is called a *statement list*. For example, the following are both legal statement lists:

```
)TEXT : PRINT "Up" : WINDOW 10,10 TO 20,20 : PRINT "Over"
)10 TEXT : LIST : INVERSE : GOTO 10
```

No statement or statement list may exceed 250 characters. If a statement or statement list should exceed 250 characters, BASIC displays a backslash at the end of the line, and ignores the entire line.

Expressions

There are four kinds of *expressions*: arithmetic, long integer, string, and logical. An expression can be a single constant or variable, or it can be an elaborate mathematical grouping of *operators* and *operands*. Operators are symbols representing mathematical operations. Operands are the variables and constants that operators work on. For example, in the expression 2 + 3 , the operands are 2 and 3 and the operator is the + symbol.

Only integers and reals may be mixed freely in expressions because BASIC automatically converts all integers into reals before an expression is evaluated. All operands must be reduced to a common type before an expression can be evaluated, if you wish to mix either long integers or strings with other types of variables and constants in an expression. (For an explanation of type conversion functions, see the Section on Apple Business BASIC Functions.)

Arithmetic Expressions

The operands of arithmetic expressions can be reals, integers, or long integers. There are nine arithmetic operators:

Symbol	Meaning	Example	Numeric Value
+	Unary plus	+5	+5
−	Unary minus	−2	−2
^	Exponentiation	2 ^4	16
*	Multiplication	4*6	24
/	Division	5/2	2.5
MOD	Modulo	7 MOD 5	2*
DIV	Integer Division	7 DIV 5	1*
+	Addition	4+7	11
−	Subtraction	9−2	7

*Long integer only

Arithmetic Operator Precedence

In a simple expression like

4+8/2

you can't tell whether the answer should be 6 or 8, until you know the order (or *precedence*) to carry out the arithmetic. Your Apple gives the answer as 8, because it follows the following rules of precedence:

1. Any part of the expression enclosed in parentheses will be computed first, according to the following rules. Sets of parentheses grouped one inside another are evaluated from the inside out. First the innermost set is evaluated, then the second innermost, and so forth. For example, the expression 2 * (4 + 3) is equal to 14, while 2 * 4 + 3 = 11.

2. When the unary minus sign is used to indicate a negative number, for example

   −3+2

   your Apple will first apply the minus sign to its operand. Thus −3+2 evaluates to −1. It is perfectly legal to use the unary

minus sign, but it is always ignored as an operator. For example, the expression

```
+(-4)
```

evaluates to −4, not +4.

3. After applying all unary plus and minus signs, your Apple does exponentiation. The expression

```
4+3 ^2
```

is evaluated by squaring 3, and then adding 4. When there are a number of exponentiations, they are done from left to right, so that

```
2 ^3 ^2
```

is evaluated by cubing 2, and then squaring the result. Exponentiation cannot be used with long integers, or a

```
?TYPE MISMATCH ERROR
```

will be displayed.

4. After all exponentiations have been calculated, all of the multiplication, division, MOD, and DIV, operators are done, from left to right. These four operators have equal precedence. For example,

```
24/6/2
```

evaluates to 2. MOD, the modulo operator, evaluates the the long integer remainder of the division of the first operand by the second. For example,

```
7 MOD 5
```

evaluates to 2. DIV, the long integer division operator, evaluates the integer result of the division of the first operand by the second. For instance,

```
7 DIV 2
```

evaluates to 3. DIV and MOD can only be used with long integers.

5. All additions and subtractions are done last, from left to right. Addition and subtraction have equal precedence.

Logical Expressions

Logical expressions are also called relational expressions and Boolean expressions. They are similar to arithmetic expressions, but use different operators. Where an example of an arithmetic expression might be 2+2, 2=2 is an example of a logical expression. The value of the first expression is 4, while the value of the second expression is "true," because 2 does equal 2. Likewise, the value of the logical expression 2=3 is "false," because 2 does not equal 3. Arithmetic expressions evaluate to numeric values, whereas logical expressions evaluate to truth values.

Since BASIC doesn't understand the meaning of truth as such, but only the value of numbers, true and false have been assigned the numeric values 1 and 0, respectively. Thus 2=2 has a truth value of 1, 2=3 has a truth value of 0.

Any arithmetic expression with a non-zero value has a truth value of 1. Any arithmetic expression with a value equal to zero has a truth value of 0. For example, this means that that 2+2, when treated as a logical expression, has the numeric truth value of 1!

There are nine logical operators:

Symbol	Meaning	Example	Truth Value
=	Equal to	3=3	True
<	Less than	3<1	False
>	Greater than	7>4	True
<= or =>	Less than or equal to	5<=4	False
>= or = >	Greater than or equal to	8>=5	True
<> or ><	Not equal to	4<>4	False
AND	Conjunction	5 AND 0	True
OR	Inclusive disjunction	8 OR 3	True
NOT	Negation	NOT 4	False

Operator Precedence

A precedence list for operators of logical expressions is given below, listed in order of execution from highest to lowest priority. Successive operators of the same priority are executed from left to right.

```
( )
+ - NOT
^
* / MOD DIV
+ -
< > = >= = > <= = <> <>
AND
OR
```

Here are some additional examples of logical expressions, all true:

```
NOT (4=5)
(3-1) OR (NOT-4)
5<>3 AND (6 OR 4)
VPOS OR HPOS
(2+2)
-2
-(2+2)
```

```
NOT 0 AND 6
2* + 3/2
9.3
(3.414) * (1.707 + 3.414)
-(2.6)
3 + -(2.5+7)
3<>23444
CONV&(3) MOD CONV&(2)
```

Be careful when writing arithmetic expressions. The expression 3<2 is false, and the expression 2<1 is also false, but the expression 3<2<1 is "true"! (BASIC first tests the expression 3<2, and finds it false. False has the numeric value 0. It substitutes the value of 0 for 3<2. It then tests the expression 0<1, which is true. The last truth value found is the value assigned to the expression.)

It is possible to use logical operators in string expressions. For example, "alpha" < "beta" is true.

The ASCII values of the strings to be compared are tested one character at a time, and the first pair of non-identical characters determines the ranking of the strings. (See the Appendix ASCII CHARACTER CODES for a table of these codes and their numeric values.) While simple comparisons of uppercase letters present no problem, the result of comparing mixed case letters and digits is less obvious. In every case the decision will be based on the ASCII code values.

Here are some examples. These string logical expressions are all true:

```
"A"<"B"
"A"<"AA"
"Z">"Antidisestablishmentarianism"
"Anti"<"Antidisestablishmentarianism"
"A">"0"
"a">"A"
"="<">"
```

Triple comparisons, <>=, <=>, etc. are legal constructs in Apple Business BASIC. These *always* force the expression to evaluate to true. Think about it!

Functions

Most of the programs that you will write in BASIC will use a relatively small number of "tools" to solve a large number of different problems.

For example, many scientific and engineering problems require the use of logarithms or trigonometric functions for their solution. You could probably solve these with the use of tables built in to your program, or with some equally tedious means, but Business BASIC includes a set of tools, called functions, to make life easier.

A function takes one or more expressions, called *arguments*, performs some defined operation on them and *returns* a single value. The function's arguments are, with the exception of string functions (described in the section on String Functions), arithmetic expressions, and are always enclosed in parentheses following the function name. The returned value is substituted for the function name in the same way that the value of a variable is substituted for the variable name when used by a program.

You should realize that a function is not a statement itself, but must be used as part of a BASIC statement. Functions do no more than return values; so statements must tell BASIC what to do with the value returned by the function.

You can either use the functions built into Apple Business BASIC, or you can define and use your own functions. The functions built into BASIC perform certain standard operations such as trigonometric functions, removing fractions from real numbers, finding the absolute value of a number, and so on.

Values returned by functions have types, just as variables and constants have types. All built-in string functions return strings. All other functions return numbers of type real (except CONV% and CONV&, which return regular integers and long integers, respectively). You can assign a real value (returned by a function) to an integer variable, provided it is within the range −32768 to 32767, since reals are automatically converted to integers by BASIC for this purpose.

For example, the INT function rounds a fractional number to the next lowest whole number (real):

```
)X=-3.3 : Y=7. 95
)PRINT INT(X), INT(Y)
-4                    7
```

This is equivalent to using

```
)X=-4 : Y=7
```

When a function is included in an expression, BASIC first returns a value for the function, and then evaluates the rest of the expression using the returned value. For instance, you could write:

```
)WIDTH=*3.3
)A=5*INT(WIDTH)+3
```

and BASIC treats this as

```
)A=5*3+3
```

If you want to use or display the value returned by the function, you must include statements in your program to that effect. For example, if you want to display the sine of e, you must first use the SIN function, and then use a PRINT statement to display the returned value, as given below.

```
)E=2.718
)PRINT SIN(E)
.411038
)
```

Not all functions operate on numeric arguments: some of them work on strings. A string is a sequence of characters and can exist in three forms: as a string constant enclosed in quotes; as the content of a string variable (a variable whose name ends with the $ character), or as the value of a string expression.

A string expression's operands are strings and it returns a string value when evaluated. The only operator allowed within a string expression is the concatenation operator, +, which joins strings together. Here are three examples of string expressions:

Expression	Value
Message$	Whatever has been assigned to MESSAGE$
Message$+"123"	Value of MESSAGE$ with "123" appended to it.
"Ap"+"ple"+" Business BASIC"	"Apple Business BASIC"

An expression having strings as operands but containing any operators other than + is not a string expression, but a logical expression. For example, the value returned by

```
MSG1$ > MSG2$
```

is not a string, but a 0 for false or a 1 for true.

A string containing zero characters is called a null string and has a length of zero. A null string is written "", so the statement

```
)A$=""
```

assigns a null string to A$.

Business Basic Functions

The remainder of this chapter describes the built-in data-manipulating functions of Apple Business BASIC.

String Functions

The names of BASIC string functions that return string values end with the $ character.

Len

LEN returns an integer value equal to the length of the string expression, in the range 0 to 255. Example:

```
)PRINT LEN("ABCD")
4
)B$="Farm":PRINT LEN(B$+"House")
9
)
```

If the string expression contains more than 255 characters, the message

```
?STRING TOO LONG ERROR
```

is displayed.

STR$

STR$ evaluates a given arithmetic expression and returns the value as a string. For example:

```
)PRINT STR$(25/3)
8.33333
)PRINT STR$(100000000000)+"More"
1E+11More
)
```

VAL

VAL evaluates a given string expression and returns the value as a real or an integer number. Example:

```
)PRINT 10*VAL("1.3E4")
130000
)PRINT VAL("13"+"77")
1377
)
```

If any character of the string expression value evaluated is not a legal numeric character (leading spaces are acceptable), the

```
?TYPE MISMATCH ERROR
```

message will be displayed.

If the absolute value of the number represented by the value of the string expression is greater than 1E38, the message

```
?OVERFLOW ERROR
```

is displayed.

If the string expression value contains more than 255 characters, the message

```
?STRING TOO LONG ERROR
```

is displayed.

CHR$

CHR$ takes an arithmetic expression as an argument and returns a one-character string corresponding to the ASCII value of the evaluated arithmetic expression.

```
)PRINT CHR$(66.8)
C
)R$="68":PRINT CHR$(VAL(R$))
D
)
```

The value of the arithmetic expression is rounded to the nearest whole number if it is a real. It must be in the range 0 to 255, or the message

?ILLEGAL QUANTITY ERROR

is displayed. See the appendix ASCII Character Codes.

ASC

ASC returns the ASCII character code corresponding to the first character of the given string expression. If the string expression value is a null string, then the value −1 is returned. Example:

```
)PRINT ASC("BEEP")
66
)d$="BEEP" : PRINT ASC(d$+"S")
66
)
```

HEX$

HEX$ returns as a four-character string the hexadecimal (base 16) equivalent of the value of the given arithmetic expression. For example:

```
)PRINT HEX$(780)
030C
)PRINT HEX$(−1024)
FC00
)
```

The value of the given arithmetic expression is rounded down to the nearest whole number if necessary. It must be in the decimal range −65535 to +65535, otherwise the message

?ILLEGAL QUANTITY ERROR

is displayed.

TEN

TEN returns the decimal (base 10) equivalent of the *last* four characters of the given string expression. The value returned will be in the range −32768 to 32767. Example:

```
)PRINT TEN("HEXNUM 030C")
780
)PRINT TEN("CCCC")
−13108
)
```

The last four characters of the value of the given string expression must represent a hexadecimal value; if not, the message

```
?ILLEGAL QUANTITY ERROR
```

is displayed.

LEFT$

LEFT$ returns a string composed of the leftmost characters of the given string expression. The length of the string returned is defined by an arithmetic expression that immediately follows the string expression in the LEFT$ argument list. For example:

```
)PRINT LEFT$("Appleskin",5)
Apple
)PRINT LEFT$("Sparkling",3)
Spa
)
```

If the value of the arithmetic expression exceeds the length of the string expression value, all of the characters of the string expression value are returned.

If the string expression value contains more than 255 characters, the message

```
?STRING TOO LONG ERROR
```

is displayed. The value of the arithmetic expression is rounded down to the nearest whole number if necessary. It must be in the range 1 to 255, or the message

?ILLEGAL QUANTITY ERROR

is displayed.

RIGHT$

RIGHT$ returns a string composed of the rightmost characters of the given string expression. The length of the string returned is defined by an arithmetic expression that immediately follows the string expression in the RIGHT$ argument list. For example:

```
)PRINT RIGHT$("Appleskin" + "Ware", 8)
skinWare
)B$=RIGHT$("Fruitbat", 3) : PRINT B$
bat
)
```

If the value of the arithmetic expression exceeds the length of the string expression value, all of the characters of the string expression value are returned.

If the string expression value contains more than 255 characters, the message

?STRING TOO LONG ERROR

is displayed. The value of the arithmetic expression must be in the range 1 to 255, or the message

?ILLEGAL QUANTITY ERROR

is displayed.

MID$

MID$ returns a substring of a given string expression. You must specify exactly where in the value of the string expression the

substring should begin, by following the string expression with an arithmetic expression. For example, if you want to retrieve the substring "keeping" from the string "Bookkeeping", use

```
)PRINT MID$("Bookkeeping",5)
```

because the first character in "keeping" is the fifth character in "Bookkeeping".

You may optionally specify the exact number of characters to be retrieved from the string expression value, by including a second arithmetic expression. For instance, if you only want the four-character substring "keep" from "Bookkeeping", use

```
)PRINT MID$("Bookkeeping",5,4)
```

because "keep" is four characters in length.

If the value of the first arithmetic expression exceeds the length of the string expression value, then a null string is returned. If the value of the second arithmetic expression specifies a greater number of characters to be retrieved from the string expression value than exist, all of the characters from the position specified by the value of the first arithmetic expression to the end of the value of the string expression are returned. For example,

```
)PRINT MID$(A$,255,255)
```

will display one character if the length of A$ is equal to 255; otherwise a null string is displayed.

If the string expression value contains more than 255 characters, the message

```
?STRING TOO LONG ERROR
```

is displayed. If the value of either arithmetic expression is outside the range 1 to 255, then the message

```
?ILLEGAL QUANTITY ERROR
```

is displayed.

Try to figure out what the following program will do, and then run it:

```
15 A$="ABCD"
25 FOR Loop1 = 1 TO 4
35 FOR Loop2 = 1 TO 4
45 PRINT MID$(A$,Loop1,Loop2)
55 NEXT Loop2, Loop1
```

Were you right?

INSTR

INSTR returns the position of the beginning of a specified substring within a given string. For instance, if you want to know where the substring "ai" is in the string "Rain in Spain on the plain", use

```
)PRINT INSTR("Rain in Spain on the Plain", "ai")
2
)
```

The first occurrence of "ai" is at character position 2. The substring can be the value of any string expression. Note that "ai" occurs at two other places within the string. To find their positions, you must include an optional arithmetic expression after the string expression to begin the string search at a position other than its first character. For example:

```
)PRINT INSTR("Rain in Spain on the plain", "ai", 9)
11
)
```

The arithmetic expression (9 in the example), specifies the character position at which the search should begin. If no the arithmetic expression is specified, the search begins with the first character position of the string expression. If the substring is not found within the string expression then the value 0 is returned.

If the arithmetic expression is greater than the length of the string expression or less than 1, then the

?ILLEGAL QUANTITY ERROR

message is displayed.

SUB$

SUB$ lets you replace any part of a string with a specified substring. The string to be changed can be any string variable, and the substring may be the value of any string expression. You must specify the first character in the string to be changed, by following the string with an arithmetic expression. For instance

```
)F$="Hardware" : SUB$(F$,1)="Soft" : PRINT F$
Software
)
```

In this example the new string (Soft) replaces part of the string (Hardware) contained in the string variable F$. The replacement begins at the first character of F$, and continues until all the characters of the substring "Soft" have been placed in their new home.

You may optionally include a second arithmetic expression to specify the number of characters in the substring to be used in changing the original string. For example

```
)F$="Hardware" : B$="Soft" : SUB$(F$,1,2)=B$ : PRINT F$
Sordware
)
```

Additional examples:

```
)A$="ABCDEFG":B$=A$:C$=B$:D$=C$:E$=D$:F$=E$
)SUB$(A$,3)="**":PRINT A$
AB**EFG
)SUB$(B$,3,1)="**":PRINT B$
AB*DEFG
)SUB$(C$,3,100)="**":PRINT C$
AB**EFG
)SUB$(D$,3)="**********":PRINT D$
```

```
AB**********
)SUB$(E$,3,9)="*********":PRINT E$
AB*********
)SUB$(F$,3,2)="*********":PRINT F$
AB**EFG
```

Numeric Functions

Numeric functions may be used in either immediate or deferred execution. The argument to all numeric functions must be an arithmetic expression.

All floating-point arithmetic in Business BASIC is done with 32-bit precision, and this sets limits on the accuracy of the results returned by numeric functions. For most work the error generated will not be a problem (or even detectable!), but you should be aware that there is a limit.

SIN

Returns the sine of an angle given in radians. For example,

```
)PRINT SIN(2.718)
.411038
)
```

COS

Returns the cosine of an angle given in radians. For instance,

```
)PRINT COS(1.571)
 -2.03673E-04
)
```

TAN

Returns the tangent of an angle given in radians. For example,

```
)PRINT TAN(3.141)
 -5.92653E-04
)
```

ATN

Returns the arctangent, in radians, of the given argument. The value returned represents an angle in the range $^{-}$pi/2 to $^{+}$pi/2 radians. For instance,

```
)PRINT ATN(.3456)
.33275
)
```

INT

Returns the largest whole number value *less* than or equal to the argument value. For example,

```
)PRINT INT(3.3)
3
)X =INT( -3.3) : PRINT X
 -4
)
```

Notice that we said "whole number" in the last sentence, and deliberately avoided the term "integer". This is because the INT function actually returns a real number.

RND

Returns a random real positive number less than 1.

RND generates a new random number each time it is used if the argument value is greater than zero.

If the argument value is negative, RND generates the same random number each time it is used with the same argument. If a given negative argument is used to generate a random number, then subsequent random numbers generated with non-negative arguments will follow the same sequence each time. A different random sequence is initialized by each different negative argument. This is particularly helpful in debugging programs that use RND.

If the argument value is zero, RND returns the most recent previous random number generated (the CLEAR and NEW statements do not affect this). Sometimes this is easier than assigning the last random number to a variable in order to save it. For example,

```
)10 INPUT X : PRINT RND(X);" ";RND(X);" ";RND(X) : GOTO
10
)RUN
?3
.73643 .21479 .53754
?3
.23458 .65986 .54193
? ¯4
.95754 .95754 .95754
?0
.95754 .95754 .95754
?
?BREAK IN 10
)
```

To get a random whole number, for example, between 50 and 100 inclusive, combine the INT and RND functions in one expression:

```
)PRINT INT(RND(8)*51)+50
```

SGN

Returns ¯1 if the argument value is negative, returns 0 if the value of the argument equals 0, and returns 1 if the argument value is positive. For instance,

```
)PRINT SGN( ¯234)
 ¯1
)PRINT SGN(2496 +234)
1
)PRINT SGN(5E4-5E4)
0
)
```

ABS

Returns the absolute value of the argument; in other words, the value of the argument if it is positive, 0 if the value is zero, and the negative of the argument value if it is negative. For example,

```
)PRINT ABS(345)
345
)PRINT ABS(24-363)
339
```

SQR

Returns the positive square root of the argument value. For example,

```
)PRINT SQR(3   2 +4   2)
5
)
```

EXP

Raises e (to 6 places, e =2.718282) to the power indicated by the argument value. For example,

```
)PRINT EXP(3)
20.0855
)
```

LOG

Returns the natural (base e) logarithm of the argument value. For instance,

```
)PRINT LOG(20.0855)
3
)
```

CONV&

Evaluates the given argument, and returns a long integer value. For example,

```
)PRINT CONV&(2178-7954)
 ⁻5776
)PRINT CONV&("4.214")
4
)
```

If the argument is a string, then the effect is the same as using VAL followed by CONV& (see the chapter STRINGS AND STRING FUNCTIONS for an explanation of the VAL functon). The value returned must be within the range 92233720368577580 7 to ⁻9223372036854775808 (⁻9223372036854775807 from the keyboard), or a

```
?OVERFLOW ERROR
```

message is displayed.

CONV

Evaluates the argument, and returns a real value. The value may be assigned to a regular integer. The conversion from real to integer is automatic in the latter case. If CONV is used with a string expression, the effect is the same as with the VAL function. For instance,

```
)G& =234234 : H& =523523 : PRINT CONV(H&-G&)
289289
)
```

CONV$

Evaluates the given expression, and returns a string value. For example,

```
)D%=345 : A%=453 : PRINT "a"+CONV$(D%*A%)+"z"
a156285z
)
```

CONV%

Evaluates the argument and returns an integer value, rounding off to the nearest whole number. The value returned must be within the range ⁻32768 to 32767, or

```
?OVERFLOW ERROR
```

is displayed. For instance,

```
)PRINT CONV%(423.94)
424
)A& =7656 : B& =364 : PRINT CONV%(A&/B&)
21
)
```

Defining Your Own Functions

You may define your own functions to perform operations that are not available from by any of the built-in functions. Once a function is defined, it is also available in immediate execution, as long as the defining program is still in memory, and the DEF FN statement has been executed. A user-defined function may not be defined and used in immediate execution without having been defined first in a deferred execution program.

If your program defines a new function and a CHAIN statement calls a new program, the new program may not use the original definition. If you try to do this an error will result.

DEF FN

The DEF FN statement is used to let the user define functions. The name of the function and its argument must follow the reserved word FN. The argument must be a single real variable whose name in the function is not in any way connected to the possible use of that name elsewhere. The statement must conclude with a *defining expression*. The defining expression may be any legal arithmetic expression, and it may contain the argument in the expression as well as any other

integer or real variables. For example, this statement defines a function named RECIP, with X as the argument, and 1/X as the defining arithmetic expression:

```
)55 DEF FN RECIP(X) = 1/X
```

More examples:

```
)10 DEF FN NEGATE(X) = -X
)20 DEF FN SWORDED.4(C) = INT(RND(3)*100)
)30 DEF FN M5BY7.MAT(DED) = DED*LOG(33)-ABS(F%)
```

Using a Defined Function

Once a function has been defined, it may be used anywhere that an arithmetic expression can be used. Using a function requires that its name be preceded with the reserved word FN. Here is an example of how you might use the NEGATE function:

```
)20 A = FN NEGATE(2)
```

In this example, the real variable A is assigned the value -2. Here is another example:

```
)30 PRINT 4 * FN NEGATE(A) * 3
```

If the value of A was -2, then 24 would be displayed on the screen.

Remarks About Defining Functions

The arithmetic expression used to define a function may refer to any real or integer variable. For instance, consider the following sequence:

```
)5 B=3
)10 DEF FN Foo(A)=A*C
)15 PRINT FN Foo(B) : REM displays 0 because C is
 undefined
)20 C=5                    : REM C now exists and equals 5.0
)25 PRINT FN Foo(B) : REM displays 15
```

The argument to the function need not appear in the defining expression. In this case, the function's argument is ignored in evaluating the expression, but the function's argument is evaluated to make sure it is of type real, so it must be something legal. For instance the formal argument ZILCH in the next example is real even though it is not used in the expression:

```
)10 DEF FN BAZ(Zilch) =2 +2
```

Functions may be redefined whenever you want to.

If a function is used in a statement before being defined, the

```
?UNDEF'D FUNCTION ERROR
```

message is displayed.

Only functions of type real can be defined. Integer variables may be used in the expression, but they are automatically converted to reals by BASIC.

A DEF FN statement uses 1 byte per character in the function name, 2 overhead bytes, and 6 bytes for information about the function in addition to the memory space required to store the arithmetic expression itself.

3

Apple Business BASIC I/O

3

Apple Business BASIC I/O

Introduction

Most of your time spent working with the Apple III, especially program development, demands that information be sent to the Apple III through the keyboard and received from the screen.

This chapter describes the tools that are provided by Business BASIC for this work.

Your Apple's keyboard and display screen are collectively termed the *console*. Console *input* and *output* (or *I/O*) statements allow you to control when, where and how specific characters are displayed on the screen or read from the keyboard.

A *tab action*, in the following descriptions, means moving the cursor to the beginning of the next *tab field* on the screen.

The screen is divided by five 16-character-wide tab fields. If you have typed (or the Apple III has printed) a character at the last character position of tab fields one through four, tabbing will move the cursor to the beginning of the second field following that character.

If the text window is less than 80 characters wide, TAB may send the cursor down one or more lines.

Displaying Text on the Screen

PRINT

PRINT is used to display text on the screen. A list of the items to be displayed must follow the reserved word PRINT. The item list may include any expression, comma, semicolon, TAB specification, or SPC specification.

Expressions in PRINT statements are evaluated and their values displayed on the screen. If there are several expressions, their values are displayed in sequence. A PRINT statement without an item list causes the cursor to move to the beginning of the next screen line. If a comma separates two expressions, a tab action separates their values on the screen; if a semicolon separates them, the second value is displayed after the first with no intervening spaces. For instance:

```
)PRINT "In";"divisible"; 543, 598/754.42
Indivisible543 .792662
)
```

Following the last expression in a PRINT statement, there may be a semicolon, comma, or nothing. If there is nothing after the last expression, the cursor moves to the beginning of the next screen line. A comma causes a tab action. A semicolon leaves the cursor in the position immediately following the last character displayed.

Here are a few examples of PRINT statements:

```
)PRINT "X"
X
)X&=3 : PRINT X&
3
)PRINT 10, 20
10        20
)PRINT 10;" "; 20
10  20
```

```
)A$="Apple" : PRINT A$
Apple
)A$="Johnny" : B$="Apple" : C$="seed" : PRINT A$;" ";B$;C$
Johnny Appleseed
```

Some expressions can run together in a PRINT statement without either commas or semicolons to separate them. BASIC will work very hard to figure out where one expression ends and the next one begins. If it succeeds, the effect is the same as using a semicolon. If it doesn't succeed, either a

?SYNTAX ERROR

appears, or wrong values are displayed. You should use the semicolon to avoid confusion.

)PRINT A$+B$

will give

?STRING TOO LONG ERROR

message if the combined length of the two strings is greater than 255. However, you can display the apparent combined string using

)PRINT A$;B$

without worrying about its length.

When you type in a program, a question mark can replace the reserved word PRINT; it lists as PRINT. You don't save any memory by using abbreviations—only typing time.

TAB and SPC

You can insert spaces into text displayed by a PRINT statement by putting a TAB or SPC specification into the PRINT statement.

A TAB or SPC can be inserted immediately before or after any expression, comma, or semicolon in a PRINT statement's expression list.

TAB and SPC are followed by an arithmetic expression enclosed in parenthesis. The integer value of the expression following the word TAB defines the number of spaces from the left margin of the text window to begin printing text. If you specify an expression that is less than the number of the current print position, no spaces will be inserted before the next character to be printed.

The integer value of the expression following SPC define the number of spaces to be inserted after the last-printed character. Examples of TAB and SPC are given below.

```
)PRINT "Great"; TAB(8); 347
Great 347
)PRINT "Underhanded"; TAB(8); 553
Underhanded553
)PRINT "A"; SPC(1); "B"; SPC(2); "C"
A B C
)PRINT "D"; TAB(5); "E"; SPC(5); "F"
D E F
```

Each SPC statement is limited to a maximum value of 255, but you can place as many spaces as you wish by stringing together a series of SPC statements like this:

```
)SPC(250)SPC(139)SPC(255)
```

The PRINT...USING statement gives you greater control over text on the screen. See the section on formatting text output.

Entering Data

INPUT

INPUT accepts numbers or text typed at the keyboard and assigns them to variables specified in the INPUT statement. INPUT can be used in deferred execution only. For instance, if you wanted users of your program to input their age in years, you could use

```
)20 PRINT "Enter your age in years"
)25 INPUT AGE
```

When INPUT is executed in this form, BASIC automatically displays a question mark on the screen, and waits until you type something (in this case, your age). For example

```
)RUN
Enter your age in years
?38
)
```

You may optionally include a string in an INPUT statement, like this:

```
)20 INPUT "Enter your age in years"; AGE
```

The optional string must be a sequence of characters in quotation marks, followed by a comma or semicolon; it cannot be a string variable or expression. When the optional string is present, it is displayed exactly as specified; no question mark, spaces, or other punctuation are displayed after the string. You can use only one optional string.

After the optional string, one or more variable names must be present, separated from each other by commas. Like this:

```
1000 INPUT "Please type three words"; A$, B$, C$
```

INPUT expects you to type a number or string for each of the variables in the INPUT statement. The numbers and strings are

expected in the same order in which the variables occur in the statement. For a numeric variable, a number is expected, and for a string variable a string is expected.

The numbers and strings that you type in response to an INPUT statement must be separated from each other by commas or returns. If you use a return and another number or string is still expected, BASIC will display not one, but two question marks on the next screen line. The numbers and strings are assigned to the variables, in sequence. For example:

```
1000 INPUT "Please type four numbers: ";A,B,C,D
```

This statement expects you to either enter four numbers separated by commas like this:

```
100, 200, 300, 400 (followed by return)
```

Or you can do it this way, with the numbers separated by returns:

```
Please type four numbers: 100
??200
??300
??400
```

Execution can be halted during an INPUT statement, if you type CONTROL-C as the first character, followed by return.

Entering Numbers

For arithmetic variables, INPUT will accept only numbers. Remember that the characters +, space, -, period, and E are all legitimate parts of numbers. Any spaces in numbers are ignored.

Input which is not a legitimate number (such as a a string or a return) will cause the message

```
?REENTER
```

to be displayed and the INPUT statement to be re-executed from the beginning.

If an input number is not the same type as the corresponding variable, it will be automatically converted to the same type; if necessary, the number will be rounded. For example,

```
)5 INPUT G%
)10 PRINT G%
)RUN
?6.87
7
)
```

Quotation Marks and Commas in String Input

Quotation marks and commas in string input are handled differently, depending on the position of the string variable in the INPUT statement to which the string will be assigned.

If the string variable is the last (or only) variable in the INPUT statement, then any quotation marks or commas are treated as ordinary characters in the string.

If the string variable is not the last variable in the INPUT statement, then the comma and quotation mark characters are treated specially. The comma separates strings in the same manner as it separates numbers. By enclosing a string in quotation marks you can include commas in it without trepidation. The quotation marks, as usual, are not considered part of the string. The closing quotation mark of a string can be omitted if the RETURN key is used to end the string. If a quotation mark is not the first character typed, then a quotation mark can be included anywhere else in the string without being treated specially. The effects of quotation marks are best explained by example (remember that a program can be halted by typing CONTROL-C as response to an INPUT statement):

```
)10 INPUT X$, Y$ : PRINT X$, Y$ : GOTO 10
RUN
?is, is
is       is
“is, “is is, “is
?REENTER
?“is,” ’is’
```

```
is          'is'
?is", is"
is"         is"
?""is, ""is
?REENTER
?is"," is""
is""        is""
?"is"," "is""
?REENTER
?"is,isn't," "is,isn't"
is,isn't        "is, isn't"
?
?BREAK IN 10
)
```

Null Strings

If the RETURN key and no other characters are typed when a string response is expected, the response is interpreted as a null string.

The open-apple key sets the high-order bit of any characters typed from the keyboard, in effect adding 128 to all ASCII values sent from the keyboard. Since this bit is ignored by BASIC, it does not affect keys typed in response to an INPUT statement.

GET

GET is used to assign a single alphanumeric character from the keyboard to a specified variable in your program, without displaying it on the screen and without requiring that the RETURN key be pressed. The specified variable must follow the reserved word GET. For example,

```
)100 PRINT "Press any key";
)110 GET PRESS$
)120 PRINT ." You pressed the ";PRESS$;" key!"
)130 GOTO 100
```

CONTROL-C is treated like any other character; it does not interrupt program execution.

Note that typing a return character when using GET to enter numeric input, or any non-numeric character causes the variable to be assigned a value of zero.

Because of this limitation, it is good programming practice to get numbers by using GET with a string variable and convert the resulting string to a number with the VAL function. In most cases it is more convenient to INPUT a number instead of getting it.

The GET statement can only be used with deferred execution.

If you display the value of a string variable that contains a control character, that control character can affect operation of the .CONSOLE device driver. For example, if in the program example above, you type CONTROL-C when asked to

Press any key

the text window will be reset when PRESS$ is displayed by line 120. (See your Standard *Device Drivers* manual).

If your program that uses GET was called up by an EXEC file, the input for the GET statement will be taken from the EXEC file instead of from the keyboard.

Storing Data Within Your Program

DATA and READ statements allow you to store data within a program itself, allowing you to use it to fill arrays, create strings, or make other use of the stored information. Although READ and DATA are not true console I/O statements, they are included here because their effect is similar to that of INPUT or GET statements.

DATA and READ

The DATA statement is used to provide a list of *data elements* to be read by a READ statement. DATA statements do not have to precede READ statements in a program. Data elements can be strings, reals, integers, and long integers. String data elements do not have to be bounded by quotation marks. For example,

```
)1220 DATAWHEN, 5, -4, "EQUALS", 1.000
```

The DATA statement can only be used in deferred execution, or an

```
?ILLEGAL DIRECT ERROR
```

results.

A data element list consists of all of the elements in all of the DATA statements in a program. READ statements are used to read from a data element list and to assign the values to variables. When the first READ statement in a program is executed, the first variable in the READ statement is assigned the value of the first data element in the data element list. The second variable in the READ statement (if there is one) is assigned the second element in the data list, and so on. For instance,

```
)30 READ A$, B%, C&, D$, E
)40 PRINT A$;" ";B%; C&;" ";D$;" ";E
)500 DATAWhen, 5, -4, "Equals", 1.000
)RUN
When 5-4 Equals 1.000
)
```

Data elements assigned to arithmetic variables generally follow the same rules that numbers assigned to arithmetic variables by INPUT statements follow.

There are two types of strings that may be used as data elements: Literals (those without enclosing quotes), and quote-enclosed (or "quoted") strings.

If CONTROL-C is a data element, it does not halt execution of the program even when it is the first character of an element. With this exception, data elements read into string variables follow the rules for INPUT responses assigned to string variables:

- Either literal or quoted strings may be used.
- Quotation marks appearing within a quoted string cause the

  ?SYNTAX ERROR

  message, but all other characters, including commas are accepted as characters in that string. The entire string may be enclosed within quotation, however.
- The colon and comma are accepted only in quote-enclosed strings; CONTROL-M (the RETURN character) is never accepted.

If a READ statement attempts to assign a string data element value to an arithmetic variable, a

?SYNTAX ERROR

occurs.

Variables are assigned values of zero or null string (depending on the variable's type) when any of the following conditions are met when an attempt is made to read a data element:

- A comma is the first non-space character following the reserved word DATA.
- There is no data element between two commas.
- The last character in a DATA statement is a comma (when the comma is being read as a data element).

So when this statement is read:

```
100 DATA ,
```

it can result in up to two element assignments consisting of zeros or null strings.

When variables in a READ statement have been assigned values from the data element list, BASIC leaves a *data list pointer* immediately following the last element read. The next READ statement executed (if any) begins using the data list from the pointer position. A RUN, CLEAR or RESTORE statement moves the pointer back to the first element in the data list.

When all of the elements in a DATA statement have been read, the pointer moves on to the next DATA statement with a higher line number, and reading continues with the first element listed. An attempt to read past the end of the data list produces the message, for example,

```
?OUT OF DATA ERROR IN 3400
```

3400 being the line number of the READ statement that asked for additional data.

In immediate execution, you can only read elements from DATA statements in a program that is currently in memory; that is a program that has been typed in, loaded, or run. If no DATA statement is in memory, the message

```
?OUT OF DATA ERROR
```

is displayed. Execution of a READ statement does not reset the data list pointer back to the first element in the DATA list after having read the last element in the list.

DATA may not appear as the first four characters of a variable name since the variable name will then be parsed as a DATA statement. Also note that everything after DATA in a given line will be parsed as a data list. You must continue to the next higher-numbered line to continue the program.

You may not follow DATA with any executable statements on the same-numbered line of your programs. Anything following a DATA statement until the next line number is considered to be part of a data list.

RESTORE

RESTORE moves the data list pointer back to the beginning of the data list. This allows you to read the same data more than once. There are no parameters or options associated with RESTORE.

Formatting Information

In a very broad sense, the only thing that your computer can do for you is manipulate information. Even games can be described in terms of information handling.

Business BASIC provides you with several tools, described below, for displaying the end result of the information manipulation performed by your program.

PRINT USING and PRINT# USING

PRINT USING and PRINT# USING, (collectively referred to as PRINT(#) USING), allow you to precisely control the format of information displayed on the screen (PRINT USING) or written to files (PRINT# USING).

The information format is controlled by printing fields defined by format specifications (or specs). A *printing field* is a template outlining the way that information may be laid out, either for screen display or to a file for later printing. The information being formatted may be either numeric or string data.

Specs may be included with the PRINT(#) USING statement in the form of a string expression, or they may be part of an IMAGE statement elsewhere in the program. If an IMAGE statement is used, the line number containing it must be included in the PRINT(#) USING argument list. Examples of legal usage of PRINT USING is given below.

```
)10 PRINT USING 100; A$, B%, C
)100 IMAGE 6A, 5#, #.6Z4E

)1 B$="6A, 5#, #.6Z4E"
)10 PRINT USING B$; A$, B%, C

)10 PRINT USING "6A, 5#, #.6Z4E"; A$, B%, C
```

All three examples given above do the same thing; display the values of the variables A$, B%, C, using the specs supplied by the user.

The expression list of a PRINT (#) USING statement is somewhat different from the expression list of a PRINT or PRINT# statement (described in the chapter File I/O). Commas are required to separate expressions in the PRINT(#) USING list, but they do not cause the cursor to move to the next tab field.

A semicolon may be used at the end of the expression list to suppress a carriage return, however, just as used in PRINT and PRINT#.

If the number of expressions in the expression list exceeds the number of specs, the spec list is used again from the beginning until all expressions in the expression list have been evaluated and used. A single spec will be used by all expressions in the list. This is especially useful if you have a number of values to be displayed using the same spec; you only have to write the spec once.

The following errors can occur when a PRINT(#) USING statement is executed:

If the USING clause references an IMAGE statement, and the IMAGE statement does not exist, the

?UNDEF'D STATEMENT ERROR

message is displayed.

If the USING clause references a string variable or contains a string, and the string value is null, a

?SYNTAX ERROR

occurs. An IMAGE statement containing no specs also results in the same error message.

If a

?SYNTAX ERROR

occurs in an IMAGE statement, the message gives the line number of the PRINT(#) USING statement, not that of the IMAGE statement.

If the type of an expression does not match the type of its spec—i.e., string with numeric or vice versa—a ?TYPE MISMATCH ERROR message is displayed.

IMAGE

An IMAGE statement contains a sequence of specs separated by commas. Each spec corresponds to an expression in a PRINT(#) USING statement, and controls the printed or displayed format of the expression value. (Exception: a literal spec does not correspond to an expression.) The IMAGE statement can only be used with deferred execution, and it must be the only statement on the line. Here is an example of an IMAGE statement with three specs:

```
10 IMAGE +5#.3#,10X,-#.5#4E
```

The following example show the output generated by the IMAGE statement in line 10 with two specs:

```
10 IMAGE +###.###, 3"."
100 PRINT USING 10; 1.5, 3.14159, 172.9, 5
)RUN
+ 1.500...+ 3.142...+172.900...+ 5.000...
)
```

Output Format Specifications

The format of your program output may be defined by format specifications (or *specs*). There are three kinds of specs:

- A *string spec* controls the format of a string value in a PRINT(#) USING statement.

- A *literal spec* inserts one or more spaces, returns, or copies of a specified string into the text displayed by the PRINT(#) USING statement.

- A *numeric spec* controls the format of a numeric value displayed by a PRINT(#) USING statement. There are three varieties of numeric specs: fixed-point specs (or *fixspecs*), scientific notation specs (*scispecs*), and engineering notation specs (*engrspecs*).

The String Spec

A string spec defines the field format and width for a string value, and specifies whether the string value is to be left-, right-, or center-justified within the field. If the string has fewer characters than the field allows, the empty positions are filled with spaces.

A string spec defines a left-justified field by using "A", a right-justified field with an "R", and a centered field with a "C". The width of the field may be set either by using the number of characters to be used in the field, or by preceding the spec with a positive integer equal to the length in characters of the field.

For example, a six-character, right-justified field could be defined either by RRRRRR or 6R. The specs 9C and CCCCCCCCC produce the same result, a nine-character field with its string value centered in the field.

Repeat factors: The numbers in the above examples are called repeat factors. A repeat factor can be any positive integer from 1 to 255; it affects only the single character immediately following it. You could also use 5AA or A5A instead of 6A, etc. A repeat factor greater than 255 causes an

?ILLEGAL QUANTITY ERROR

message.

Here's an example of using string specs to format string output into three columns:

```
10 IMAGE 15A, 15C, 10R
15 PRINT
100 PRINT USING 10;"COMPOSER", "TITLE", "KEY"
200 PRINT USING 10; "GRINSZ", "SONATA FOR
    HARP","F SHARP"
300 PRINT USING 10; "RIBBITT", "WATER SONG", "E
    FLAT"
400 PRINT
```

The IMAGE statement in line 10 defines three fields: a 15-character field with a left-justified string in it, a 15-character field with a string centered in it, and a 10-character field with a right-justified string in it. Note that these add up to 40 characters, so they will fill one half of a line on the screen. This program runs as follows:

```
)RUN

COMPOSER          TITLE                KEY
GRINSZ            SONTA FOR HARP       F SHARP
RIBBITT           WATER SONG           E FLAT
```

Changing the string spec in line 10 of the program fragment given above has the following effect:

```
)10 IMAGE 5A, 5C, 5R

)RUN

COMPOTITLE KEY
GRINSSONATF SHA
RIBBIWATERE FLA

)
```

All string values exceeding the new length of five characters are truncated to fit in the newly-defined fields.

Element	Function
A	Left justify field.
C	Center field.
R	Right-justify field.

Table 3-1. String Spec Elements.

The Literal Spec

A special kind of spec called a literal spec does not format the value of an expression: instead it inserts a fixed number of spaces, a fixed number of returns, or a fixed string into the output.

In a literal spec, an X means print a space. For example,

```
4X
```

inserts four spaces into the output.

A slash (/) character means print a return. For example, the spec

```
2/
```

inserts two returns.

When a repeat factor is placed in front of a literal spec string, it affects the entire string. For example, the spec

```
3"AB"
```

inserts

```
ABABAB
```

A given literal spec value can only insert one kind of thing into the output. For example, two spec values are needed to insert three spaces followed by five asterisks:

```
3X,5""
```

or

```
"*****"
```

Element	Function
X	Prints space.
/	Prints RETURN.
”	Encloses literal string to be printed.

Table 3-2. Literal Spec Elements.

Numeric Formatting

Any numeric value, regardless of its type, can be formatted by PRINT(#) USING statements in any of three ways: fixed-point, scientific, or engineering format. There is a separate kind of numeric spec for each of these outputs.

With slight differences in usage, all three numeric formats use *digit specs*. Digit spec characters and their function are given in the table below.

Character	Function
#	Reserves one numeric digit position. Leading zeros suppressed.
&	Reserves position for digit or comma. At least five digit positions must be reserved to the left of the decimal point.
Z	Reserves one numeric digit position. Leading zeros are printed.

Table 3-3. Digit Spec Characters.

The FIXSPEC

The fixed-point specification (fixspec) controls the output format of fixed-point numbers. Fixed-point numbers are any numbers displayed without exponents including integers, long integers, and

real numbers. Here is an example of a simple fixspec appearing in a PRINT USING statement:

```
)PRINT USING "+####.###"; 3.14159
+ 3.142
```

The spec given above consists of a + followed by a digitspec. A digitspec reserves positions in the field for the digits of a numeric value, and may also reserve a position for the decimal point. Three different characters — #, Z, and & — can be used to form a digitspec, and repeat factors are allowed.

The # character reserves one numeric digit position. Leading zeros (if present) are replaced with spaces. For example:

```
)PRINT USING "+6#.3#"; 09999
+ 9999.000
```

A "Z" reserves one numeric digit position, just like a "#", however, leading zeros are printed. For example:

```
)PRINT USING "+6Z.3Z"; 09999
+009999.000
```

An "&" character reserves one position for a numeric digit or comma. Commas are inserted after every third digit left of the decimal point. Commas are included in the character count and leading zeros are replaced with spaces. At least five digit positions must be reserved to the left of the decimal point when using &. For example:

```
)PRINT USING "+6&.3&"; 09999
+ 9,999.000
```

The examples above all show a decimal point with digits to the left and to the right. However, no decimal point, or a decimal point with nothing to the left, or a decimal point with nothing to the right may legally be specified. Remember that integer expressions can have no fractional part. If you specify a fixspec with a fractional part and apply it to an integer expression, only zeros will appear to the right of the decimal point, unless the SCALE function is used.

The value to be displayed is rounded-off, if necessary, to fit the number of digits specified to the right of the decimal point. However, if the number exceeds the number of digits specified to the left of the decimal point, the entire field is filled with exclamation points.

The characters &, #, and Z may be mixed in a digitspec, but those appearing to the left of the decimal point yield to the character with the highest precedence. The precedence of these characters is, in order, & followed by #, followed by Z. If an & appears in a digit spec, the formatted output is going to have commas inserted and leading zeros suppressed.

The digitspec is only part of a fixspec. In all of the examples we have shown a + as part of the fixspec to reserve a position for the sign of the number, and in almost all cases you should do this. The + causes the sign to be printed in all cases, while a - causes the sign to be printed only if the number is negative; a space is printed if it is positive. The sign of the number can also follow the last digit.

For financial output, a $ reserves a character position for a dollar sign. A pair of asterisks (**) causes asterisks to be printed instead of leading spaces when there are unused digit positions in the output field. For example,

```
)PRINT USING "**+6#.3#"; 09999
+**9999.000
)PRINT USING "**$6#.3#-"; 09999
***$9999,000
)PRINT USING "+$6#.2#"; 09999
+ $9999.00
```

Note that the **, if used, must be the first thing in the fixspec and cannot be used if Z is used for the digitspec, because Z leaves no unused digit positions. The dollar sign can come next, or the number sign (+ or -).

If you do not reserve a character position for the sign, and the sign of a value in the PRINT(#) USING statement is negative, the sign will be displayed in the rightmost unused character position if there is one. But if there is no unused position, and the sign is negative, the entire field will be filled with exclamation points indicating that the number of digits exceeds the number of places specified to put them! Therefore you should normally reserve a position for the sign in all numeric specs.

In the forms of fixspec shown above, the dollar sign and the number sign go exactly where positioned in the spec. But you can also cause them to be printed in the rightmost available position(s) by using $$ or ++, or in the leftmost position by using --. For instance,

```
)PRINT USING "$$+6#.3#"; 09999
$+9999.00
)PRINT USING "++6#.3#"; 09999
+9999.00
)PRINT USING "$--6#.3#"; 09999
$9999.00+
```

You can also place the sign of the number at the end of the digitspec to do the same with the output:

```
)PRINT USING "$$6#.3#+"; 09999
$ 9999.00+
```

When Z is used in a digitspec, you cannot use $$, ++, **, or --, because everthing to the left of the first digit is moveable. All non-space characters are shifted to the right, displacing any spaces. The spaces remain to the left of the field so that its width does not change.

The best way to learn the ins and outs of the fixspec is to experiment. Here is a program that should help:

```
5 REM NumericSpecTester
10 INPUT "Enter desired spec: "; SPEC$
15 ON ERR PRINT "You entered a bad spec, try again!" : GOTO
   10
20 F=1000:PRINT
30 X=1: GOSUB 100
40 X=12: GOSUB 100
50 X=123: GOSUB 100
60 X=1234: GOSUB 100
70 PRINT: PRINT " Spec was: "; SPEC$: PRINT
80 GOTO 10
100 PRINT USING SPEC$; X;: PRINT,X
110 PRINT USING SPEC$; -X;: PRINT,-X
120 PRINT USING SPEC$; X/F;: PRINT,X/F
130 PRINT USING SPEC$; -X/F;: PRINT,-X/F
200 RETURN
```

The program NumericSpecTester first asks you to enter a spec from the keyboard. It then displays two columns of numbers. In the left colum are values displayed according to the spec you entered, while the right column contains the same values output by a PRINT statement using no format statement. You can use this program to study the scispec and engrspec (described below.)

In addition to the digit spec characters, the fixspec uses the following characters. If Z is used, $$, ++, and -- may not be used.

Character	Function
+	Reserves character position for sign.
−	Reserves position for "−" if number is negative.
$	Reserves position for "$" sign.
**	Prints asterisks instead of leading spaces.
++	Reserves rightmost positions for signs and dollar sign (if any).
−−	Same as ++, except sign is printed only if number is negative.
$$	Reserves rightmost unused position for "$" and numeric sign (if any).

Table 3-4. Fixspec Characters.

The SCISPEC

The scientific-notation specification (scispec) formats numeric output in scientific notation. The scispec is simpler than the fixspec, having either one digit or none to the left of the decimal point. The number of digits to the right of the decimal point is defined by # characters, either stated explicitly or by using a repeat factor. The exponent position is defined with the letter E, and a repeat factor is legal. Either three or four character position *must* be allowed for the exponent. For example:

```
)PRINT USING "+#.4#4E"; 3.1415926
+3.1416E+00
)PRINT USING "+.4#4E"; 3.1415926
+.3142E+01
```

When the spec calls for one digit position to the left of the decimal point, the first significant digit of the value is placed there; when there is no digit position to the left of the decimal point, the most significant digit is placed to the right of the decimal point. In either case, the exponent is then calculated to make the displayed value correct.

Notice that with four character positions for the exponent, only two are available for the exponent's digits; with three character positions for the exponent, only one is available for the exponent's digit. If the calculated exponent will not fit the available space, the entire numeric field is filled with exclamation points.

The letter Z can be used instead of # in a scispec, but the effect is exactly the same. Note that if the sign is not explicitly specified in a scispec, the sign of the value will only be printed if there are enough available character positions and the value is negative.

Character	Function
E	Defines specification as Scispec.

Table 3-5. Scispec Character.

The ENGRSPEC

The engineering specification (engrspec) is closely related to the scispec. It forces the exponent's value to be a multiple of 3, and has a *maximum* of three digit positions to the left of the decimal point.

Either #'s or Z's can be used to indicate digit positions, and their choice is significant only to the left of the decimal point: # replaces leading zeros with spaces, and Z prints leading zeros. For example:

```
)PRINT USING "+3#.4#4E"; 1729
+ 1.7290E+03
)PRINT USING "+3Z.4Z3E"; 1729
+01.729E+3
```

SCALE

SCALE is used in conjunction with PRINT USING to shift the decimal point of a displayed value to the left or the right. SCALE uses two arithmetic expressions as arguments. The first argument defines the number of places to the right (or left, if negative) that the decimal point should be moved. The second argument is the actual numeric variable to be output.

SCALE takes ten raised to the power equal to SCALE's first argument and multiplies that by the value of its second argument. If the first argument value is 5 and the second is equal to 22435, the value output will be equal to 22435.0E5 output in the format specified by the PRINT(#) USING statement. For example:

```
)A&=12345678901234567
)PRINT USING "$$20&##";SCALE(-3,A&)
$12,345,678,901,235      (Note rounding of cents)
```

SCALE enables Apple Business BASIC to handle calculations in cents using long integers, and then to output the results with the decimal point placed to correctly indicate dollars and cents. You can take the same characters given in the example above and, with slight change, convert cents to dollars:

```
)A&=12345678901234567
)PRINT USING $$20&.##;SCALE(-2,A&)
$123,456,789,012,345.67      (Note cents!)
```

The first SCALE argument must be in the range -128 to 127, and the resulting exponent of the value must be between -99 and +99, or the

```
?ILLEGAL QUANTITY ERROR
```

message will be displayed.

4

Controlling Program Execution

4

Controlling Program Execution

Introduction

This chapter describes the statements and functions supplied by Business BASIC to help you control the path of execution of your programs. Even the most trivial program will use one or more of the items described here, and a very large program could require the use of most of them.

Assignment Statements

Assignment statements are used to assign the values of expressions to variables. A typical assignment statement might be

```
)Ripple=5*4 ^1/2
```

The variable to the left of the replacement sign (=) (RIPPLE, in the example), is assigned the value of the expression on the right side. Then you could type

```
)PRINT Ripple
10
)
```

The following are all legal assignment statements:

```
)IV%=9+PMT
)DSJ&=234324
)10 Body$(Arm,Leg)="Bone"
)10 ReassuringWords=Comforting-12
```

Variables and expressions of any type can be used in an assignment statement. But if the type of the variable on the left side of the replacement sign is different from the type of the expression on the right side, type conversion must occur. This happens automatically for integer and real variables and expressions, but must be handled explicitly for other types. (See the section on functions in the chapter Tools of Your Trade for an explanation of type conversion functions).

LET

The reserved word LET may optionally precede an assignment statement:

```
)LET Henry=FatherofJack
```

While LET is not required, it can make a program listing somewhat easier to read and understand.

Note: only one assignment may occur per statement. That is, the statement

```
)A=B=0
```

does not assign both A and B the value 0; instead, the logical expression B=0 is evaluated, and the result is assigned to the variable A. See the discussion about logical expressions in the chapter EXPRESSIONS AND STATEMENTS, if this is not clear.

SWAP

SWAP does exactly what you might expect. It swaps the value stored in one variable for the value stored in another. The names of the two variables whose values are to be swapped must follow the reserved word SWAP. For instance, the statements

```
)A=4 : B=8 : C=A/B
```

store the value .5 in the variable C. But

```
)A=4 : B=8 : SWAP A,B : C=A/B
```

stores the value 2 in the variable C.

You can use string, long integer, regular integer, and real variables with SWAP, but both variables must be of the same type. If the two variables are not of the same type, then a

?TYPE MISMATCH ERROR

results.

Remark Statements

Since programs are not written in natural langauges such as English, they are not always easy to understand. It seems to be a law that all non-trivial programs need to be maintained, and without remarks to aid the programmer, the task becomes nearly impossible.

Be generous and clear in your inclusion of remarks, and other programmers will be able to maintain your programs. Be stingy with them, and in a few months *you* won't know what your own programs are supposed to do or how they do it.

REM

REM allows you to insert remarks into your program. BASIC ignores everything in a statement list following the reserved word REM up to the next line number, and carries this text along with the rest of the program. For example:

)REM Munge BASIC : Bake Apple : INVERSE : LIST

will not cause BASIC to execute any of the words or statements following the reserved word REM.

The reserved word REM must be the first thing in a statement or the statement will not be treated as a remark. For example,

)HPOS REM arkably Tall Buildings

is not legal statement; but the following is:

```
)REM HPOS kyscraper
```

Like all other statements, REM statements must not exceed 250 characters in length. If you comment your programs heavily, use several REM statements in successive lines.

Branching

A program is said to *branch* when it does not execute the next higher numbered statement in sequence but, under program control, *jumps* to some other line instead. There are two kinds of branching, *conditional* and *unconditional*.

Since branching in BASIC is by reference to line numbers, things can get quite confusing if a programmer works in a sloppy fashion. The line numbers themselves do not carry any indication of the function carried out by the various parts of the program, and indiscriminate branching carried out with gay abandon can result in a tangled, spaghetti-like mass almost impossible to maintain or understand (or digest).

Unconditional Branching

A statement causing program execution to branch each time it is executed, under all conditions, is known as an unconditional branch.

GOTO

GOTO causes execution of the program to jump to the beginning of a specified statement list. You specify the statement list to which execution should jump, by following the reserved word GOTO with the line number of the statement list. For example,

```
)10 PRINT 10 : GOTO 40
)20 PRINT 20
)30 PRINT 30 : STOP
)40 PRINT 40
)50 PRINT 50 : GOTO 20
```

```
)RUN
10
40
50
20
30

BREAK IN 30
)
```

Most versions of BASIC begin at the lowest line number and search sequentially until the desired line is found. Business BASIC reduces the time spent in searching for line numbers in the following way: If the line number referenced by the GOTO statement is greater than the number of the GOTO, the search for that line begins with the current line, otherwise the search begins at the start of the program.

If the line number given in the GOTO statement does not exist, or if there is no line number given, then the error message,

```
?UNDEF'D STATEMENT ERROR IN 5293
```

is displayed. In this case, 5293 is the line number of the erroneous GOTO statement. Immediate execution of a similarly illegal GOTO statement such as

```
)GOTO 45
```

when there is no program in memory will generate an

```
?UNDEF'D STATEMENT ERROR
```

Conditional Branching

One of the most powerful capabilities of a computer program is the ability to alter how the program executes depending on the data it is working with; in effect, allowing the computer to make choices that depend on circumstances outside the computer. This is called conditional branching.

IF. . . GOTO and IF. . . THEN

IF statements allow the order of execution of statements to depend on the truth value of a logical expression. The IF statement must include both a logical expression to be evaluated, and instructions for BASIC to follow if the expression is true. If the expression is false, execution passes to the next higher numbered line of the program, and the instructions given in the IF statement are ignored. The logical expression to be evaluated must follow the reserved word IF, and the instructions must follow the reserved words GOTO or THEN.

The instruction in an IF...GOTO statement must be a line number to which execution should branch. For example,

```
)IF A=4 GOTO 473
)IF KP+BH GOTO 3785
)100 IF G& MOD F& >2 GOTO 121
```

In an IF...THEN statement, the instructions can be any line number to which execution should branch, or a statement list for BASIC to execute. For example,

```
)IF 0 THEN PRINT 1
)50 IF 2+2 THEN 2500
)IF S/4>=17*NOT 2 THEN GOSUB 3000 : INVERSE : PRINT
"Hi"
)IF Language=German THEN PRINT "Gesundheit" :
Sneezes = Sneezes + 1
)10 IF Language=English THEN PRINT "Bless you!" :
Sneezes = Sneezes + 1
```

These are all equivalent statements:

```
)IF G=5 THEN 200
)IF G=5 GOTO 200
)IF G=5 THEN GOTO 200
```

ELSE

Usually when you test a condition with an IF..THEN statement, you have some alternate operation to perform, and it helps if the program syntax reflects this. The optional ELSE clause helps improve readability of your programs.

ELSE allows you to specify instructions for BASIC to execute if the truth value of the logical expression is false. In other words, when the expression is false, instead of having execution pass to the next higher numbered line, you can have BASIC execute some instructions. The instructions following the reserved word ELSE can be a line number to which execution should branch, or a statement list to execute. If the logical expression is true, the ELSE clause and any statements following on that line are ignored. For example,

```
)IF X=1 THEN Y=2 : ELSE Y=3
)IF 3<PL5 THEN PL5=-PL5 : ELSE NORMAL : GOTO 376
)718 TEXT : IF NOT Y THEN 3200 : ELSE WINDOW 1,1 TO 4,4
   : GOTO 457
```

and,

```
)10 IF 0<5 THEN PRINT 101 : ELSE PRINT 100
)20 IF 0>5 THEN PRINT 201 : ELSE PRINT 200
)RUN
101
200
```

An ELSE clause not immediately following an IF...THEN statement acts exactly like a REM statement. For example,

```
)ELSE Whatever you want to remark about
here!
```

Nested Conditional Statements

The statement list following THEN and ELSE may contain as many additional IF...THEN or IF...THEN...ELSE statements as the 250 character limit will allow. Conditional statements contained inside other conditional statements in this manner, are said to be *nested*. In these circumstances, BASIC matches each ELSE with the most recently encountered and unmatched THEN. For an example from baseball:

```
)IF bottom_of_the_ninth THEN IF bases_loaded THEN bunt :
ELSE swing_at_it
```

In this case, the ELSE clause goes with the second THEN, which is the one most recently unmatched.

Assume it is time for your yearly dental checkup. If you have a toothache and the dentist is on vacation, then there is nothing to do but suffer until you can get an appointment. If you don't have a toothache, you can smile.

```
)IF toothache THEN IF dentist_is_on_vacation THEN suffer
: ELSE call_for_appointment : ELSE smile!
```

Remarks about Conditional Statements

The following IF statements have legal logical expressions:

```
)IF 1<2 THEN PRINT "Yup"
)IF 0 THEN PRINT "Nope"
)IF A<B THEN PRINT "Yup" : ELSE PRINT "Nope"
)IF "A"<"B" THEN PRINT "Yup"
)IF 1+2*Z THEN PRINT "Maybe so" : ELSE PRINT "Maybe
not"
```

The following IF statements do *not* have legal logical expressions:

```
)IF "Fred" THEN PRINT "Fred"
)IF KAREN$ THEN PRINT "Fred's friend"
)IF "Fred"+KAREN$ THEN PRINT "Fred 'n Karen"
```

Here's why: remember that a logical expression, can be either a comparison of arithmetic expressions, or a comparison of string expressions. It is important to remember that logical expressions reduce to a truth value of 1 or 0. While it is legal to substitute a single arithmetic expression for a logical expression, it is not legal to substitute a single string expression for a logical expression. This is because BASIC has rules for treating the value of an arithmetic expression as true or false. However it does not have similar rules for string expressions. If you want to test whether a string is null or not, use the LEN function (described in the section STRINGS) like this:

```
)IF LEN(KAREN$) THEN (or GOTO) PRINT "Karen exists!"
: ELSE PRINT "No Karen"
```

An IF without a matching THEN (or GOTO), or a THEN (or GOTO) without a matching IF generates a

```
?SYNTAX ERROR
```

message.

ELSE must be preceded by a colon when it follows an IF...THEN statement. For example:

```
100 IF X=12 THEN 1000 : ELSE 2000
200 IF Minnow$=Fish$ THEN Answer$=True
210 ELSE Answer$=False
```

Looping

Many problems suitable for solution by your Apple require that one or more operations be carried out repetitively. Looping describes this process.

Loops can be divided into two general types: Those operating a determined number of times, and those operating either as long as a specified condition is true or until a specified condition is met. For example:

```
10 INPUT "Enter your Social Security Number: ";SSN$
20 IF SSN$="562-43-0666" THEN STOP : ELSE GOTO 10
```

The first type is exemplified by the FOR..NEXT structure described below, and the second type is usually constructed in BASIC with IF..THEN (or IF..GOTO) or GOTO statements like the program fragment above, which will continue asking for social security numbers until 562-43-0666 is entered, when it will stop.

FOR and NEXT

FOR and NEXT allow a group of statements to be executed a specified number of times. For example, if you wanted to display the numbers 1 to 5, you might write a program like this:

```
)10 Number=1
)20 PRINT Number
)30 Number=Number+1 : IF Number<6 THEN 20
```

The FOR and NEXT statements offer you an alternative method for constructing such loops as the one given above. The FOR statement defines the beginning of the statement list making up the body of the loop and sets the number of times it is to be executed, and the NEXT statement defines its end. For example, to display the numbers 1 to 5, you could use this program:

```
)10 FOR Number=1 TO 5
)20 PRINT Number
)30 NEXT Number
)RUN
1
2
3
4
5
)
```

In line 10, the *control variable* Number is assigned a beginning value of 1, and an ending value of 5. Line 30 in this program does exactly what the statements in line 30 did in the program before it. The NEXT

statement increments the value of Number by 1, and then checks to see if the value of Number is greater than the ending value that was specified in the FOR statement.

If the value of Number is less than the ending value, execution loops back to the statement immediately following the FOR statement (line 20). If the value of the control variable is greater than the ending value, execution continues with the next statement immediately following the NEXT statement.

The control variable can be either a real or an integer variable, but not a long integer or string variable. The beginning and ending values assigned to the control variable in the FOR statement can be the result of arithmetic expressions. For instance,

```
)FOR Repeat%=T+44-F6/D*NOT R TO 54*5/FJ : NEXT
Repeat%
```

is perfectly legal, if somewhat obscure.

FOR..NEXT loops may contain other FOR..NEXT loops; loops contained within other loops are said to be nested. For instance:

```
)10 FOR Row=1 TO 3
)20 FOR Column=2 TO 3
)30 PRINT Row, Column
)40 NEXT Column
)50 NEXT Row
```

NEXT statements may contain as many control variable names as you would like. For example:

```
)60 FOR Loop1=1 TO 4 : FOR Loop2=4 TO 55
)70 FOR Loop3=-4 TO 55
)80 NEXT Loop3, Loop2, Loop1
```

The first control variable given in the NEXT statement must be the same as the one named in the most recently executed FOR statement; the second control variable given must match the second most recently executed FOR statement, and so on. Incorrectly

matched FOR and NEXT statements cause the message

```
?NEXT WITHOUT FOR ERROR
```

to be displayed. The following example contains incorrectly nested loops:

```
)90 FOR A=1 TO T%
)100 FOR B=1 TO 43
)110 NEXT A, B
```

STEP

FOR statements may optionally include a STEP clause allowing you to specify the amount to increment the control variable with each iteration of the loop. For instance:

```
)FOR BYTE=2 TO 10 STEP 3 : PRINT BYTE : NEXT BYTE
2
5
8
)
```

If the STEP clause of the FOR statement is absent, the control variable is incremented by 1, by default, when the following NEXT statement is executed. You can use any arithmetic expression to specify the value to increment the control variable.

If a negative increment value is specified in a STEP clause, and the value of the control variable is less than the ending value after it has been incremented by a NEXT statement, execution passes to the statement following the NEXT; in other words the loop is terminated when the NEXT statement is reached. For example,

```
)FOR Loop=1 TO 10 STEP -2 : PRINT Loop : NEXT Loop
1
)
```

If a negative value is specified in a STEP clause, the loop counts backwards. For instance,

```
)FOR Loop=10 TO 1 STEP -2 : PRINT Loop : NEXT Loop
10
8
6
4
2
)
```

If the increment value is 0, the control variable is incremented by 0 each time a NEXT statement is executed. Which means that the value of the control variable will never be greater than the ending value, and the statements between FOR and NEXT will be repeated indefinitely (unless the ending value is less than the control variable at the start). This is known as an infinite loop. The only cure for such a situation is to type CONTROL-C, or press the RESET button.

The control variable is incremented and compared to the ending value only when the NEXT statement at the end of the FOR..NEXT loop is executed. This means that the statements between the FOR and NEXT are *always* executed at least once.

A NEXT statement without a specified control variable defaults to the control variable given in the most recently executed FOR statement still in effect. A NEXT statement without a specified control variable executes faster than one with a specified control variable. For example,

```
)FOR G6=4 TO REV3/21 : NEXT
```

If there is no FOR..NEXT loop in effect, executing a NEXT statement generates a

```
?NEXT WITHOUT FOR ERROR
```

message. The same error message is displayed if the control variable specified by a NEXT statement is different than the control variable given in the most recently executed FOR statement.

Nesting more than 9 FOR..NEXT loops inside one another elicits the

?STACK OVERFLOW ERROR

message.

If a deferred-execution FOR statement is still in effect, an immediate execution NEXT statement can cause a jump to the deferred execution program, where appropriate.

Subroutines

A subroutine is a group of statements that perform some specialized or frequently repeated task.

GOSUB

GOSUB causes BASIC to branch to a subroutine. You must follow the reserved word GOSUB with the line number of the first statement in the subroutine. When BASIC executes a GOSUB statement, a pointer to the statement immediately following the GOSUB statement is placed at the top of a list of pointers called a *program stack*, then execution is transferred to the line number given in the GOSUB statement.

If the line number given in a GOSB statement does not correspond to an existing line in the program, then the message

?UNDEF'D STATEMENT ERROR IN 746

is given, where 746 indicates the line containing the erroneous GOSUB statement.

Return

RETURN has no parameters or options. When executing a RETURN statement, BASIC removes one pointer from the top of the *program stack* and branches to the statement indicated by the pointer. This is normally the statement immediately following the most recently executed GOSUB statement. For example, this program

```
)10 GOSUB 200
)20 PRINT "Back"
)30 GOTO 999
)200 REM The subroutine goes here
)210 RETURN
)999 END
```

runs as follows. When line 10 is executed, a pointer to the next statement (line 20) is placed on the top of the program stack, and execution branches to the subroutine at line 200. After line 200, execution transfers to line 210, where the RETURN statement causes BASIC to remove the top pointer from the stack and branch to the line indicated (line 20).

Another example of GOSUB and RETURN:

```
)10 PRINT "Now branching to subroutine 100"
)20 GOSUB 100
)30 PRINT "Hello again"
)40 END
)100 PRINT "Subroutine 100 speaking"
)110 RETURN : REM Line 30 will now be executed.
```

A program can have nested subroutines (subroutines calling other subroutines) up to 23 deep. If GOSUBs are nested more than 23 deep, the

```
?STACK OVERFLOW ERROR
```

message is given. For example, here is a program with subroutines nested 3 deep:

```
)10 GOSUB 100
)20 END
)50 GOSUB 200
)60 RETURN : REM Branch to 120
)100 GOSUB 50
)120 RETURN : REM Branch to 20
)200 RETURN : REM Branch to 60
```

You should note that the BASIC statement RETURN is not the same thing as pressing the key marked RETURN. The RETURN associated with subroutine is a normal BASIC statement, and is spelled out.

If BASIC attempts to execute one more RETURN statement than it has encountered GOSUB statements, the

?RETURN WITHOUT GOSUB ERROR

message is displayed.

POP

POP allows you to jump out of one level of subroutine nesting.

When a POP statement is executed, BASIC removes (POP's) the top pointer from the program stack and discards it, without causing execution to branch anywhere. When the next RETURN statement is encountered after a POP statement is executed, instead of branching to the first statement beyond the most recently executed GOSUB, BASIC branches to the first statement beyond the *second* most recently executed GOSUB.

If a POP statement is executed before a GOSUB has been encountered, or if more POPs and RETURNs are encountered than GOSUBs, the message

?RETURN WITHOUT GOSUB ERROR

is given because more pointers have been removed from the stack than were placed on it.

Example of POP:

```
)10 GOSUB 100
)20 PRINT "End of program"
)30 END
)100 REM This subroutine has no return statement
)110 PRINT "Subroutine 100 speaking"
```

```
)120 PRINT "About to branch to subroutine 200"
)130 GOSUB 200
)140 REM This line is never executed
)200 PRINT "Subroutine 200 speaking"
)210 PRINT "Popping to avoid returning to line 140"
)220 POP : REM Causes pointer to line 140 to be removed
from the stack
)230 RETURN : REM Execution now resumes at line 20
)RUN
Subroutine 100 speaking
About to branch to subroutine 200
Subroutine 200 speaking
Popping to avoid returning to line 140
End of program
)
```

Computed Branching

You will find that many programs require that a different set of operations should be for each different possible value in a range. Computed branching allows you to easily tailor your program to respond to a number of possible conditions.

ON...GOTO

ON...GOTO is used to specify different program branch points, based on the value of an arithmetic expression. The arithmetic expression must follow the reserved word ON, and the line numbers to which execution branches must follow the reserved word GOTO. For example,

```
)1000 ON X GOTO 100, 10, 300, 40
```

If X=1, execution branches to the first line in the list of numbers (line 100). If the value of X is 2, then execution branches to the second line in the list (line 10). If X=3, execution branches to line 300 (the third line in the list), and so on.

The value of the arithmetic expression must be within the range 0 to 255, or an

?ILLEGAL QUANTITY ERROR

results. If the value of the arithmetic expression is 0, or greater than the number of line numbers given in the ON...GOTO statement, the list of line numbers is ignored, and execution continues with the next statement in the program.

ON...GOSUB

ON...GOSUB is identical to the ON...GOTO statement, except that the line numbers following the reserved word GOSUB must be line numbers of subroutine entry points. For example:

)1000 ON X GOSUB 1000, 2000, 3000, 4000

When a RETURN statement within the subroutine is executed, execution branches to the statement immediately following the ON...GOSUB statement.

The value of the arithmetic expression must be within the range 0 to 255 or an

?ILLEGAL QUANTITY ERROR

message is displayed.

The list of line numbers in the ON...GOSUB statement will be skipped entirely if the value of the arithmetic expression is 0, or greater than the number of line numbers given; execution will continue with the next statement in the program.

ON KBD and OFF KBD

ON KBD is used to cause BASIC to execute a specific statement list immediately when any key is pressed. The statement list to be executed must follow the reserved word KBD.

After an ON KBD statement has been executed, BASIC continues executing the program normally—but as soon as any key is pressed, execution branches back to the most recently executed ON KBD

statement. Then the statement list pointed to by the ON KBD statement is executed.

The branch to the ON KBD statement list is treated as a GOSUB to a subroutine, so the program segment that KBD causes to be executed must end with a RETURN statement. To enable ON KBD to handle more than one keystroke, the last statement in the list should be another ON KBD statement. For example,

```
10 ON KBD GOTO 100 : REM BASIC branches here when any
key is pressed
20 PRINT "."; : REM Print periods while not handling key-
strokes
30 GOTO 20
100 PRINT KBD : REM Display the ASCII value of the key last
pressed
110 ON KBD GOTO 100 : REM Re-enable ON KBD. Must be
before return
120 RETURN : REM Program jumps back to the statement
following the one during which a key was pressed
```

This displays zillions of periods (more or less), and whenever a key is pressed, the instructions in the ON KBD statement are executed.

BASIC forgets the last ON KBD statement as soon as a key is pressed, even before BASIC executes the statement list in the ON KBD statement. This is why the example program had to execute another ON KBD statement in line 110. BASIC also forgets the last ON KBD statement executed, if the program returns to immediate execution for any reason.

Execution of an OFF KBD statement causes BASIC to forget the last ON KBD statement that was executed.

An ON KBD statement must be executed just prior to the RETURN statement, or a

```
?STACK OVERFLOW ERROR
```

message may result.

When ON KBD is in effect, the program can not be halted with CONTROL-C, since this is treated just like any other keystroke. However, the ON KBD statement could cause a branch to a STOP or END statement if a CONTROL-C is typed. A RETURN statement placed after the STOP would allow the CONT statement to be used.

The Reserved Variable KBD

Business BASIC allows you to read, but not write to, several of its system variables. These are called *reserved variables*. KBD contains the ASCII value of the last key struck (see the appendix, ASCII Character Codes). When you use the reserved variable KBD in an ON...GOTO or ON...GOSUB statement, you must enclose KBD in parentheses, or BASIC will become confused as to the meaning of your statement:

```
)ON (KBD)-64 GOTO 100,200,300
```

Handling Errors

If your program is going to be used by someone else, you might want to have expected errors handled by the program rather than have Business BASIC's normal error responses bring everything to a crashing halt. The tools to let you do this are described below.

ON ERR and OFF ERR

ON ERR is used to force BASIC to let your program handle any errors that might occur.

When an ON ERR statement is not in effect, and an error occurs in deferred execution, BASIC displays a terse error message on the screen and halts execution; this can cause confusion if the person running the program does not understand programming.

ON ERR is typically used to give a more enlightening or appropriate error message or to provide the innocent user of your program with a chance to avoid causing another error.

A statement or statement list must follow the reserved variable ERR. Execution branches to the subroutine referenced by the statement list whenever an error is encountered, and BASIC does not display an error message on the screen or halt execution.

The ON ERR statement should not be used as a tool for finding errors in programs. (Use the TRACE statement for this instead). ON ERR is intended primarily to allow you to write nicer, more forgiving programs for users.

For the ON ERR statement to be most effective, it is important that it be placed near the beginning of the program, since BASIC must execute it before it can use it. An ON ERR statement must be executed before the program encounters an error.

If a program contains more than one ON ERR statement, only the most recently executed one will be used.

OFF ERR cancels the most recently-executed ON ERR statement. There are no parameters or options associated with this statement. After an OFF ERR statement has been executed, BASIC resumes displaying error messages and halting execution just as it did before the ON ERR statement was executed.

The statements that ON ERR causes to be executed must themselves be free of errors, or an endless loop may result. The endless loop will be unstoppable by CONTROL-C, because CONTROL-C is itself considered an error. For a complete list of BASIC errors, see the appendix ERRORS.

The following program illustrates one simple way to use the ON ERR statement. In this example, the computer is expecting the e 2 user to enter a number, and the error handling statements are executed if a letter or word is typed instead:

```
10 REM EXAMPLE OF ERROR HANDLING
20 ON ERR GOSUB 1000
```

```
30 INPUT "Please type a single number between 1 and 100
";X
40 PRINT "The number you typed was ";X
50 END
1000 REM ERROR HANDLING SUBROUTINE
1010 PRINT : PRINT "I'm very sorry, but only a number will
do. Please try again."
1020 RETURN
```

RESUME

If your error-handling routine ends with a RESUME statement, execution will begin again at the start of the line where the original error occurred.

Business BASIC ignores RESUME statements that it encounters until an error occurs. If you try to use RESUME in immediate execution, an

```
?ILLEGAL DIRECT ERROR
```

results.

ON ERR subroutines using RESUME must be error-free. Errors in your error routines may "hang" the system. If this happens, you will have to reboot BASIC, and anything in memory will be lost.

The Reserved Variable ERR

When BASIC encounters an error, it assigns the reserved variable ERR a code number corresponding to the type of the detected error. You can then refer to the reserved variable ERR to determine what kind of error occurred. For a list of these codes and the corresponding error messages, see the appendix ERRORS.

The Reserved Variable ERRLIN

Your error-handling routines called by an ON ERR statement can check the reserved variable ERRLIN to find out which line contained the error.

Error-Recovery Strategies

The statements that ON ERR causes to be executed can be any legal BASIC statements. Since they must be able to handle any BASIC error, certain strategies suggest themselves.

One thing you can do with the ERR reserved variable is to use it in an ON...GOSUB statement to handle the individual errors that can occur. Each individual subroutine could then do whatever you like about its own particular likely error conditions. When the subroutine returns, the last statement in the ERR statement list would be a RESUME statement.

5

File I/O

5

5
File I/O

Introduction

This chapter explains how to use devices and files with BASIC. You should read the chapter in your *Apple III Owner's Guide* dealing with files to understand files in general and the terminology describing Apple III files.

Apple Business BASIC treats every peripheral device connected to your Apple as a file, including the keyboard, screen, printer, and disk drives. This means that there need be only one method of doing input or output for all the different peripherals connected to your Apple. Business BASIC allows you to have a maximum of 10 files open at one time, including 1 EXEC file (for a total of 11 files).

Filenames

To use any given file, you must refer to it by its *local filename* or *pathname*.

Local filenames can be any sequence of 15 or fewer letters, digits, or periods, beginning with an alphabetic character. They may not begin with a slash character or contain spaces.

The local filenames of character devices always begin with a period (.). For instance, .PRINTER, and .CONSOLE are the filenames assigned by BASIC to refer to the printer device driver, and the console (the Apple screen and keyboard) driver.

Pathnames may be up to 128 characters in length. The *Apple III Owner's Guide* describes pathnames in greater detail.

You can refer to a specific disk drive by using the device reference names .D1, .D2, .D3, or .D4 as a partial pathname. .D1 refers to the volume in the built-in disk drive, and .D2, .D3, .D4 refer to volumes in any additional external disk drives.

Diskettes get their volume names and root directories when they are formatted. To format diskettes, you must use the FORMAT DISKETTE option on your Utilities diskette as described in your *Apple III Owner's* Guide.

To create BASIC *program* files, you must use the SAVE statement. To create BASIC text files, data files, or subdirectories, you must use the OPEN or CREATE statements.

The Modifiable Reserved Variable PREFIX$

Since BASIC requires a full pathname any time you want to use a file, how can you use partial pathnames?

Pathnames starting with any character other than a period or a backslash (partial pathnames) are interpreted by BASIC as the contents of the modifiable reserved variable PREFIX$ concatenated with the partial pathname as entered.

To set the prefix in BASIC, so that you can use partial pathnames to refer to files, assign a string constant to the reserved variable PREFIX$. For example, if you wanted the prefix to be /Personnel/Communication/Internal, use the statement

```
)PREFIX$= /Personnel/Communication/Internal
```

After you have set the prefix, you can refer to files by using either partial pathnames or the local filename. If you wish to access another disk and override the prefix, use a full pathname.

When you boot Apple Business BASIC, PREFIX$ is automatically set to the volume name of the diskette that was used to boot the system. You can find the name of the current prefix with a PRINT statement:

```
)PRINT PREFIX$
```

A pathname can refer to a specific device by preceeding the name with a period, for example; .PRINTER or .D1

A pathname begining with any alphabetic character is a partial pathname that refers to a file defined by the contents of PREFIX$ plus the partial pathname supplied.

A pathname begining with a "/" is assumed to be the complete pathname of a file.

Creating Files

CREATE

CREATE is used to make root directories, subdirectories, text files, and data files. You must specify exactly what type of file you want to create, and its name, by following the reserved word CREATE with the new pathname and either TEXT, DATA, or CATALOG, separated by a comma. TEXT specifies that a text file be created; DATA specifies that a data file be created; and CATALOG specifies a root directory or subdirectory. For example, to create a text file called Applepie on a diskette whose volume name is Pies, use

```
)CREATE "/Pies/Applepie", TEXT
```

Recall that you can use the prefix stored in the reserved variable PREFIX$ by leaving off the initial backslash of the partial pathname. For instance,

```
)PREFIX$= "/Pies"
)CREATE Applepie, TEXT
```

causes the full pathname to be /Pies/Applepie. Or,

```
)CREATE Fruitpies, CATALOG
```

creates a subdirectory called Fruitpies, (using the prefix specified in PREFIX$ as the first part of the pathname) that may then contain files.

A files' *record* size may be specified by appending an arithmetic expression to the CREATE argument list. The record size is required only for random access files, and must be in the range from 3 to 32767. If no record size argument is given, the file record size defaults to 512 bytes. For example,

```
)CREATE Attache, TEXT, 4212
```

creates a file with the local filename Attache, having records of 4212 bytes each. When creating subdirectories, the arithmetic expression specifies the size of the directory. The directory size defaults to 512 bytes if unspecified (enough to hold twelve files).

An attempt to create an already-existing file generates a

```
?DUPLICATE FILE ERROR
```

message.

As a convenience in immediate execution, the pathname may be entered directly in CREATE and OPEN statements rather than either as a string variable or as a literal enclosed by quote marks. For example, in immediate execution, the statements

```
)CREATE /Foo/Fighter, DATA
```

and

```
)CREATE "/Foo/Fighter", DATA
```

are equivalent. In deferred mode, the quotation marks are required. Not using them results in a

```
?TYPE MISMATCH ERROR
```

Manipulating Files

Business BASIC provides you with several statements to directly manipulate files. Using these statements you can see what files are on a volume, remove unwanted files, rename, and lock and unlock files.

CATALOG

CATALOG displays a listing of a root directory or subdirectory specified by the pathname following the reserved word CATALOG. If the specified pathname is a diskette volume name, then the names of all files in the diskette root directory are displayed on the screen and the names of any subdirectories of the root directory are also displayed. For example, to see a catalog of a diskette named Apple1, enter

```
)CATALOG /Apple1
```

If the pathname specified is a diskette subdirectory, then the names of all files in that subdirectory are displayed. For example, if Applekind is a subdirectory,

```
)CATALOG /Apple1/Applekind
```

will list the names of all the files in the Applekind subdirectory.

If a partial pathname is specified as the CATALOG argument, the prefix stored in the reserved variable PREFIX$ is used.

The reserved word CATALOG may optionally be abbreviated as CAT.

The file types displayed by CATALOG are shown below in Table 5-1.

BAD—An area on the diskette that is damaged in some way.

BASIC—BASIC program file (created with SAVE).

BINARY—An Apple III assembly-language file.

CAT—root directory or subdirectory.

DATA—BASIC data file.

FONT—An Apple III file containing binary information about a character set.

FOTO—A file containing data representing a picture on a graphics display screen.

PASCOD—Pascal code file.

PASDTA—Pascal data file.

PASTXT—Pascal text file.

RESERV—Reserved for future file types.

SYSTEM—system program file.

TEXT—BASIC text file.

UNKNWN—BASIC data or text file; the file has been opened, but not written to with either a PRINT or WRITE statement.

Table 5-1. File Types Displayed by CATALOG.

File types are discussed in detail later in this chapter.

DELETE

The DELETE statement is used to remove the subdirectory or file specified as its argument. A subdirectory may be removed only if all files in that subdirectory have been deleted. If the last file in a root directory is deleted, the empty root directory will still remain. For example, to delete a file named Banana in a root directory named Tree, you would enter

```
)DELETE /Tree/Banana
```

A number of errors can occur with improper arguments appended to a DELETE statement. They are summarized in the table below.

Error Message	Cause
?VOLUME NOT FOUND ERROR	Volume name given does not exist.
?PATH NOT FOUND ERROR	Subdirectory does not exist.
?FILE NOT FOUND ERROR	Non-existent local file name.
?FILE LOCKED ERROR	Subdirectory contains files, or specified file is locked.
?WRITE PROTECTED ERROR	Diskette is write-protected.
?FILES BUSY ERROR	One or more files are open.

Table 5-2. Errors Possible with DELETE.

RENAME

RENAME is used to change the names of volumes, subdirectories, and local files. RENAME's argument list is composed of the old pathname, followed by a comma, followed by the new pathname. For example,

```
)RENAME /Floppy2/Animals/Dogs, /Floppy2/Animals/Pigs
```

changes the name of the file Dogs, in the subdirectory Animals of the diskette whose volume name is Floppy2, to Pigs. If the second pathname specified indicates a file that already exists, the item is not renamed, and the

?DUPLICATE PATHNAME ERROR

message is displayed.

Remember that using PREFIX$ reduces the length of the pathname you must supply.

You cannot use the RENAME statement to create a file or subdirectory, only to rename an existing one. Use the CREATE statement to make new files and root directories.

A local filename or subdirectory may not be changed to another volume name or subdirectory. For example, either

```
)RENAME /Thisdisk/Tweedledee/File1,
/Thisdisk/Tweedledum/File2
```

or

```
)RENAME /Thisdisk/Tweedledee/File1,
/Thatdisk/Tweedledum/File2
```

will cause a

?BAD PATH ERROR

LOCK and UNLOCK

LOCK prohibits writing to, saving, or deleting the file named as its argument. Locked files are shown with an asterisk to the left of their file type when cataloged. Volume names may not be locked, but subdirectories may be.

A locked file may not be deleted, renamed, changed or saved until it has been unlocked with the UNLOCK statement. The reserved word UNLOCK must be followed by the file's name.

To protect all the files on a diskette, a write-protect tab may be placed over the write-protect cutout on the upper right edge of the diskette.

File Types

Your most useful programs are likely to be those that read from or write to files. The two types of files that your programs will be using are text and data files, described below.

Text files contain only text in the form of characters and strings of characters. Numeric information stored in text files is automatically converted into string form as if done by the STR$ function.

A string representing a numeric value to be assigned to a numeric variable when read from a text file is automatically converted to the proper form for that variable.

Reading from or writing to a file is referred to as *accessing* the file. A single access operation usually affects only a portion of the data or text within the file being accessed.

The type of a file is determined at the time that the file is created, either by assignment with a CREATE statement or by the access method used first after creating the file with an OPEN# statement. Once the type of a file is determined, it can not be changed except by deleting the file and then recreating it—and then it's really a different file even if it has the same name. Statements directly affecting files are described in the remainder of this chapter.

Opening and Closing Files

Before a file can be accessed it must be opened, and it should be closed after the user is finished with it. An open file is like an open box—you can put things into the box, and you can take things out of the box. Similarly, a closed file is like to a box with a lid on it—you can't access the box's contents. Apple Business BASIC allows you to have up to 10 files open at the same time.

OPEN#

OPEN# is used to open files for access, and must precede any file I/O statements accessing a given file. The arguments following OPEN# are a file reference number and the file's pathname. The file reference number is used in all subsequent I/O statements to refer to the file while it is open. The file reference number can be any arithmetic expression having a value between 1 and 10, inclusive. OPEN# examples are given below:

```
)OPEN #6, Door
)OPEN #4, Window
)OPEN #2, .Console
)OPEN #1, .Printer
```

If the OPEN# file reference number is followed by the reserved words AS INPUT, the file is opened as a read-only file and may not be written to. For example,

```
)OPEN #2 AS INPUT, Arms
```

If the OPEN# file reference number is followed by the reserved words AS OUTPUT, the file is opened as a write-only file and may not be read from. For example,

```
)OPEN #10 AS OUTPUT, Session
```

The AS EXTENSION option is a variant of AS OUTPUT, and is used in sequential access (explained later) to allow PRINT# or WRITE# statements to append new information to the end of an existing file without disturbing any data that was put in the file earlier. For instance,

```
)OPEN #1 AS EXTENSION, Ladder
```

Note that no comma follows the file reference number if an AS option is specified.

If no AS options are specified the file can be both read from and written to if the file type supports such access. For example, you can only write to a printer device driver.

If an OPEN# statement contains a file reference number equal to one presently in use, the first file using that file reference number is automatically closed.

As a convenience in immediate execution, the pathname may be entered directly in OPEN statements rather than either as a string variable or as a literal enclosed by quote marks. For example, in immediate execution, the statements

```
)OPEN /Foo/Fighter, Data
```

and

```
)OPEN "/Foo/Fighter", Data
```

are equivalent. In deferred mode, the quotation marks are required. Not using them results in a

```
?TYPE MISMATCH
```

CLOSE and CLOSE#

Before ending program execution, all open files should be closed with either a CLOSE# or CLOSE statement. Any files closed during program execution must be reopened before they can be accessed again. Each time a file is opened, even if it was used earlier in the same program, BASIC treats it as a new file.

CLOSE# closes the file whose file number is equal to the arithmetic expression that follows CLOSE#.

CLOSE closes all files that are open when the statement is executed. All open files are also closed by a LOAD, CLEAR, NEW, or RUN statement. The CHAIN statement does not close any files.

Accessing Files

Text files and device driver files are accessed with the same set of BASIC I/O statements; INPUT#, PRINT#, and PRINT# USING. Data files are accessed with a different set of I/O statements, described later.

INPUT#

INPUT# reads a line of text for each variable in its list of variables. The file to be read from is defined by a file number following the reserved word INPUT#. If the file number is followed by a comma, the arithmetic expression following the comma specifies a record number to begin file access at. Examples of legal INPUT# statements are given below.

```
)INPUT# 2; Payment%, Grease$
)INPUT# 8, 34; DG(0), DG(2), DG(4)
```

INPUT# automatically performs any necessary string to numeric type conversions (similar to the VAL function), in order to store newly-read information into any numeric variables in the variable list.

If INPUT# is reading to a numeric variable from a random-access file and the record being read from is either empty or contains only non-numeric information an

```
?TYPE MISMATCH
```

error is generated.

An INPUT# statement with no variables in the variable list is allowed, but does nothing.

You may open a directory as you would a text file by specifying the pathname. INPUT# statements may then access the directory to return, one line at a time, the same information returned by the CATALOG statement.

If an INPUT# statement calls for a numeric variable and the file record accessed does not contain numeric data, a

?TYPE MISMATCH

will occur. If the accessed record contains numeric characters followed by non-numeric characters in that record, the numbers are accepted, the other characters are discarded and an

?TYPE MISMATCH

warning message is displayed.

If the file record being read has more fields than the number of variable entries being requested, an

?EXTRA IGNORED

warning is issued.

OUTPUT#

Normally, BASIC sends all its output, such as error messages, prompts, and so on to the console. Sometimes you might want to *redirect* output to another file to get a record of a program's output, for example.

OUTPUT# redirects screen output to a specified file. All PRINT, LIST, TRACE, and CATALOG statement output is sent to the specified file, but error messages and keyboard input is still echoed to the screen. The file used for output is specified by its file number (set by an OPEN# statement) following the reserved word OUTPUT#. For example,

)OUTPUT #5

Note that INPUT prompt strings will be sent to the file specified in the OUTPUT# statement, and not to the screen.

If there is no file open with the same file reference number, the

?FILE NOT OPEN ERROR

message is displayed. If the file specified is not a file that can accept characters, the message

?TYPE MISMATCH ERROR

is displayed. To resume normal screen output, type

)OUTPUT# 0

and characters will again be displayed on the screen.

The TRACE statement should not be used to debug programs using the OUTPUT# statement, unless you want the TRACE-generated line numbers sent to the file.

PRINT#

PRINT# writes information to files like PRINT writes information to the screen. After the file number (and record number, if included) a list of expressions separated by commas must follow. For example,

```
)PRINT# 1; W$(0,0,0), LEFT$(W$(0,0,1))
)PRINT# 10, 4755; A&+24, T&/43, R%
```

One line of text is written for each expression in the list. PRINT# automatically performs any necessary numeric to string type conversions (similar to the STR$ function) in order to transfer the information from the expressions to the file.

A PRINT# statement in which a specific record number is given, starts writing information to the file at the beginning of the specified record.

The SPC specification can be used with PRINT# statements in the same way it is used with PRINT statements.

Just like PRINT, PRINT# allows commas in place of semicolons, but if you use commas strange line breaks appear in the output, because files have no tab positions. The TAB specification can also be used as it is in PRINT—but, it too may cause strange results. As in PRINT, some expressions can be run together without commas or semicolons. This is not recommended.

PRINT# USING

The PRINT# USING statement is the same as a PRINT statement with a USING clause used to control the format of information sent to the file. Both PRINT and USING are described in detail in the chapter on CONSOLE I/O.

Data Files

READ# and WRITE# are used for diskette data file access. Data file access is much faster than text file access because no data conversion is required. The advantage of a text file is that it allows you to use PRINT# and INPUT#, which are usually the most convenient statements for handling text I/O.

READ#

READ# gets information from a data file specified by its file number. An optional record number may be included to specify a particular record in a random-access file to begin reading. A variable list following the file number (and optional record number, if included) defines where to put the information being read. For example,

```
)READ# 7; Pip1, Pip2
)READ# 8, 54; Twelve%, Strong&(2)
```

One line of data is read for each variable in the list. READ# automatically performs any necessary type conversions as needed for numeric data. However, type conversions are not automatically performed between numeric data and string variables (and vice versa), and an attempt to read a string with a numeric variable (or vice versa) results in a

?TYPE MISMATCH ERROR

message.

The following table defines the conversion limits of the READ# statement:

Variable	Data Field Type	Result
Real	Real	OK
"	Integer	OK
"	Long Integer	OK; Possible loss of accuracy
"	String	?TYPE MISMATCH ERROR
Integer	Real	OK in the range ±32767
"	Integer	OK
"	Long Integer	OK in the range ±32767
"	String	?TYPE MISMATCH ERROR
Long Integer	Real	?OVERFLOW ERROR if greater than ±E18
"	Integer	OK
"	Long Integer	OK
"	String	?TYPE MISMATCH ERROR
String	Real	?TYPE MISMATCH ERROR
"	Integer	?TYPE MISMATCH ERROR
"	Long Integer	?TYPE MISMATCH ERROR
"	String	OK

Table 5-3. READ# Conversion Limits.

The message

?TYPE MISMATCH ERROR

is displayed if you attempt to use the READ# statement with a file that is not a data file.

If a record number is specified, then the value in the specified *record* in the file is assigned to the first variable in the READ# statement.

WRITE#

WRITE# sequentially writes the value of each item in its expression list to a field in a specified data file. You may optionally follow the file number with a comma and an arithmetic expression specifying a record number to begin access at. The list of expressions must follow the file number (or optional record number), and the expressions in the list must be separated by commas. For example,

```
)WRITE# 3; Major%, Minor%, Xlow
)WRITE# 4, 11; Map(1,3 ,5,7,9)
```

One line of data is written for each expression in the list. WRITE# performs no numeric to string type conversions while transferring information from the expressions to the file, it just writes a binary image of numeric data to the file.

If a record number is specified, then the value of the first expression in the expression list is written to the first field in the specified record. Otherwise, records are written sequentially.

If a file record lacks enough room for all the fields being written to it, the extra fields will be written to the next record. Note that writing any new data to a record will cause the old data in that record to be lost.

If you try to write a data field to a file that is longer than the file's record length, an

?OUT OF DATA ERROR

message is displayed.

Note that the usual rules for expression type are in effect except that an integer is written as an integer only if an integer variable is specified and is not part of an expression. For example:

```
)WRITE# 1;N%
```

writes an integer.

```
)WRITE# 1;N%+1
```

writes a real.

Sequential Access

There are two ways to access text and data files on a disk: *sequential access* and *random access*. Sequential access is like reading a book; accesses begin at the front of the file and continue on toward the end. Random access requires that the file be made of equal sized records, and allows access to be made of any record in any order. Character devices may not be accessed randomly but block type devices, such as disks, allow either form of access.

Text Files

Here is an example of a sequential access program:

```
10 REM Program PrintSequential
20 File$ = "SequentialText"
30 CREATE File$, TEXT
40 OPEN# 1 AS OUTPUT, File$
50 FOR X=1 TO 10
60 PRINT# 1; "This is line ";X
70 NEXT X
80 CLOSE# 1
90 END
```

This little program writes both string and numeric values into a sequential text file.

Line 20 assigns the string "SequentialText" to the string variable FILE$. Line 30 (which is optional) creates a file using the value of FILE$ as a partial pathname.

Line 30 is optional since even if the file does not exist, the OPEN# in line 40 would create a new file. In that case, however, you could not specify the file type explicitly. If a text file named SequentialText already exits on the volume in use, line 30 does not need to be used. Line 40 opens the file and assigns the number 1 to it as a file reference number. As long as the file is open, it is referred to as "#1".

Lines 50 and 70 define a loop that will execute 10 times. Each time through the loop, X has a different value: first 1, then 2, and so on up to 10.

Line 60 writes two values into file #1 each time it executes. The first value is the string "This is line " and the second is the numeric value of X; automatically converted to a string. These two strings are joined together (because of the semicolon between them) to occupy one line of text in the file.

Line 80 closes the file. In a larger program, other routines might need to access files, and unless closely controlled, problems arise with attempts to operate with more than 10 files open at one time or by accessing the wrong file.

After the program runs, the contents of the file are:

```
This is line 1
This is line 2
This is line 3
      .
      .
      .
This is line 10
```

To see the contents of this file on the screen, you need another program:

```
10 REM Program InputSequential
20 FILE$ = "SequentialText"
```

```
30 OPEN# 1 AS INPUT, File$
40 ON EOF #1 GOTO 80
50 INPUT# 1; Accept$
60 PRINT Accept$
70 GOTO 50
80 CLOSE# 1
90 END
```

InputSequential opens the file to read its contents. Each time the loop in lines 40 through 70 executes, line 50 reads the next line of text from file #1, and stores it in the string variable Accept$. Then Line 60 displays the string on the screen.

Data Files

The file SequentialText was a text file because of the way it was accessed. We can use a data file to achieve the same result. The following program creates a data file and writes some data into it:

```
10 REM Program WriteSequential
20 File$ = "SequentialData"
30 CREATE File$, DATA
40 OPEN# 1 AS OUTPUT, File$
50 FOR X=1 TO 10
60 WRITE# 1; "This is line ",X
70 NEXT X
80 CLOSE# 1
90 END
```

WriteSequential is like the program PrintSequential, but uses WRITE# instead of PRINT#. WRITE# does not allow the use of semicolons to separate values that are written to a file. Each WRITE# sends a complete line (or field) to the file for each variable in its argument list. WRITE# also does no conversion of numbers to strings, but places them in a file using the same format as numeric types are stored in the Apple's memory.

In the data file SequentialData, every other field contains the string "This is Line", and the fields in between contain binary coded numeric values from 1 to 10.

The program below reads data from the file SequentialData back into memory and displays it on the screen:

```
10 REM Program ReadSequential
20 File$ = "SequentialData"
30 OPEN# 1 AS INPUT, File$
40 FOR X=1 TO 10
50 READ# 1; Accept$,Innum
60 PRINT Accept$; Innum
70 NEXT X
80 CLOSE#1
90 END
```

Where the program InputSequential used INPUT#, ReadSequential uses READ#. The READ# statement reads the values of two fields from the file: the first (a string value) is stored in ACCEPT$, the second (numeric) value is stored in INNUM. In line 60, PRINT displays the two values as one line using the semicolon.

ReadSequential displays the following on the screen:

```
This is line 1
This is line 2
      .
      .
      .
This is line 10
```

In the simple examples given above, either text or data files work about as well. You will find in most real-world programming that text files are most convenient for handling string-type data, while numeric data is best handled with data files.

Remarks

Notice that the programs above with PRINT#, INPUT#, WRITE#, and READ# statements all used sequential access; we never had to specify where to begin in the file. PRINT# or WRITE# statements cause one access for each expression in its expression list, and an INPUT# or READ# causes one access for each variable in its variable

list. BASIC automatically advances by one data or text item each time an expression is written, or a variable read, so that the next access to the file will be positioned correctly.

Sequential Access Rule 1

When a file has been opened, the first access begins at the beginning of the file. Each subsequent access begins where the last one left off.

According to Rule 1, each time an existing file is opened and written to, at least some of the original content will be written over. If some of the original file is not written over, where is the end of the file? At the end of its original content, or the end of the new content? The answer is that you can't be sure. If the old content has been fully overwritten, all old content is lost. If not, a portion of the old contents will remain after the end of the new contents, and BASIC will not tell you either way. Moral: don't open an existing file AS OUTPUT for sequential access. Instead, delete the old file, then open a new file AS OUTPUT using the same file reference number. (Before you delete the old file, be sure you read any information you need from it.)

If you just want to append information on the end of a file, it can be opened AS EXTENSION, allowing you to write additional information beginning at the end of the existing file. This allows you to retain information previously saved in the file.

Sequential Access Rule 2

When an existing file has been opened AS EXTENSION, the first access begins at the end of the file. Each subsequent access begins where the last one left off.

Random Access

Random-access files are structured as a sequence of *equal-sized* records. A record is a sequence of bytes storing data or text. In random access operation, you specify exactly where each file access should begin by specifying a record number in the PRINT#, INPUT#, WRITE#, and READ# statements.

The CREATE statement allows you to specify the record size of a new file. If you don't specify it, the record size defaults to 512 bytes. You can never change the record size of an existing file.

Note that in sequential access, the record size is irrelevant; you never need to think about it. A sequential file may not be successfully accessed in a random access fashion.

In a data file, the content of a record is organized into fields. Each field contains either a numeric or string value. Any given field is always contained wholly within one record, there will be no overlapping.

In a text file, each record is a series of bytes; each byte containing a character.

To use random access, you must have a clear idea of how information will be organized in your file. In a data file, you should usually plan to have each record contain the same kind of fields: for example, each data record might contain two real values, an integer, and a string, in that order. The record size must be large enough to contain all of the fields you will write in each record. The following table shows how many bytes are used by each kind of field:

Data Type	Bytes Used
integer	3
real	5
long integer	9
string	string length + 2

For text files, if you should want to use random accessing of the text, bear in mind that INPUT# always reads one line of text. Therefore you should usually plan to have each record contain one line. This means the record size should be at least big enough to contain the longest line your program will ever write into it. A line requires one byte for each character in the line, plus one for the return at the end of the line.

Here are the essential rules of random accessing:

Random Access Rule 1

When a record number is specified in a file I/O statement, the access begins at the beginning field of that record.

Random Access Rule 2

When you overwrite any part of an existing record using WRITE# (not PRINT#), all of the previous content of that record is lost. But an existing record that is not overwritten remains unchanged.

Random Access Rule 3

Data Files: If a READ# statement contains more than one variable or a WRITE# statement contains more than one expression, then the access will move from one field to the next.

Text Files: If a PRINT# statement writes more than one line, each is placed in the file in the order written, regardless of record boundaries. If an INPUT# statement reads more than one line, it assumes that each one begins where the last one left off, regardless of record boundaries.

Random Access Rule 4

Data Files: If a field won't fit in the space remaining in the record, BASIC goes to the beginning of the next record. But if a field is too big to fit in any record, the

?OUT OF DATA ERROR

message is displayed.

Text Files

You can change the sequential access examples above to random access, so that when you display the file on the screen, you will display only the even-numbered lines.

To change program PrintSequential to program PrintRandom, you must specify a record size in the CREATE statement and a record number in the PRINT# statement:

```
10 REM Program PrintRandom
20 CREATE "RandomText", TEXT, 16
30 OPEN# 1 AS OUTPUT, "RandomText"
40 FOR X=1 TO 10
50 PRINT# 1,X; "This is line ";X
60 NEXT X
70 CLOSE# 1
80 END
```

It isn't necessary to specify the record size as 16; it could be bigger. But the file would contain wasted space if the record size were bigger than 16. Sixteen bytes is enough to contain the longest string we will be writing, namely "This is line 10"—15 characters plus one for the return at the end.

In line 50, X is the record number. Each time through the loop, the PRINT# statement writes to a different record: first to record 1, then record 2, and so on, up to record 10.

In every other respect, program PrintRandom is like program PrintSequential.

To change program InputSequential to program InputRandom, we don't need to specify the record size; BASIC stored the record size with the file when the file was created. When the file is opened, that information is retreived.

You need to make two changes: a change in the FOR statement, and a change in the record number in the INPUT# statement.

```
10 REM Program InputRandom
20 OPEN# 1 AS INPUT, "RandomText"
30 FOR X=2 TO 10 STEP 2
40 INPUT# 1,X; ACCEPT$
50 PRINT ACCEPT$
60 NEXT X
70 CLOSE# 1
80 END
```

The FOR statement now starts with X=2 and has STEP 2, causing X to take on the values 2, 4, 6, 8, 10 as the loop repeats five times. In the INPUT# statement, we specify X as the record number; so the first time through the loop we access record 2, the second time record 4, and so on, up to record 10.

When InputRandom is RUN, it displays the following on the screen:

```
This is line 2
This is line 4
This is line 6
This is line 8
This is line 10
```

In the same fashion, program WriteSequential can be changed to a program to randomly access data files. Try it!

File Functions and Statements

Business BASIC contains statements that allow you to control program execution according to information contained in files that your program accesses.

ON EOF#

ON EOF# is used to force BASIC to allow your program to control what happens if BASIC reads past the end of a file. EOF stands for End Of File. When ON EOF# is not in effect and BASIC reads past the end of a file, an

```
?OUT OF DATA ERROR
```

message is displayed, and execution halts. ON EOF# is very similar to the ON ERR statement, except that ON EOF# recognizes only one error code.

A statement or statement list must follow the reserved word EOF#. Execution branches to that statement list whenever an

```
?OUT OF DATA ERROR
```

occurs, and BASIC does not display an error message or halt execution. For example,

```
)100 ON EOF# 1 PRINT "End of file"
)1000 ON EOF# 2 GOTO 2000 : IF A<47 THEN NEXT X
```

The statement list is executed as though a GOTO statement had caused execution to jump there. Unlike ON ERR, you cannot use RESUME in conjunction with ON EOF# statements.

OFF EOF#

The OFF EOF# statement cancels an ON EOF# statement. After an OFF EOF# statement has been executed, BASIC resumes displaying error messages and halting execution when an end of file is reached, just as it did before the ON EOF# statement was executed. You must follow the reserved word EOF# with a file reference number to specify which file's ON EOF# statement should be canceled.

The Reserved Variable EOF

When BASIC encounters an

```
?OUT OF DATA ERROR
```

it assigns the file reference number of the file causing the error to the reserved variable EOF. You can then check the reserved variable EOF to determine the affected file.

When you use the *reserved* variable EOF in an ON...GOTO or ON...GOSUB statement, you must enclose EOF in parentheses. For example,

```
)ON (EOF) GOTO 100,200,300
```

TYP

TYP is used to determine what type of data will be read from a particular file on the next access to that file. The argument to the function can be any arithmetic expression, but its value must specify a particular file reference number. The number returned by the TYP function denotes what type of data will next be read from the specified file. For example,

```
)ON TYP(3) GOSUB 1000,1200,1400,1600,1800,2000
```

For a data file, TYP returns the following values:

Value	Meaning
0	File type indeterminate
1	Next datum is Real
2	Next datum is Integer
3	Next datum is Long Integer
4	Next datum is String
5	End of file

For a text file, TYP always returns the value 8. If there are no more characters in the file, the value returned is 5.

If the type of a file is as yet undetermined (i.e., whether it is a data or text type) a zero value will be returned for the TYP function.

If the value of the argument of TYP is not between 1 and 10, the

?ILLEGAL QUANTITY ERROR

message is displayed. If a file with the specified reference number is not open, the

?FILE NOT OPEN ERROR

message is displayed.

REC

REC returns the current record number of the file specified by the value of the arithmetic expression following the reserved word REC.

If you use the INPUT# or READ# statements to access the catalog of a directory, REC returns the number of the line currently being accessed.

REC has the same error conditions as the TYP function.

File I/O Example

Here is an example of file use. Assume that you have inserted a BASIC diskette named APPLE1 in disk drive 1, and turned on the power. Assume that there is no initialization file (no HELLO program) so that no program is run automatically; you just see the prompt character. Now you type RUN DEMO to load and run a program called DEMO, designed to make copies of existing files and shown below.

```
10 PRINT "Text file copy utility"
20 INPUT "Type input file pathname: ";A$
30 OPEN #1 AS INPUT, A$
40 INPUT "Type output file pathname: ";A$
50 REM Open new output file
60 OPEN #2 AS OUTPUT, A$
70 ON EOF #1 PRINT "Done" : CLOSE : END
80 INPUT #1; A$ : PRINT #2; A$ : GOTO 80
```

Line 10 displays a message on the screen. Line 20 displays a prompt and then waits for you to enter a pathname. The line entered must be a legal pathname or an error message results, and the program halts. Line 30 opens the named file, to be referenced hereafter as #1. Line 40 asks for another pathname; and line 60 opens the file for output, assigning to it the file reference #2. Line 80 performs the actual copying. Line 70 will be executed only when the end of the input file has been reached to end the program. Until line 70 is executed, line 80 reads in lines from the input file, and writes them to the output file.

Recall that when you boot your Apple, the volume name of the diskette in the built-in disk drive is stored in the reserved variable PREFIX$. This means that a valid pathname can be as little as the name of the file (assuming that there are no subdirectories on the diskette). So responding simply Afile and Bfile to the program prompts would cause /Apple1/Afile to be duplicated to a new file named /Apple1/Bfile. Remember that not using a slash before a pathname causes the contents of PREFIX$ to be added to the beginning of the pathname.

This program can be used to print a file by responding Afile and .PRINTER, assuming that a printer is properly connected. Responding .CONSOLE and /Apple1/Text will take input directly from the Apple's keyboard and write it to the new file /Apple1/Text created by the program.

Note that in the last case, the program will not terminate normally, because these block devices (.CONSOLE, .PRINTER) have no end of file! In the next-to-last case, .PRINTER is used for output, so EOF will be triggered when the end of input file is reached. Use CONTROL-C or RESET to end the program.

6

External Routines

6

External Routines

Introduction

Apple Business BASIC allows external subroutines (assembly-language procedures or functions that are not part of BASIC) to be loaded and executed from BASIC programs. You can use both external subroutines that you write, and those included with your Apple. For example, there is a library of special purpose subroutines for displaying graphics supplied with Apple Business BASIC, described in the appendix on the invokable Graphics Module, that can be used by your programs.

If you catalog your Business BASIC master diskette, you will see some files with the extension ".INV". These are external routines. There are two other groups of related files that you should be aware of, both having the same name as the external routines files: One with the extension ".DOC" (for documentation), and one with no extension, which are demonstration programs.

If you run a program with the .DOC extension, it will describe the use of its associated external routines. The program file with no extension demonstrates program use of its associated external routine.

A subroutine is a separate part of a program called by one or (usually) several other parts of the program to perform a specialized or frequently-repeated task. A subroutine may have a list of arguments enclosed in parentheses following its name, either variables, pointers, or expressions. There are two types of external subroutines: procedures and functions. A function returns a value, while a procedure does not.

An external subroutine uses its argument list to communicate with the calling program as to what information is available for its use, what is to be operated on, or where it should leave the results of its operation for the use of the calling program.

Since internal BASIC routines (called with a GOSUB statement) have access to constants and variables used by the program in memory, they don't require the passing of arguments that external routines require.

Four BASIC statements (*INVOKE, PERFORM, EXFN.* and *EXFN%.*) are needed to use external subroutines. These statements are used to load a file containing external subroutines into memory from disk files and execute them at the BASIC program's demand.

The INVOKE statement loads external subroutines and, depending on the subroutine's type, either the PERFORM, EXFN. , or EXFN%. statements execute it. These statements are described below.

INVOKE

INVOKE loads specified assembly-language subroutines into memory. For example, to load a subroutine named FastPrint, enter

```
)INVOKE FastPrint
```

You may load as many subroutines at once as you like by separating each file's pathname by commas:

```
)INVOKE FP1, FP2, /Floppy2/Subr/FP3
```

After loading, subroutines are executed from your BASIC program with the PERFORM, EXFN%. , or EXFN. statements described below.

Executing INVOKE effectively erases any external subroutines previously loaded by other INVOKE statements and returns any unused memory to BASIC.

If you don't need your external subroutines any longer and want to make the memory available to BASIC, execute an INVOKE statement with no pathnames following it. Only the invoked subroutines are removed from memory, variables defined in BASIC are not touched, nor is the main program altered.

If there is not enough memory to load an invoked file, an

?OUT OF MEMORY ERROR

occurs.

If the file loaded is not an executable assembly language subroutine, a

?TYPE MISMATCH ERROR

message results. If the file is not found on the named diskette, a

?FILE NOT FOUND ERROR

message results.

PERFORM

PERFORM executes a specified assembly language procedure previously loaded by an INVOKE Statement. If an argument list is present (enclosed in parentheses after the procedure name) each argument is evaluated and passed to the procedure before execution.

To pass real numbers or the values of single variables, just include them in the argument list as an expression (for an explanation of expressions see the section EXPRESSIONS AND STATEMENTS in the chapter Tools of Your Trade).

To pass integer values, precede the expression with a percent (%) sign. If you want to pass a long integer value, precede the value with an ampersand (&). For example,

PERFORM StrangeRites(&Pennies, %Accountants)

If no leading character is supplied, real number values will be sent.

To pass *addresses* of variables, precede the variable name with an at sign (@). For example,

```
)PERFORM Errproc(R,13-6,@D)
```

passes the value of variable R, the value 7, and the address of the variable D to the procedure named Errproc.

If you want your subroutine to operate on a string in memory, using the @ sign gives an address pointing to the string's descriptor in memory. The subroutine should be designed to act on the string from that point on.

Values passed to an external subroutine are pushed on the system stack in memory. When the routine is executed, it must read the values from the stack.

Addresses of variables to be passed to an external subroutine are pushed on the system stack by BASIC *only* if the variable name is preceded with an @ sign. It is the responsibility of the subroutine to distinguish between variable values and addresses. Only single variables can be preceded by an @ sign in this manner; using an expression is not legal.

Let's say we have two subroutines that each take one argument. The first one, MyProc, takes the value of a real expression. The other one, MyOtherProc takes the address of a real variable. Various legal and illegal combinations of arguments to these subroutines are:

```
)PERFORM MyProc(4.5) : REM Legal: A simple expression.
)PERFORM MyProc(NUMS) : REM Value of NUMS is passed
)PERFORM MyProc(NUMS+4.5) : REM NUMS+4.5 is legal
expression.
)PERFORM MyOtherProc(@NUMS) : REM The address of
NUMS is passed.
)PERFORM MyOtherProc(@NUMS+4.5) : REM Illegal
operation
)PERFORM MyOtherProc(@4.5) : REM Illegal address
```

Arguments that are variable addresses (as opposed to variable names or values), must be preceded by the @ sign; otherwise the executed procedure will be passed the *value* of the variable instead of its *address*.

If the number of bytes of the arguments supplied to a procedure does not match the number of bytes of the arguments that the procedure expects to be given, a

```
?TYPE MISMATCH ERROR
```

message is given. All BASIC knows about the argument list to be passed to the procedure is how many bytes of memory the evaluated argument list should fill.

EXFN.

EXFN. executes an external assembly language function that returns a real value and has been loaded by an INVOKE statement. For example, suppose you have a function named CalcX that performs some operation on a supplied argument and returns the result of the operation to the caller. You could execute CalcX in immediate mode with the following statement:

```
)PRINT EXFN.CalcX(2)*32/256
```

The value returned by CalcX is multiplied by the expression 32/256. Remember that the argument passed to CalcX is contained within the parentheses following the function's name.

The rules for passing arguments to external procedures also apply to external functions. To pass the *address* of a variable, precede its name with an @ sign. Values of variables or numbers to be passed must be given in expressions.

Like PERFORM, if the number of bytes of argument that you supply differs from the number of bytes the function expects, a

?TYPE MISMATCH ERROR

message is displayed.

Also like PERFORM, an

?UNDEFINED FUNCTION ERROR

occurs if the function named was not part of a currently invoked file.

Remember, if you want to pass an integer argument, you must precede the expression being passed with a percent sign.

EXFN%

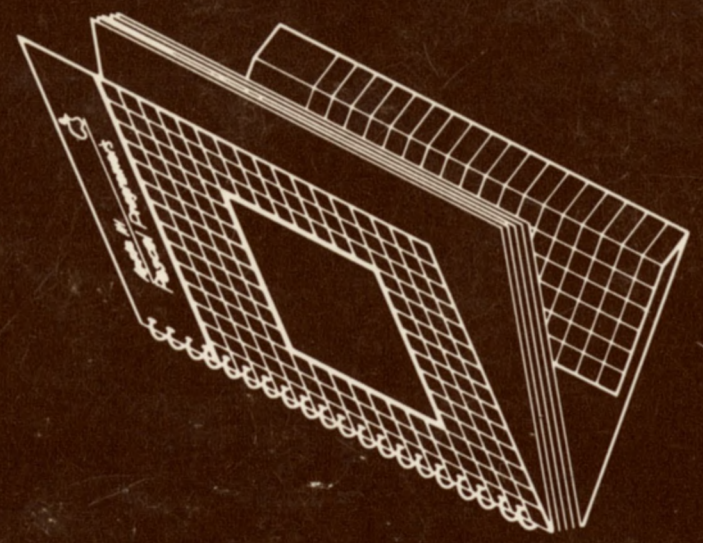
EXFN%. is identical to EXFN. except that the assembly language function involved returns an integer instead of a real value. To pass the address of a variable, precede its name with an @ sign. Values of variables or numbers to be passed must be given in the form of expressions.

Apple III

Apple Business BASIC

Reference Manual - Volume 1

Tuck end flap
inside back cover
when using manual.



apple computer

10260 Bandley Drive
Cupertino, California 95014
(408) 996-1010

030-0122-B